Story of the VAX Strategy: Gordon Institute Talk Given on 9/5/87

VAX Project 4/1/75

Product Position at the time: evolution of 10 vs 11. Also PDP-8.

Goals and Constraints were important incl. 1000:1 range using lots of techniques (VLSI, mP, Tandem).

Every existing group (who reported to me) thought they were "safe" from VAX (8,10, 11). VAX was initially a very high end 11, but still a "mini" because it was to be operated by users not a staff.

VAX Strategy occured later: 9-12/78. BOD Approval 12/78.

We were ready: machine, technology, organization, direction

An initial, bet it all strategy, would have killed the project/idea.

What was known when VAX was started

A model of the entire market. Requires understanding... historians (cf IBM)

Levels-of-integration;

Machine classes and user styles (supers, mainframes, minis, micros, etc.) for central, distributed, and personal computing (GB '75)

IBM family idea and how VAX was different to cover range of styles

The technology(t) and where you fit and need to be, who you need to use

Memory drives it... a discovery. Memory addressing is the mistake.

Perhaps there's a corporate strategy for what you want to accomplish:

make money, nicher, whole market, best computers, segment for org., lowest cost producer, most expensive and powerful, honorable, fun,

technical marketplace, spread entrepreneurism, make great products,

... DEC had all these. Mostly a "produce driven" company. No overall vision, but rather visions around the PDP-8, 10, and 11 products/mkts

How was it communicated?

Clear, one page statement communicating VAX Strategy, plus 14pp:

Essence and Rationale of the Strategy (1)

How Can We Win Against IBM (1), and others (1)

Products in 81-82 timeframe, including a tree (4) plus a block diagram showing how all of the products interconected

Five pages of Answers to: Why Change the Current Strategy, Why Not Aggressively Evolve all Four Base Hardware Architectures, Why not segment products by market, how do customer perceive the situation, Why have a single architecture for hardware and software, Why base the architecture on VAX, Why not use the 10/20 as the base, Why distributed computing

How was the Strategy Managed?

When the VAX Strategy was put forward, many projects changed! This was useful, simply to tell the organization it wasn't "business as usual" or any computer for any

engineering group x marketing group. (Shift from hi end 11's, eventual elim. of 8 and 10, move to applic. instead of evolving about 12 operating systems.)

The Planning process was organized around the strategy. A very small staff met informally with me re. fleshing out the details. A large architecture group, distributed with the development groups managed all the myriad of standards.

Each year, a set of plans were published that tied into an Engineering Overview which I maintained, and authored. The overview "graded" every group, revised the strategy and/or tactics as necessary to encompass changes in products and external shifts in the markets including the shifts to PC's.

Like all great architectures, I believe the vision, motivation, and drive has to come from a single individual.

<u>What was good about VAX?</u>

It was incredibly simple-everyone could remember it,
everyone had a role
Low risk from where DEC was, incredible upside
It was able to encompass 8, 10, 11. It was compatible.
It included shift to VLSI... which should begin to affect IBM.
Everyone else was so bad

Why it was easy to get IBM... they had more history: machines, operating systems, technology (gate arrays) was evolutionary and couldn't get there with VLSI, lack of mP,

UNIX was perceived/understood as the "competition". It hasn't been as much as it could have been with a real driving force.

It did exploit IBM weakness. It took market share in commercial.

<u>What was wrong with the strategy?</u>

Address space too small. Will have to get to >32 bits by 90
RISC: VAX architecture should not have been defined at hardware level
Reliance on a central group ... VMS, for too wide a style range.
Still no significant mP's!
Venus (8600) was late, Ethernet took too long and risked strategy,
PC's came in and still DEC ain't doing it right.
Ken drove trilogy (3 PC strategy... losing a 1B; a bargain)
Didn't manage change in technical computing. The understanding was present.
<u>Cray 1 for <1m by 85.</u> Alliant and Convex did it.
The organization couldn't build workstations-- hence Apollo, Sun
These together amounted to a major loss in market share
and can never be recovered.
Losing technical market is sure fire road to decay.

GB 9/5/87

Beyond VAX: What's next for DEC

Computerworld interview with Gordon Bell.

Draft 6/30/87.

What is the origin and essence of VAX?

VAX came from a tiny task force I lead in April 1, 1975. It was a new computer family to be "culturally compatible" with the successful PDP-11. It's principle design goals were: compatibility with key operating systems and languages; having a much larger address space than any existing computer; being efficient at implementing high level languages including Fortran, C (for UNIX), and Cobol; being implementable over a wide range of sizes; and simply being the highest performance computer in its class when first implemented.

In December 1978, after the 780 had achieved instantaneous success, the company adopted the VAX Strategy to provide a VAX Homogeneous Computing Environment for a range of interconnected computers. A user could compute in any of three styles from a cluster of large machines behaving as a single system, distributed traditional minicomputers, and distributed clusters of workstations. The Strategy also specified compatibility with other DEC computers and intercommunication with other standards and products.

Why has VAX been so successful?

The concept was incredibly simple, and hence everyone (customers and the company) could understand and support it. Also, the 3-level computing hierarchy was right ... even IBM discovered and endorsed it by the early 80's. VAX provided the best and only, totally compatible, single interconnected environment. This required a range of computers from VAX on a chip, to the highest performance computers that could be built. It gave DEC a product monopoly since no other manufacturer had (has) anything like this capability. It specifically exploited the fact that most manufacturers had a menagerie of product lines so designed to segment the user base, fill product size/application gaps, or help the manufacturer's organization.

Recently IBM started to provide similar capabilities by having 370-compatible minis and a plug-in card for a PC, but this is hard because they have several operating systems, a worse problem than having multiple hardware

architectures. Also, given the complexity of the IBM architecture, including the I/O and operating systems, it's probably hard to make the architecture serve the wide range at this point in its life.

Did things happen pretty much according to your VAX Strategy?
In the large, yes, although it wasn't as trivial to do as one would think. Ethernet (an essential component) was questioned by various committees even after the whole system was working. Having adopted a strategy of VAX in '78, in '80 the organization decided it had to enter the PC market with a "trilogy" of non-VAX PC's, which only loosely fit the strategy. VAX was too large to build as a workstation until 1982-84. At the same time, the high end implementation of VAX, Venus (the 8600) was over two years late as engineers hit the compexity wall, and essentially forgot the receipe of how to design computers. These two events account for DEC's poor financial performance in the early 80's.

Do you see anything around to challenge the VAX Strategy yet?
No. In '78 I thought the only possible threat was UNIX because it provides compatibility at a higher level somewhat like VAX. I imagined that innovative or small companies would develop UNIX systems for interconnected computing environments by the mid-80's. Now, I'll push that back 3-5 years. A critical hole is in the PC space where MS/DOS is like UNIX, but isn't. UNIX needs to evolve in range, human interface, and applications. Having AT&T control it doesn't help --it has to be a really public standard. The government support of UNIX (Posix) still could have an impact.

Also, I don't see a single large computer company coming up with anything like VAX because of the cost and committments to preserving their code museums for running old programs.

How far can DEC go with the VAX architecture?
I don't believe all the capabilities in the architecture as constrained by its addressing have been exploited yet. DEC still has uniqueness.

Critics point out, even if clustered the architecture will sooner or later top out. What do you think?

Here, history is a good guide. Every architecture has sooner or later either run into a limit or been inappropriate to the technology. With the exception of the IBM 360, which had an inherent 32-bit address to get it to the 90's, history also tells us that companies try to evolve their architectures too long. They end up with 100% of their user base market, but a declining share of the entire market. Eventually even that market declines as users desert the obsolete machines.

A major computer technology generation lasts about a decade. I believe it is hard to design an optimum architecture that lasts much longer. While VAX may top out, it should be a fine base for evolution.

What will this mean to the thousands of sites committed to DEC's single architecture prescription?

Again, let me rely on history. VAX was a major new hardware architectural evolution from the PDP-11, yet it preserved programming interfaces, languages, and databases. The same concept could be reapplied even if DEC changes the underlying hardware architecture. The program and database interface must be preserved--in effect, it should be transparent if users adhere to certain VAX/VMS standards.

It's probably important to define VAX (or VMS) compatibility, and whether a new, basic hardware architecture could be used to implement this environment, that is, without object compatibility. The problem is much easier than with the PDP-11 or with the 370 because VMS is a single interface which subsumes the network, but includes the Command Language, DCL, and various languages. Fortunately, nearly all programs are written in a high level language today and would be compatible. This sort of interface is being identified by IBM for its applications program environment.

Can DEC engineers develop a totally new architecture for the 90's and beyond that will play on DECnet and run software from existing machines?

Companies with different, underlying hardware architectures provide existence proofs of VMS user level compatibility. Certainly DEC should be able to do this too.

Are they at work on such a scheme now and if so, what is the best guess as to what it is?

Hopefully, because it is essential. The key is to identify the key limits of VAX and to eliminate them. It again might have goals similar to those we used to create VAX in the first place. The only goal I would add to the original VAX set would be the independence of the ISP (Instruction Set Architecture) hardware architecture. Just as VAX added new dimensions of comparision, it must.

I would hope it addresses parallelism of all forms and performance for the scientific and engineering community, including the ability to collaborate effectively via the computer using very high speed interconnects (LANs, Campus Area Nets-CANs, and a revolution in Wide Area Nets). It would handle very large scientific and engineering databases. For example, today we see VOXEL data sets of over 4 gigabytes (1K x 1K x 1K x 4 bytes).

A radical view of data integrity and databases is also needed. Improvements in the cost of ownership and availability dimensions are quite possible. In addition, DEC could address the mass market for users who want a great computing environment, but don't want to become system programmers or administrators. This would rule out any compatiblity with MS/DOS and the PC! (The PC has allowed everyone to relive and retrace computing history and to become system adminstrators with all the acroutrements, including large manuals. I'm happy to avoid this trip back to the 70's; I use a MAC).

You've made several comments about needing higher performance VAXes. What is the biggest VAX you can build?

There are two basic measures of performance: total processing power available to a single job stream, i.e. throughput; and power available to a single job. For the former, VAX clusters partially provide this, but multiprocessors extend the range even more and in a more cost-effective fashion. Furthermore multiprocessors are starting to use parallel processing to provide speedup of a single job. This can be done either by the compiler or the user.

DEC should have already introduced a significant multiprocessor with dozens of micro-processors, ie. a "multi" like Encore, Masscomp, Sequent, Stratus and others. VMS as a multiprocess operating system shouldn't be the limit. By

using the CMOS microVAX, over 100 mips could be put in a small box. This approach would provide at least one or two non-me-too products. Moreover, it gets the price into the $10K/MIP range versus the $100K-$200K range typical of the large mainframe. These ridiculous prices aren't sustainable except for large mainframes where users are locked-in to buying code museums -- and someday the users may get smart.

What markets would such a machine address?
DEC seems enamoured with the commercial and transaction processing markets. Multi's are the best computers for these markets because the applications only demand total mips for a large collection of jobs. The system has advantages for a general, interactive job stream such as program development as demonstrated by the "multi" suppliers.

The microprocessor inherently provides the best performance/cost by almost two orders of magnitude (we simply look at the mips/chip). By ganging them, and matching them to a memory, one can get the most power in a single system at a small fraction of the cost of an ECL-based computer with a few, expensive processors. It also has inherently much better availability characteristics.

Would such a system address all the concerns you have about inadequate scientific and engineering performance?
Not entirely, but two "multis" could replace an entire product line and provide 100 mip level performance, and substantially better price/ performance for the user than the "model" approach.

In addition, consitstently competitive compute servers are needed which would run technical work in the CRAY 1 speed range. In the long run, a "multi" might do the job, but for now, the vector multiprocessor is the main line... in effect, another Crayette.

What's the largest uniprocessor VAX that can be built?
The speed of a uniprocessor such as the VAX, or a 370 is correlated with the clock speed. A high end machine with a 40-60 Mhz clock could probably be built and still be in the mini price range with a power of 2-3 times the current

models. Note the current 3090 uses about a 60 Mhz clock and the Cray XMP clock is almost twice as fast, although both have roughly the same scalar speed. Clock speed isn't always a good indicator.

Could you look retrospectively on what DEC might have done in the four years since you left?
Let me provide my own reference point first. I've been involved with a bunch of new computers, three of which are on the market, plus several startups that are creating new markets. All of the computers provide more capabilities than VAXen and the engineering has been done in a small fraction of the time and budget of DEC product designs.

Aside from the evolutionary extensions and products, I would have probably urged for greater innovation and carried on enough experiments to have selected a VAX II architecture by 1986, with benchmarking now, and delivery in '88 a decade after the 780.

Would VAX II be "RISC" based?
Probably.

Given your preference for micros, would you use standard micro?
No, none of them on the market today offer enough. The ones from the merchant suppliers don't offer any more performance than the CMOS MicroVAX.

What other big issues face DEC in the future?
1. Thinking VAX is the end, not simply the best thing around today, is an enormous hurdle. While nothing is in the marketplace to yet challenge it, several new systems do and will. This leads to arrogance. VAX gave DEC a monopoly in much the same way that the 360 gave IBM a product monopoly by the 70's that only lasted a decade. DEC should compare its products with the best small companies, not old line suppliers.

2. Thinking VAX can do it all by itself... or by relying on the old IBM applications being converted to VAXen just because VAX is better than the 360. Radically new applications should be sought that build on the environment and do things

no other environment can support. Also understanding the limits that come from new uses is critical to VAX II.

3. Being enamoured with the commercial interests and not attending to the scientific and engineering base especially in the universities. The commercial market tolerates high prices for higher peformance, but they are unique. The technical markeplace is far more demanding on products.

4. Poor presence on the desk, and even picking MS/DOS and xxx86 to implement. I don't see what another clone brings to the marketplace--certainly not profitability. Service revenue can be obtained simply by going into that segment of the service business. Integration with the MAC is also important.

5. Responsive, efficient, and creative manufacturing still appears to be non-existent. While DEC's probably no worse than the average, it's not adequate to compete in the 90's when the Japanese and others arrive.

6. Lethargy, typical of large monolitic (engineering) organizations. Today their R & D expenditure is almost 1 billion dollars per year. I thought all the really dumb things that caused low productivity were done in my tenure. Given the responsiveness, and lack of innovative products, new engineering management based on leadership and people instead of committees and processes is clearly required. Instead, I find the only person whom I felt capable of running engineering, just resigned.

That's a big set of worries. Are you optimistic?
Certainly. They are making lots of money, have lots of cash, and have exceptional people. All they need is a challenge. The plethora of new startups certainly provide that.

BASIC STRATEGY DO NOT COPY Appendix 1 Page 1
OOD/Gordon Bell
Last edit: 11/17/78 -- Latest edit 1/10/79 - Wed

ID#354 COPY # 95 ASSIGNED TO: Original Product Strategy Document

BASIC PRODUCT STRATEGY

Provide a set of homogeneous distributed computing system products so a user can interface, store information and compute, without re-programming or extra work in many styles and the following computer system sizes:
as a single user computer within a terminal;
at a small, local shared computer system; or
via a large central computer or network.

Achieve a single VAX, distributed computing architecture by 1985 (as measured by revenue)through:
focusing on homogeneous distributed computing with varying computing styles including high availability and ease (economy) of use as the DEC advantage;
building new 11 hardware to fill the product space below VAX;
building new 11 software products that also run on VAX; and
developing software for ll-VAX migration and 11 user base protection.

Provide essential standard IBM and international network interfaces.

Define, and make clear statements internally and to our users about programming for DEC compatibility.

Provide general applications-level products that run on 8, 10/20 and 11/VAX-11 above the language-level to minimize user costs, including:
word processing, electronic mail, and profession-based CRT-oriented calculators; "Visicalc"
transaction processing and data base query; Lisa package
general libraries, such as PERT: simulation, etc. aimed at many professions that cross many institutions (industry, government, education, home); and
general management libraries for various sized business.

Provide specific profession (e.g. electrical engineering, actuarial statistician) and industry (e.g. drug distributor, heavy manufacturer) products as needed via the product line groups.

Provide cost-effective 8, 10/20 systems through:
building hardware that runs current operating systems; and
making market support and DEC-standard language enhancements.

This strategy is intended to cover the full range of DEC's future products. Since technology shifts rapidly and market opportunities emerge that we don't now understand, it may be necessary to provide non-compatible, point products. These should be proposed and reviewed accordingly.

Essence and Rationale of the Strategy

The essence of the strategy is simplicity through adopting a single architecture. This simplicity is needed so that we can build the network and distributed processing structures which our customers are now demanding. The strategy is a" evolutionary result of the 1975 choice to extend the 11 architecture and cover its customer base.

Given that the architecture and early customer acceptance are in place, the strategy moves to build our subsequent products on VAX, while continuing to sell 8's, 10/20's and 11's. Focus is imperative in order to avoid the redundant development efforts across base hardware and software, and to move development to fully distributed computing and to applications. The strategy also minimizes manufacturing and field start-up costs and takes advantage of the learning effect by moving to a single architecture.

The motivations for the homogeneous architecture are numerous and include the customer desires for a range of products on which to build products (in the case of OEMs) and applications (in the case of end users). Such a range in size and over time, allows planning and investment of software and it permits computers to be associated with various organizational units (eg. central group, small group, office, the person, or the home) on a" "as needed" basis. Although, superficially it appears to be possible to have numerous architectures that are segmented by size and by market, the user requirements to cross both size and applications boundaries are significant. I" fact, give" that IBM is segmenting its products both by size and application, the main strength of the strategy is to have a single architecture with which a user can be comfortable rather than bounded by a manufacturer segmentation.

The most compelling reason for basing the strategy on the single VAX architecture, besides the technical excellence of the product is the belief that we can not build the truly distributed computing system of the 80's with heterogenous architectures. It is possible to build distributed computing networks as we do today, but the homogeneous architecture approach insures that programs may be assigned to any "ode, where they will give the Same results. There is no need for the organizational and computation overhead signified by different manuals, separate training, recompilation of programs, and translation of data among machines in the network.

This strategy is aimed at beating the competition using our existing highly tuned minicomputer hardware and software to support and grow our existing user base. It provides us with a unique offering in the marketplace of the '80's which is likely to be based on the defacto standard IBM 360/370 architecture and the ensuing defacto architectures coming from the semiconductor companies. Since VAX is fundamentally better than either of these architectures, we must make it the standard architecture via transition from the PDP-11, which has been the standard architecture of the 70's.

The strategy is aimed at high volume through multiple channels of distribution, versus a more stable, low growth through support of an existing multi-system, customer base.

How Can We Win Against IBM?

A competive viewpoint is the most important check on strategy. Both the recently accounced IBM 8100 Distributed Processsing system and the System 38 computers are the first computers from IBM that, on the surface, look worth owning. They may be as significant as the 360 and their Selectric typewriter. The System 38 with a 48-bit virtual address is technically unique and may offer the user some very large benefits.

The 8100 is a radical departure from IBM pricing as 0.5 Megabytes of primary memory and a 60 Megabyte disk are $ 29 K. A comparable DEC product sells for several times this now. The 8100 is exactly in the price range of the systems we sell and where we make most of our revenue. It is the second product in this price range within a year; the Series 1 minicomputer family patterned after the 11/04-11/34 was the first product. On the surface, the product is low priced, with lots of capability, but it also has a new communications structure (versus the one we have used substantially unchanged since 1961). This structure permits easy peripheral and terminal interfacing for both the office and factory environment. There is a" extensive range of peripherals, terminals and communications to the 360/370. Since the product is sold by DPD, the strategy seems to keep account control and to make the money on the numerous locked-i", generally overpriced terminals.

IBM will 'have: a 3601370 line in the $100 K to $10 M pi-ice range with lots of plug compatible competitors, several operating systems to support, a large backlog, a newly announced 8100 for Distributed Processing around the mainframe; a System 32/34/38 for Distributed Processing and as a Mainframe for small organizations; the Systems 3 to 15 for Distributed Processing; the System 1 for the would-be minicomputer buyer; the 5100-series Personal Computers for the scientist, engineer, analyst and small business; and several inevitable products for computing in the terminal. All of these are incompatible, except for a communications link and the fact that they all use the 8-bit EBCDIC byte. Products are relatively segmented to customer clauses and different languages are used to further segment and hinder application mobility. Finally, they've sold via DPD and GSD, with Office Products no doubt looking on and waiting for a" entry via electronic mail and word processing.

· The IBM PC + IBM Tech. PC
· IBM 802 PC in 1½ yrs
1/83 GB

While on the surface, the 8100 stands to be IBM's most significant product, it seems to be a serious mistake as it introduces another incompatible computer system with which customers will have to deal. This means that the making of a compatible, fully distributed processing system will be essentially impossible. However, since IBM feels it can not move very rapidly in any product space because of the installed base, product options are limited. Hence new products seem to be highly targeted at specific, new non-IBM markets in a" incompatible fashion to get incremental revenue and growth.

They have a series of ~~three~~ random point products which they interconnect in a hodge-podge fashion via SNA. A user has no basis to protect software or share info across systems!
1/83

How Can We Win Against Other Competition?

There are established competitors too, such as DG, HP and Prime. DG and Prime have very simple, single architectures and have been most profitable and have grown most rapidly. HP is converging on a single architecture around the 3000, but it will have to be extended eventually. The NOVA will also be extended. The large manufacturers (Univac, Honeywell and Burroughs) which operate with an established base are less profitable, have grown slowly and have multiple, poor architectures. Honeywell, with a simple, but adequate minicomputer architecture seems to be doing well by selling minis to its old line, mainframe base. There is no evidence that they're developing or pursuing the mainframe business actively.

There are probably more significant threats from the companies that can be easily founded to build systems into disks by using the newly announced zero-processor-cost, 16-bit microprocessors which have 22-bit address spaces and the performance of the 11/34-11/45. All of these architectures need to be extended for multiprogramming and to handle larger virtual memories. High level systems, functionally equivalent to our systems such as RSTS can be built easily and cheaply and can quite possibly target a specific existing, trained user base.

There are also the Japanese and TI which can be lumped together because of their similar behavior. Both believe in targeted, high-volume products with forward pricing. Neither have an adequate architecture. TI is strictly limited to 16-bits with almost no escape, and the Japanese are aimed at copying, using U.S. companies to distribute hardware. It's inevitable that they'll supply IBM compatible 360/370's to the Service Bureaus for distribution. This later channel of distribution is another formidable competitor.

The strategy supports very high volumes for dumb, pre-programmed (smart) and programmable (intelligent) terminals using the 11 until VAX is appropriate in terms of price and functionality. In the mid and high priced minis, the strategy is compatibility and volume, phasing as appropriate from 11 to VAX. For example, since there is not a high priced 11 after the 11/74 and the 11/44, there is a phasing to VAX (through COMET) and lower priced 11's based on 11 microprocessor implementation. The question here will be how fast we can provide high performance microprocessors using HMOS and narrower line VLSI technologies.

PRODUCTS IN 1981-82

HARDWARE COOMPONENTS

HMOS LSI, with first "test" product

Interconnection hierarchy with software compatibility
- 1-10 Mhz and/or 10-100 Mhz inter-computer bus ICCS
- 50+ Khz comm.-compatible multidrop for terminals, peripherals, and small systems;
- 0.3-19.2 Khz comm.-compatible for low cost terminals.

Significant competitive memories
- Solid state modules for software
- Low end floppies and low cost tape
- Removeable and low cost disk RL04
- Hi-volume mid- and hi-end disks in R80/R81 with backup

Terminals for everyone!
- Low cost (dumb) and block mode (VT162)
- Office environment for quality printing, electronic mail, and full-page text
- Professional using graphics (and/or color) with target application software
- Factory environment terminals and interface systems

HARDWARE SUBSYSTEMS

- Remoteable printers, job entry, concentrators, sensor-control
- Communications concentrator - Mercury
- Memory (Hierarchy) Management - HSC50 for R80/R81, RL04, tape and disk cache

KERNEL SYSTEMS based on processor-disk-commlrnications (see family tree figure)

780 replaced by Superstar (const. price >3x performance)
- 780 - Memory Manager - Comm. Concentrator
- 780 - Multiprocessor
- 780 - RP/R80-81 + RL02-04
- 780 - RK/RL04

Comet - RP/R80-81 + RL
Hydra (Including Memory Manager - Comm. Concentrator)

Nebula - R80-81 + RL
Nebula - RL02/RL04 (higher cost, quick to market personal computer)

LSI VAX - RL04 - Graphics Terminal (personal computer)

11/74 with no hi end replacement
- 11/74 - multiprocessor
- 11/74 - RP/R80-81 + RL02-04
- 11/74 - RK/RL04

11/44 replaced by HMOS LSI-11 with >256 Kbytes
11/44 - RP/RB0-81 + RL
11/44 - RL

11/23 - Unibus Fonz RL replaced by HMOS >256 Kbyte
11/22 - Q-Fonz RL
11/22 - Q-Fonz - RX (floppy)
PDT Fonz - RX (floppy)
PDT Fonz - TU58

Tiny chips, replaced by HMOS tiny <256 Kbytes

SOFTWARE

Diminish the 11 software investment for mature products (RSTS, IAS, MUMPS) and provide only minor enhancements to recent 11 based products (TRAX, SCS-11, PDT Software) to extend the market life and limit the VAX transition risk. Orient new development on VAX and 20 toward IBM compatibility and explicitly invest in tools designed to permit easy customer movement between VAX and 20. DEC 20 development will be aimed at high level tools and applications support. Shift the bulk of the PDP-11 software investment to VAX, tracking VAX hardware and aggressively moving to round out commercial capability.

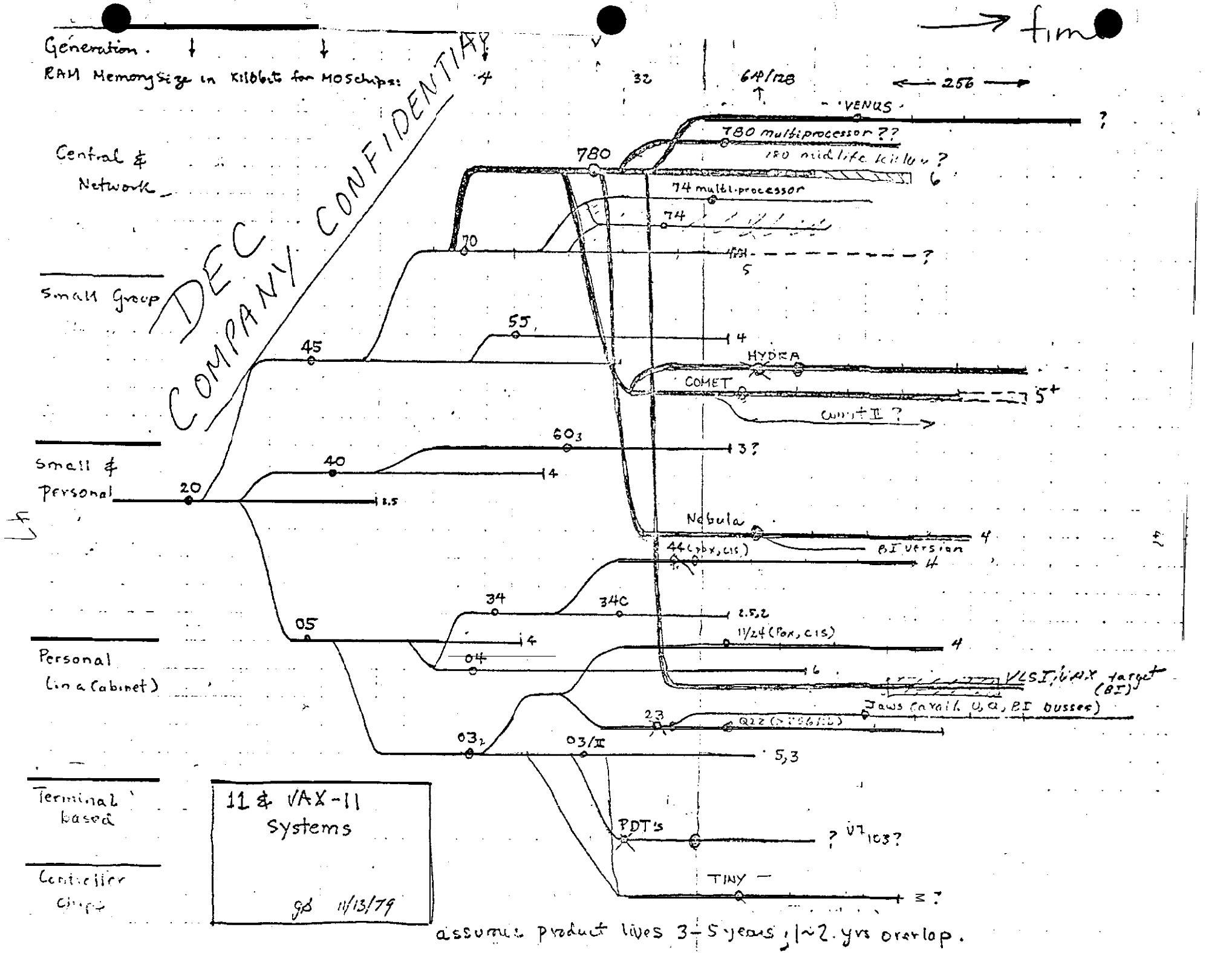
Develop a single VMS operating system to span the product range if technically and operationally feasible; "low end" products will mask the VMS capability for the unsophisticated users or, if efficiency demands, new code compatible at all interfaces with compilers and utilities will be developed. VMS will offer full mainframe capabilities allowing concurrent batch transaction, processing, and time-sharing, along with limited real-time.

- Provide superior data-base capabilities in the two - three year time frame.
- Focus on data access and data manipulation tools for the non-programmer, heavily based on graphics terminals.
- Provide word processing and electronic mail as applications on the general purpose VAX systems.
- Data integrity will be a feature available independent of high-availability (non-stop) operation through Hydra.
- High-availability (Hydra) will be a standard attribute of VAX systems at the customer option.
- Fire-wall funds to stimulate acquisition of cross-industry applications packages. Provide industry specific applications via internal development or acquisition. Leverage field resources by investing heavily in product quality assurance and self installing systems capacity including remote software update and diagnostic strategies.
- Move systems-level code for 11 based software (SCS-11, TRAX) to VAX compatibility mode if technically or strategically viable (under investigation now) otherwise provide user-level compatibility via

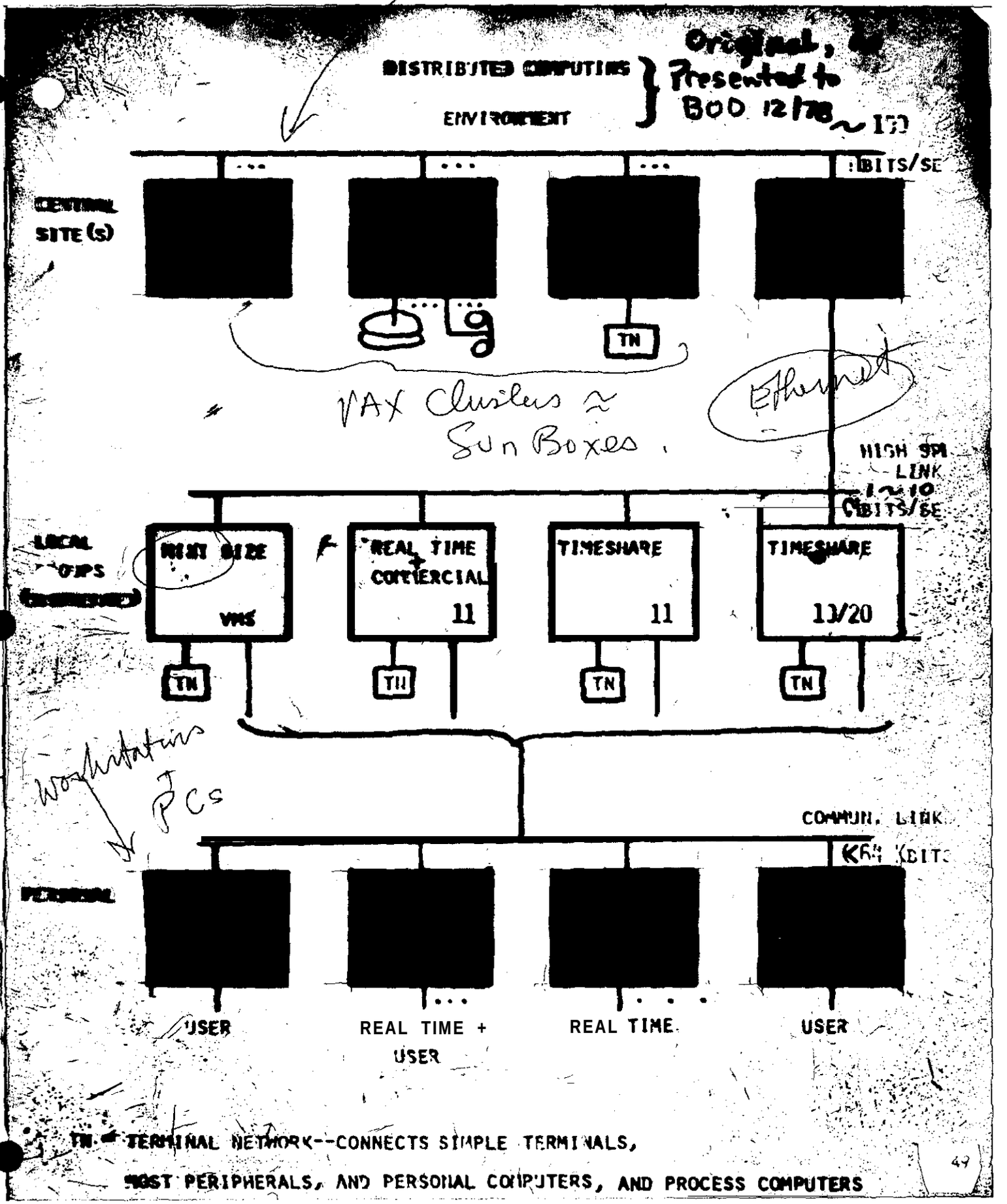
native mode VMS layered products.

. Shift DECNET strategy to strong IBM interconnect and VAX binary image compatibility for distributed processing; constrain PDP-11 DECNET FUNCTIONALITY EXTENSIONS, speedup DEC 20 network capabilities.

. Converge on ease of DEC '20 to VAX movement through common language definitions, (common implementations where feasible) common user-level utilities and data conversion routines. For each new DEC 20 or' VAX customer, as time progresses, make the movement between systems more attractive.


Generation
RAM Memory Size in Kilobits for MOS chips:
4
32
64/128
256
time
DEC COMPANY CONFIDENTIAL
Central & Network
Small Group
Small & Personal
Personal (in a cabinet)
Terminal based
Controller chips
11 & VAX-11 Systems
gb 11/13/79
VENUS
780 multiprocessor ??
780 mid-life ECO ?
780
74 multi-processor
74
70
45
55
20
40
60₃
34
34C
05
04
03₂
03/II
23
HYDRA
COMET
Comet II ?
Nebula
BI version
44 (pbx, cis)
11/24 (Pox, cis)
VLSI VAX target (BI)
Jaws (avail. U, Q, BI busses)
Q22 (?? 56kB)
PDT's
? U7 103?
TINY
assume: product lives 3-5 years; 1~2 yrs overlap.

2 x 80 Mbits/sec



TN - TERMINAL NETWORK--CONNECTS SIMPLE TERMINALS, HOST PERIPHERALS, AND PERSONAL COMPUTERS, AND PROCESS COMPUTERS

49

ID#332

Why Change the Current Strategy?

We have arrived at the current strategy by integrating our past customer needs, with the result that nearly every past system we have ever built is being evolved. This evolution creates too many systems with converging functionality. By prolonging the phaseover to VAX, we're unable to invest enough in VAX due to continuing and evolutionary support costs. Also, we're unable to provide applications, or have any slack resources to respond to competitive threats (eg. large micros or focused products such as the 8100).

We are just beginning to get a feel for the expense of putting new software systems in the field, and there are other systems still to come. Since we provide many choices, we find our sales and customers have difficulty deciding what to sell and buy. This makes us difficult to understand and to do business with. Lots of low volume products mean we don't have adequate volume to amortize the start-up manufacturing, sparing and training expenses.

Why Not Aggressively Evolve All Four Base Hardware Architectures?

In reality, our past strategy has been almost a divisional product structure. Customers can choose among the 4 basic hardware computer systems with 2+3+7+1 models and then select the appropriate software system, among 2+2+7+1 software systems for 8, 10/20, 11 and VAX respectively. This gives us several hundred systems. The number of alternatives is too large, resulting in small and decreasing volumes of each of the systems as all architectures are extended to cover a full range that we believe our customers require. We can not afford all the necessary enhancements to support four architectures over the range of size and use that our customers demand.

While any of the architectures can be implemented at any size down to and including LSI chips, there is no significant differential cost of the processor between the 10/20, VAX and an 11 with commercial and scientific instruction-sets.- An evolved 8 to handle the strategic range would even be the same cost. The main differentials are: the cost of the memory to hold the task:: and the size of the operating system software. The 10/20 operating systems have been oriented to generality, and while VMS and TOPS 20 have roughly the same functionality, the 10/20 requires 512K bytes of resident memory, whereas VMS require 256K bytes. This occurs because TOPS 20 has evolved and because of the efficiency of VAX architecture. VMS also has real time capability. Similarly, it is now inappropriate to consider 10/20 based architecture for terminals and personal computers, when compared with VAX, because small problems cannot be encoded to be competitive with modern 8- and 16-bit microprocessors. Furtheremore, extensions to the 10/20 architecture would require basic work in the operating system and languages to build a VAX competitive product.

Why Not Segment Products By Market?

Since the 10/20 has significant commercial software and since it is believed that our customers arc insensitive to architecture, we might simply have a market segmented approach and use 11's at the low eitd and 10/20's in the high end. Lower priced 10/20's would be implemented over time as appropriate.

Our technical users (EDU, ESG and even LDP) do not segment computer purchases into commercial vs scientific. A "control" customer such as DuPont doesn't segment its applications either. Even NASA wants COBOL to off-load their mainframe and to do administrative EDP. Universities likewise want a single machine, and hence the software will be "pulled" into existence. Version 4 of VAX COBOL executes faster than the 20's already.

Since there is basic incompatibility between the 11 and 10 architectures, the migration problem is enormous. Now our large commercial customer base is with 11's. Our users perceive VAX and 11 are of the same family.

The 10/20 still requires basic changes (CIS, 30-bit addressing) to bring it up to VAX performance and capability together with compilers and some basic software (eg. multi-keyed ISAM). TRAX-36 and RSTS 36 will also have to build off our 11 base. In short, while it might bc feasible to build 10/20 software so that our 11 users could meet our strategic goals for distributed processing, we would still fall short of the distributed system we can build with a single architecture as described in a subsequent rationale.

How Do Customers Perceive The Situation?

In mid October, a group at Bell Laboratories, building PBX systems visited us and made the comments:

> "Only you have the basic architecture in VAX to cover the range of products we need for distributed processing. This includes: terminals, offices and large offices.
>
> Give us a truly compatible range of VAX machines, starting with a VAX-on-a-chip and extending through the IBM 3033. (Don't corrupt VAX, since as in the 11, we must preserve our software base, given that the processor is only 4% of the cost.)
>
> The machines must have a reliability and security orientation.
>
> Why don't you do it?
>
> We will help fund the development."

Recent discussions with Stanford, ITT, CERN, NASA(Ames) indicate concurrence even though they are large 10/20 and 360/370 users. MIT is proposing to build a homogeneous VAX-based network. DuPont wants a similar structure, but is less rigid on the need for a homogenous architecture even though they've standardized on the RSTS machine internally for many of their systems. (There's a videotape describing their needs arid ideas.) CERN, and NASA (Ames), for example, feel that the large mainframe may be on the way out as we offer small group-level computing with VAX.. There are probably 10/20 customers who feel strongly that we should base our future on 10/20's. The main reason to focus on the single architecture is that it is part of the **11** family.

Why Have A Single Architecture? — Homogeneous Computing

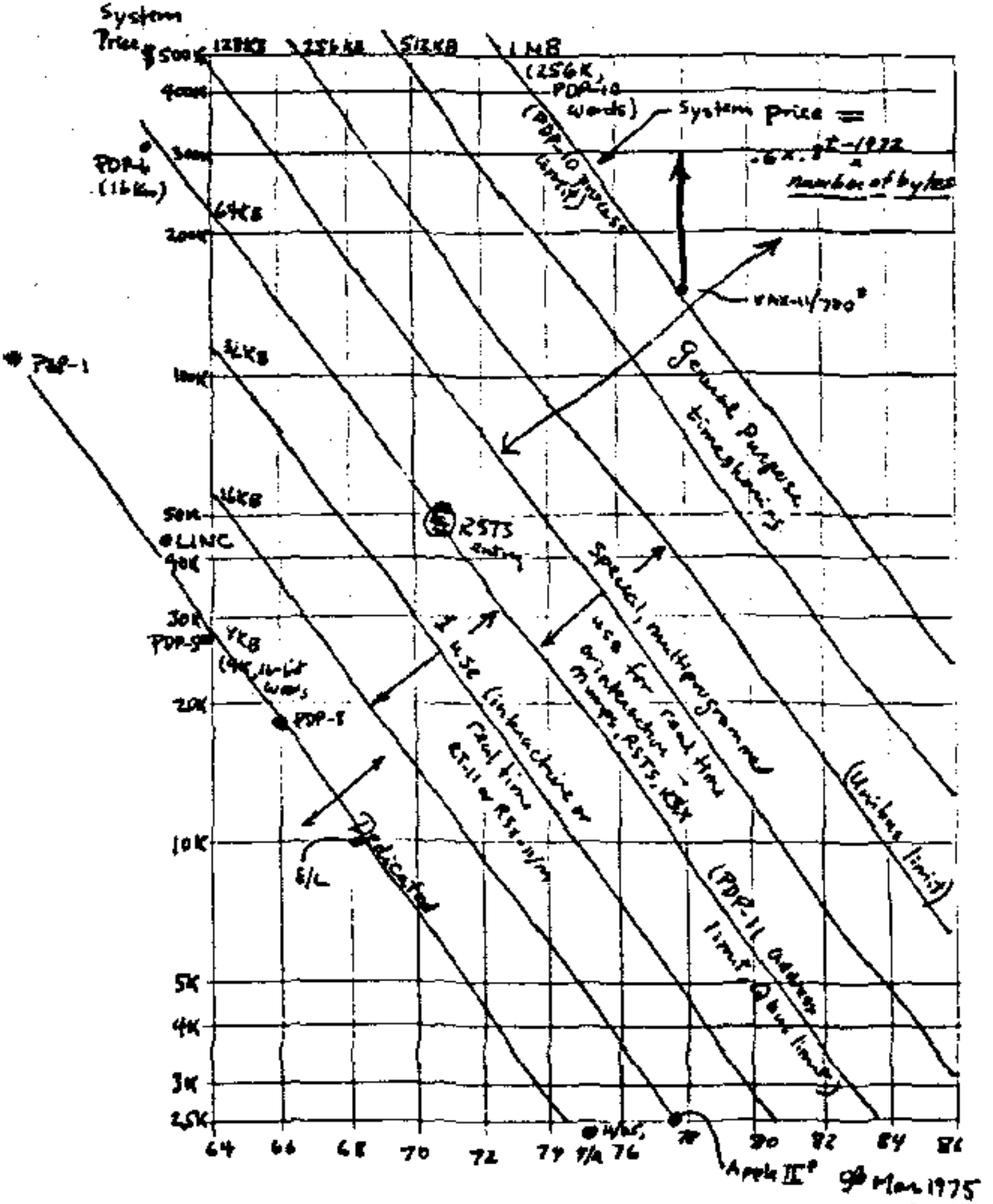
There are technical, marketing and economic reasons for choosing a single architecture at this time on which to base a major part of our future. However, this does not mean that we must neglect our 12- and 36-bit user base.

While computer networks can and have been built with heterogenous computers and IBM is betting that it can build distributed computing systems with only similar machines, a single architecture is the most effective for distributed computing systems. The homogeneous (identical) architecture approach insures that software will give the came results no matter where executed and therefore programs may be run anywhere in the network, data stored anywhere and programs moved about in their object form without the overhead of recompiling or translation as data is transferred. This also insures that the human interface to the system remains constant, because identical software is executed in different machines instead of relying on software that is specificed to have identical interfaces (e.g. languages, command languages, file systems, utilities).

From a user viewpoint, the homogeneity is ideal, and the success can be verified by reviewing the history of IBM's decision to build the 360 (and not continue with the 1401 1410, 7070 and 7090 series machines), even though there was an incredible base of these machines. This was also the time that Honeywell established itself with the 200-series and RCA with the 301. The homogeneity provides a simpicity for the entire DEC organization and its customers, and lets us all focus on end use applications rather than choosing a particular operating system and language. Currently, we have too many low level, incomplete choices and the software efforts of us and our users are not focused. An applications base can only be built effectively on a good, stable architecture.

1/43

Economically, a homogeneous architecture is essential because it allows us to concentrate and become a focused, high volume manufacturer and take advantage of learning curves. While 10% learning curves mean a doubling of manufactured quantity causes a 10% decrease in cost, they also imply that having two very similar products at one-half volume causes 10% higher costs in each. There are similar effects of learning in hardware, software and sales training costs, although the learning costs are small in comparison with the logistics and start-up costs associated with our many, different though functionally equivalent, products. We become difficult to do business with in the process.


System Price
$500K
400K
300K
200K
100K
50K
40K
30K
20K
10K
5K
4K
3K
2.5K
128KB
256KB
512KB
1MB
(256K, PDP-10 words)
System price = .6x.$\frac{\$}{1972}$ x number of bytes
PDP-10 limit
PDP-6 (16K)
64KB
32KB
16KB
PDP-1
PDP-5
8KB (9K, 18-bit words)
PDP-8
PDP-11 limit
VAX-11/780*
General purpose timesharing
Special, multiprogrammed use for real time monitors, RSTS, RSX
Real time, CAMAC, RSX-11M
Dedicated
E/L
(Unibus limit)
PDP-11 address limit
2STS analog
LINC
Apple II*
64
66
68
70
72
74
76
78
80
82
84
86
9th Mar 1975
* w/75 addition

Why Base The Architecture on VAX?

Although we went through the arguments in the spring of 1975 when we decided to build VAX instead of building lower cost versions of the 36-bit architecture, we now have a real machine that met it:; development goals and has user acceptance on which to base future products in a natural, evolutionary fashion.

Mostly, the choice of VAX in 1975 was based on having a large, PDP-11 user base. Furthermore, the choice to stay with the 8-bit byte was of convenience because of the IBM and communications worlds.

The VAX architecture was designed to permit the building of a range of machines with sizes that are important to us. Our targeted range of implementation was 1000:1 and this is attainable with an LSI implementation for terminal applications in January 1982. This is why a small page size and simple paging system was chosen, versus a larger page size and more complex scheme that would have been particularly oriented at large systems. However, it would not be wise to build the machine 1000 times as large in 1982, because it would take the system size beyond the suggested $250 K selling price limit and into mainframe price and customer expectations territory. Thus, in January 1982 the LSI VAX could sell for several *hundred* dollars at a board level. An ECL technology machine might be configured to sell for $ 400 K, giving the 1000:1 in price and a range of 64 Kbytes of RAM and ROM for VMS in the terminal to as much as 32 Mbytes in the large configuration (4000 64 K chips, costing $60K and occupying 20 PC Boards).

VAX was also designed to address the high cost of programming. Already VAX has been acclaimed (by a customer in our ads) as the best machine for implementing software. The large address space eliminates the need for much of the effort we spend encoding large programs into overlays. The architecture has instructions for the important data-types, the addressing is independent of the data-types and the important language constants are built into the hardware. There is clear separation among program and data. The procedure call instruction allows more sub-program sharing than with architectures that are dependent on conventions (e.g. 360 and 10/20) and it eliminates a class of systems programming errors resulting from the multiple assignment of general registers among different programs.

The 32-bit address space of VAX appears adequate for the computing needs in the foreseeable future and there is extension capability given that any special needs arise. The address space and protection modes also give us a capability to run sub-programs written in different languages as a single program. This capability is unique and may turn out to be the single most important attribute of the machine. Since only one other computer has the capability, we don't understand it or how valuable it will be.

Another technical reason is based on the encoding efficiency of the VAX instruction-set. The VAX architecture can encode a Fortran program in about 1/2 to 2/3 the space of a comparable large computer such as a 360 or our 36-bit computer, while providing 32-bit addresses versus 24 or 20 bits of addressing for the 360 or 10/20. Benchmarks in BLISS and FORTRAN show this now, and the Common Family Architecture studies also indicate similar results. While memory cost is decreasing, memory is still a significant fraction of system cost. Three years of cost decline at the historical rate of 20%, is required to get factor of 2 the cost difference back. That is, from a memory cost viewpoint, we have a 3-year cost edge on the market. More importantly, there is a similar effect on performance. By having only 1/2 the bits to move between primary and secondary memories, the performance is higher due to disk-MOS memory swapping bottlenecks.

Finally, we have an 11 user base on which to build that is approximately 7 and 50 times as large as our 36-bit base in terms of installed equipment dollars and installed units.

Why Not Use The 10/20 As The Base?

The software and user base on the 10/20 is the major reason to not arbitrarily reject the architecture. On the other hand, since the 11 user base is larger and has grown more rapidly, its software base is larger and we have to protect and build on it as a higher priority.

Right now, the 10/20 requires incremental investment to make it competitive with VAX and the rest of the mainframe market. Extension to provide a large address space, to extend the floating point range to fulfil customer commitments, and to give a competitive commercial instruction set for COBOL are needed. Making these hardware investments requires comparable software investments and we must again wait to compete because there is a new machine and software to support.

Subsequent implementations for small systems will be expensive both in terms of new software and start-up because TOPS 20 has been oriented toward large mainframe generality. Smaller systems will require contractions. Also it stands to only cloud the market more as alternatives for mid-range systems will include 2 VAX and 1 or 2 11-based systems. As small systems are implemented there is a need for compatibility with the even larger 11 base.

Why Distributed Computing?

Distributed computing keys off our strength in interactive computing through timesharing, small systems, real time computation, terminals, and networks. Furthermore, we believe this is what our customers want. The issue is not distributed computing, but solving the problems that it creates.

This was the book I maintained during VAXA, design process — until it was turned over to Bill Demmer as a project!

COMPANY CONFIDENTIAL
BLUE Book
11 VAX WORKBOOK

I. INTRODUCTION 11(VAX) PROPOSAL)
RATIONALE

II. TECHNOLOGY AND SYSTEM ENVIRONMENT 1975-1985

III. PMS STRUCTURES
PMS CONTROLLER STRUCTURE POLICY

PMS STRUCTURE OF 11VAX MODELS

GB

IV. PRINCIPLES OF OPERATION FOR SELECTED MACHINE--THE ISP
A BYTE ORIENTED ISP PROPOSAL FOR THE EXTENDED PDP-11 ARCHITECTURE

SUBROUTINE CALL & RETURN MECHANISM

THE SKELETON CALL INSTRUCTION

PENDING ISSUES FOR BYTE ISP

AN ADDRESS MAPPING PROPOSAL FOR THE EXTENDED -11 ARCHITECTURE

PROCESS STRUCTURE

A HIGHER-LEVEL INTERRUPT PROPOSAL

Not included

V. HARDWARE IMPLEMENTATIONS

VI. SOFTWARE IMPLEMENTATION

VII. APPENDICES
C.11 CONSTRAINTS, GOALS, IMPLICATIONS AND SOME ALTERNATIVES

RATIONALE OF MAJOR DESIGN CHOICES

mjk [6/4/75, CONTEN.VAX]

SUBJ: INTRODUCTION 11(VAX) PROPOSAL

The 11 VAX is a proposed, upward compatible follow-on to the PDP-11, which we would deliver as early as 18 months from now (depending on the size of the system first implemented). This proposal, though incomplete, is a first level cut at the total design and will be reiterated. It does, however, meet the goals set down in the appendix. We do not expect radical changes.

The 32-bit computers which are beginning to appear have acted as a catalyst to generate this proposal; but in reality, the very small address space for a user program is the real protagonist of the proposal. Nearly all computers can be implemented with wide memory and data and register paths, but since currently 11 doesn't have large addresses, it lacks a 32-bit integer data type to deal with addresses.

Had we the foresight, it was clear the pure, 16-bit 11 was born to have a short, happy, prolific, profitable life. In 1969, an address of 16-18 bits, and a system size being sold of 13-15 bits, left only 3 bits of address growth left. At the constant-price historical memory growth rates of 26 to 41 percent per year, only 6 to 9 years of comfortable lifetime is allowed, bringing it to 1975-1978.

This address problem is so severe, that based on the current memory size increases, the 18-bit UNIBUS will be limited to provide total systems which will sell for less than $25K. About 1.5 years ago, we embarked upon a strategy of bringing the 10 (i.e. 11/85) down in price to cover what had been traditional high end 11 business. Now, we--a group called VAXA--has been chartered to propose a follow-on 11, which will also run current PDP-11 programs.

VAXA, a small group of developers, (G. Bell, D. Cutler, T. Hastings, R. Lary, S. Rothman, W. Strecker) have been meeting since April 1, to determine various alternatives to extend the 11. This document describes the proposed design.

A second group, VAXB, has met with us 10% of the time to review the proposal. The format we have used has been to:

1. Outline problem areas: virtual address and memory management, instruction-set, process structure, I/O, and compatibility with existing software.

2. Posit goals and constraints we believe the area imposes.

3. Carry out a design activity in the area by generating proposals, and then testing the proposals against the constraints and goals to select a design.

4. Present the resulting design to the VAXB group.

Therefore, we have continued to refine the goals and constraints as we know more about the design alternatives. The current G's

and C's are presented in the Appendix. It will continue to be updated as we move around in the design space.

The exploration of the design space has been relatively straightforward, because:

1. Strecker, Mudge, and Arulpragasam had proposed various alternative memory management schemes to extend the address space. In making the proposals, we learned that modifying the address radically affected the instruction-set; and hence, the software compatibility. Since addresses are stored on the stack, nearly all references to data are changed. Also, nearly all data operators have to be changed to get at longer address words.

2. Strecker had been working on this problem for several years and had 2 proposals for instruction-sets that were relatively close and relatively far from the current 11 design. Hence, there was a notion of the cost and payoff for different instruction-sets.

3. There has been much work on machines since the PDP-11, and we have borrowed ideas from many different computers and papers.

The workbook is divided into these parts:

1. Rationale--this section goes into the reasons we believe the VAX is a significant computer, and why it should be considered. It is compared with the 11 and with the 10, and includes what we have learned from the 11.

2. The environment for machines in 1975-1985.

   We believe that 11VAX must cover a larger dynamic range, and have a longer lifetime than the 11. Hence, it is important to have a vague understanding of what we feel the marketplace will look like in 1980.

   Implications of the technology projections are also included.

3. Principles of operation.--this section describes the proposed machine in somewhat brief form.

4. Implementation plan for hardware--which models, when.

5. Implementation plan for software--which software, coding conventions, implementation languages.

6. Appendices

   A. Goals and constraints--the formal document which has guided the design, and used as a yardstick for it.

   B. Rationale for selecting the instruction-set base. Here, we take the appropriate goals and constraints, and measure various ISP's against the goals and constraints.

   C. Rationale for selecting the virtual addressing/memory protection mechanism.

Again, we show some of the alternative designs, and the criteria we think are important for judging the mechanism. We are recommending a mechanism that is not conceptually trivial, hence it will take some time to understand it. Since we are not working on this problem of adding understanding now, please bear with what will require several tutorials to get through. We believe the mecanism has better protection and programming than any system we know of.

GB:mjk [5/28/75, PI.V01]

SUBJ: VAX RATIONALE

Why should we consider another machine (11VAX) beyond an 8, 10, or 11? or...what 11 VAX provides.

---

0. The 11 runs out of performance in the larger memory configurations in our $50K-100K market as memory prices decline. The UNIBUS system will limit the system to $25K in 1980!! Our mini users require a migration to a "compatible" machine...we have to do something. (For example, a more radical alternative would be to push the PDP-10 down further to cover the 11.)

1. It is 8-bit byte organized. While this isn't technically very significant, the market feels it is. It does, however, offer a significant technical benefit by being a "standard", hence there's no hardware and/or software to pack/unpack data.

2. It will capitalize on user familiarity with 11's; all 11 instructions map into the 11VAX instructions. Programs will be written in the same style as the 11.

3. It will be sold as an 11, and there will be compatibility such that 11 user mode programs run on it.

4. It is also DEC's "new" architecture.

5. (Hopefully) we have learned a great deal about the use of the 11-style architecture, and hence can provide a system with significantly better hardware, operating system, languages, and applications.

6. The range of price (memory size), performance, etc. is significantly greater than with any of the alternative strategies (designs). We are providing the hardware mechanism to have either dynamic (demand) segmentation and/or paging. This means that all program segments need not be resident; hence, a large, segmented program can be run on comparatively smaller machines. In this way, a user can run a program across a range of machines in a "downward" fashion...i.e. programs will run independent of the primary memory size. Thus a user can at least run comparatively large programs on small configurations and his programming costs will be less due to manual program segmentation.

7. It provides much growth to get us to 1985 by having:

   A. Many extra data-types and their operations.

   B. A very large virtual address.

   C. Multi-processors--allowing users to field upgrade a particular system configuration along all performance dimensions: memory size, processor power, secondary

memory (disks), and terminals.

8. It will merge systems development to a single system to cover a large range.

9. A general programming environment that we believe is better than any machine currently on market through:

   A. Better language support (see 10).

   B. General mechanisms that favor high level language programming by increased instruction symmetry (hence, people are less likely to do significantly better at low level coding).

   C. More code sharing by global protection and large shared address space among all parts of the system including: operating systems, utilities, compilers, run-time systems, and user programs. Since all subprograms share the same large memory space, they can call one another, independent of their location in a part of the system.

   D. Since all programs share the same address space, operating system programs will execute faster by not having to relocate addresses on behalf of its users.

   E. More protection among the parts through a ring structure will enable better automatic protection of the parts. This in turn will permit faster execution by not requiring so much checking on the part of lower level programs.

   F. More reliability throughthe procedure call mechanism by having automatic saving and restoring of registers at the callee in terms of registers the caller wants preserved ,AND, the callee uses.

   G. Sharing of data and/or user program segments by independent processes. This permits a wide variety of system uses:

      i. Conventional shared operating systems, utility and languages.

      ii. Separate (shared) processes for each terminal in a TPM system.

      iii. A small number of processes for each activity in a TPM system which exchange terminal data in a queued, pipelined fashion. Messages are transmitted among processes either with shared buffer segments and/or dynamic message segments attached to the process.

      iv. True, parallel processing can be accomplished with shared program and data segments. (Fast, synchronization primitives at the user level are required.)

10. We believe it has better support for languages than current

machines by:

A. More data types and consistent operators for them: bytes, 16-, 32-, and 64-bit integers, floating, double floating, and byte (character) strings.

B. Generalization of the existing registers (i.e. floating point) to provide 16, more generally available registers. This permits better code generation by compilers.

C. Some dedicatedness of 2 additional registers (the argument and local pointers) to improve I-stream size, and enforce coding conventions along modules.

D. More support of R6 for better operation of the stack-- many languages require this...particularly the more modern (e.g. APL, PL/1, PASCAL).

E. Really good procedure calling instructions.

11. It provides better (faster) and cleaner control of context by:

A. Mapping state into a single register array which is only 16 x 32 versus 1 X 16 + 8 X 16 + 6 X 64.

B. correct operation on segment faults.

C. Instructions for rapid context switching together with implementations which provide additional sets of registers.

D. A single address space (see 9C, D, E).

12. It has better control of I/O for the Operating system and the user. A user program can directly control an I/O device with complete protection. Interrupts return (almost) directly to the user program.

The 11VAX is an upward compatible 11, but it is another machine to compete for resources.

---

0. Already, it is clear that it is difficult to get resources for 11VAX. The study group had nearly all the resources it could manage--still, we lacked resources to develop a general operating system philosophy (or design), coding conventions, and general software plan as part of the architectural design.

1. It is a fourth machine requiring software (beyond 8,10,11).

2. It has to eventually replace 11's in production, otherwise it makes no sense as a fourth, supported machine.

3. Much current 11 software would be either brought over directly (e.g. compilers, utilities) or convert (e.g. monitors) to operate on it.

4. Much new software will be written for both 11 and 11VAX for sometime...until we can bring out the low end part

and replace the 11.

5. It looks feasible to write software which will run both on 11 and 11VAX, but given our previous record of compatibility, one should be extremely skeptical! The only way this can be accomplished, I believe (GB), is through the use of an implementation language which has code generators for 2 machines, and even then common programs will sacrifice 11VAX or 11 space and time. We have not used implementation languages such that our software engineers are convinced they work!

What's different between 11VAX and an 11 (or closer relative)?

1. It increases the virtual address space for a single task to 29 bits--well beyond that provided by all competitors. This (in turn causes a significant perturbation in the software, since many programs use the stack for addresses, hence all accessing is modified.

2. Its goal is to provide a significantly greater performance, price, and price/performance range than 11's. It (hopefully) does so by:

   A. Providing a bigger VA, thus permitting a larger address range (hence implementation range).

   B. Using a highly variable instruction length.

   C. Using several bus schemes for interconnection of components (11's could too).

   D. Multiprocessors (11's could too)...but no use w/o bigger VA.

   E. Using a segmentation scheme such that demand segmentation and/or paging can be used to provide large VA function on small physical memories (11's could too).

3. A closer relative has problems:

   A. There are only a few opcodes left in the 11, hence no practical way to extend it. Any form of escape would increase the instruction stream to be non-competitive.

   B. The alternative VA mechanisms are too segmented and would require much software for efficient utilization.

   C. A more radical change, such as 11VAX is probably inevitable.

4. A strict 11 might be more efficient, given the small addresses, but we would have to convince our users that a large VA is not what they need...something we've been unable to do so far.

5. A "closer 11" would be easier to maintain programming compatibility, but since the larger address perturbs all programs, it isn't clear that we gain by not taking a bigger step in the instruction-set. What we are most likely

to lose with 11VAX is the ability to write new programs that run on 11's. We can only do this effectively by using a higher level language with separate compilers!

What's different between 11VAX and a 10-based system?

1. A 10 has some advantages:

   A. It could be extended downward and provide many of the advantages that 11VAX would.

   B. It has much software--much of which is written for larger memory configurations.

   C. We understand it, and have solved many problems which have to be carried over into the 11VAX.

   D. The memory required to express an algorithm is relatively small. The 10 is particularly efficient at encoding FORTRAN.

   E. It's here.

   F. There's a cult.

2. The 10 does have disadvantages:

   A. The software and its documentation is tuned to the knowledgeable user...VIRUS has a goal to include more users.

   B. It is not based on an 8-bit byte, even though it processes it.

   C. It is unfamiliar to the people who would market, sell, and service it...and buy too.

3. The 11VAX is later, hence can learn from the past.

   A. It is "familiar" to 11 users since all 11 instructions map into 11VAX.

   B. Generally, there are minimal compatibility constraints, as compared with the various 10 models which are very compatible.

   C. There are better sharing facilities with the memory management and subroutine calling mechanisms than the 11's and this is comparable to or exceeds the 10 capability.

   D. Having a "standard byte" could simply standards including text, communications, etc.

4. The 11VAX will have to span a significantly greater range than the 10. The minimal 10 is 32K X 4 or 128K bytes versus the 8K byte 11. While we could rewrite the 10 software for a low end 10, there is some concern as to whether we could get down to the small size currently provided on an 11. Glom Hastings, at least, believes this need not

be true, but is only the result of the operating system implemented on the 10.

5. There is some relatively high risk associated with achieving the 11VAX range goals with compatible user level software. In this, we end up with another large machine, requiring a very large software investment which replicates some of the 10. The 11 would still remain in production with software being written for it.

What have we learned from the 11 that is useful to 11VAX?

To a certain extent, nearly all we know appears rather trivial post facto, but it should be noted, as it impacts 11VAX.

1. The address space of 16 or 18 bits was clearly tiny, given that the system range in 1969 was 4Kw-16Kw or 13-15 bits. Only 3 bits expansion were provided and with a conservative 26% to a more rapid 41% growth/year, then only 6 to 9 years growth or until 1975 to 1978--assuming an 18-bit max. Each extra bit provides 2 to 3 years of growth (depending on whether we assume 40 or 26% growth). Hence, assuming we are out of memory space now with 16 bits, the extra 13 bits provide 26 to 40 years of growth. (By contrast, the 10 had 20 bits per user--enough to carry it until at least 8 years beyond an 11. The KL10 has been expanded to 36 bits!)

2. The range of models and options as we have implemented, create numerous software problems:

   A. All CPU arithmetic options require totally different software--for EAE, EIS, FIS, and FPU. The best solution would have first defined the registers and opcodes. Machines not having a hardware feature, would trap, and execute the instruction in a common way. This would have avoided the plethora of different operating systems, run time and compiler options.

   B. By putting in high complexity options in only 1 model (e.g. I+D-space), it gets no support.

   C. In not having a complete protection system, e.g. KT11, adds overhead in general operating systems in order to do the mapping (e.g. RSX-11/D).

   D. It is too early to tell about the multiple ASCII consoles.

3. Devices of the same class differ widely among direct options--(e.g. the COMM options). As a result, device support is varied. This problem could be solved in numerous ways:

   A. Planning--"someone(s)" watches it all.

   B. Funding--the device funds operating system support.

   C. A software device driver standard--low level software would bring device to a standard internal operating system level.

D. Devices would be more standard across option types (e.g. 36v).

4. UNIBUS address assignment is a mess. Since every computer we build is essentially unique with differing options, and since all options take up more than the 4K of address space, there is a conflict in the naming of I/O devices. There are several solutions:

   A. A rom mapping.
   B. Automatic relocation of devices.
   C. Take a larger address space for I/O.

5. Multi-computer systesm. Two UNIBUSSES have an inherent deadlock problem. That is A and B make simultaneous requests, A can't get B's UNIBUS, nor can B get A's to complete the transfer.

   In the case of the 11/85, requests are separated from access, and requests can be passed among busses without deadlock.

6. Multiprocessors on a single bus or multi-port memories. CSS has built some. Here we intend to provide for them.

7. Timers would be useful.

8. Software coding conventions would increase the ability to share software.

9. There are many operating statistics regarding frequency of use. These statistics have been employed in the design of the new instruction-set. Some of these include:

   A. More generality of operators and data-types. This will simplify code generation and also aid the user in remembering the machine.

   B. Addressing of floating point array elements by context. This eliminates the ASL's to get the index right so it can be added. This gets 10%-20% on some benchmarks.

   C. Dedication of 2 more registers with support by modes to access the stack and arguments with shorter addresses.

GB:mjk [6/4/75 Version 2, P11.VM1]

C1488

D I G I T A L INTEROFFICE MEMORANDUM

PAGE 1

SUBJ: INTRODUCTION 11 (VAX) PROPOSAL

DATE: 05-28-75
FROM: GORDON BELL
EX: 2236
MS: ML12-1/A51

TO: FILE

SUBJ: INTRODUCTION 11(VAX) PROPOSAL

The 11 VAX is a proposed, upward compatible follow-on to the PDP-11, which we would deliver as early as 18 months from now (depending on the size of the system first implemented).

The 32-bit computers which are beginning to appear have acted as a catalyst to generate this proposal; but in reality, the very small address space for a user program is the real protagonist of the proposal. Nearly all computers can be implemented with wide memory and data and register paths, but since currently 11 doesn't have large addresses, it lacks a 32-bit integer data type to deal with addresses.

Had we the foresight, it was clear the pure, 16-bit 11 was born to have a short, happy, prolific, profitable life. In 1969, an address of 16-18 bits, and a system size being sold of 13-15 bits, left only 3 bits of address growth left. At the constant-price historical memory growth rates of 26 to 41 percent per year, only 6 to 9 years of comfortable lifetime is allowed, bringing it to 1975-1978.

This address problem is so severe, that based on the current memory size increases, the 18-bit UNIBUS will be limited to provide total systems which will sell for less than $25K. About 1.5 years ago, we embarked upon a strategy of bringing the 10 (i.e. 11/85) down in price to cover what had been traditional high end 11 business. Now, we--a group called VAXA--has been chartered to propose a follow-on 11, which will also run current PDP-11 programs.

VAXA, a small group of developers, (G. Bell, D. Cutler, T. Hastings, R. Lary, S. Rothman, W. Strecker) have been meeting since April 1, to determine various alternatives to extend the 11. This document describes the proposed design.

A second group, VAXB, has met with us 10% of the time to review the proposal. The format we have used has been to:

1. Outline problem areas: virtual address and memory management, instruction-set, process structure, I/O, and compatibility

with existing software.

2. Posit goals and constraints we believe the area imposes.

3. Carry out a design activity in the area by generating proposals, and then testing the proposals against the constraints and goals to select a design.

4. Present the resulting design to the VAXB group.

Therefore, we have continued to refine the goals and constraints as we know more about the design alternatives. The current G's and C's are presented in the Appendix. It will continue to be updated as we move around in the design space.

The exploration of the design space has been relatively straightforward, because:

1. Strecker, Rudge, and Arulpragasam had proposed various alternative memory management schemes to extend the address space. In making the proposals, we learned that modifying the address radically affected the instruction-set; and hence, the software compatibility. Since addresses are stored on the stack, nearly all references to data are changed. Also, nearly all data operators have to be changed to get at longer address words.

2. Strecker had been working on this problem for several years and had 2 proposals for instruction-sets that were relatively close and relatively far from the current 11 design. Hence, there was a notion of the cost and payoff for different instruction-sets.

3. There has been much work on machines since the PDP-11, and we have borrowed ideas from many different computers and papers.

The workbook is divided into these parts:

1. Rationale--this section goes into the reasons we believe the VAX is a significant computer, and why it should be considered. It is compared with the 11 and with the 10, and includes what we have learned from the 11.

2. The environment for machines in 1975-1985.

   We believe that 11VAX must cover a larger dynamic range, and have a longer lifetime than the 11. Hence, it is important to have a

vague understanding of what we feel the marketplace will look like in 1986.

Implications of the technology projections are also included.

3. Principles of operation,--this section describes the proposed machine in somewhat brief form.

4. Implementation plan for hardware--which models, when.

5. Implementation plan for software--which software, coding conventions, implementation languages.

6. Appendices

   A. Goals and constraints--the formal document which has guided the design, and used as a yardstick for it.

   B. Rationale for selecting the instruction-set base. Here, we take the appropriate goals and constraints, and measure various ISP's against the goals and constraints.

   C. Rationale for selecting the virtual addressing/memory protection mechanism.

      Again, we show some of the alternative designs, and the criteria we think are important for judging the mechanism. We are recommending a mechanism that is not conceptually trivial, hence it will take some time to understand it. Since we are not working on this problem of adding understanding now, please bear with what will require several tutorials to get through. We believe the mechanism has better protection and programming than any system we know of.

GB:mjk

D I G I T A L INTEROFFICE MEMORANDUM

SUBJ: VAX RATIONALE

DATE: 05-28-78
FROM: GORDON BELL
EX: 2236
MS: ML12-1/A51

TO: FILE

SUBJ: RATIONALE

Why should we consider another machine (11VAX) beyond an 8, 10, or 11? or...what 11 VAX provides.

---

0. The 11 runs out of performance in the larger memory configurations in our $50K-100K market as memory prices decline. The UNIBUS system will limit the system to $25K in 1980!! Our mini users require a migration to a "compatible" machine...we have to do something. (For example, a more radical alternative would be to push the PDP-10 down further to cover the 11.)

1. It is 8-bit byte organized.

2. It will capitalize on user familiarity with 11's; all 11 instructions map into the 11VAX instructions. Programs will be written in the same style as the 11.

3. It will be sold as an 11, and there will be compatibility such that 11 user mode programs run on it.

4. It is also DEC's "new" architecture.

5. (Hopefully) we have learned a great deal about the use of the 11-style architecture, and hence can provide a system with significantly better hardware, operating system, languages, and applications.

6. The range of price (memory size), performance, etc. is significantly greater than with any of the alternative strategies (designs). We are providing the hardware mechanism to have either dynamic (demand) segmentation and/or paging. This means that all program segments need not be resident; hence, a large, segmented program can be run on comparatively smaller machines. In this way, a user can run a program across a range of machines in a "downward" fashion...i.e. programs will run independent of the primary memory size. Thus a user can at least run comparatively large programs on small configurations and his programming

costs will be less due to manual program segmentation.

7. It provides much growth to get us to 1985 by having:

   A. Many extra data-types and their operations.

   B. A very large virtual address.

   C. Multi-processors--allowing users to field upgrade a particular system configuration along all performance dimensions: memory size, processor power, secondary memory (disks), and terminals.

8. It will merge systems development to a single system to cover a large range.

9. A general programming environment that we believe is better than any machine currently on market through:

   A. Better language support (see 10).

   B. General mechanisms that favor high level language programming by increased instruction symmetry (hence, people are less likely to do significantly better at low level coding).

   C. More code sharing by global protection and large shared address space among all parts of the system including: operating systems, utilities, compilers, run-time systems, and user programs. Since all subprograms share the same large memory space, they can call one another, independent of their location in a part of the system.

   D. Since all programs share the same address space, operating system programs will execute faster by not having to relocate addresses on behalf of its users.

   E. More protection among the parts through a ring structure will enable better automatic protection of the parts. This in turn will permit faster execution by not requiring so much checking on the part of lower level programs.

   F. More reliability throughthe procedure call mechanism by having automatic saving and restoring of registers at the callee in terms of registers the caller wants preserved ,AND, the callee uses.

G. Sharing of data and/or user program segments by independent processes. This permits a wide variety of system uses:

i. Conventional shared operating systems, utility and languages.

ii. Separate (shared) processes for each terminal in a TPM system.

iii. A small number of processes for each activity in a TPM system which exchange terminal data in a queued, pipelined fashion. Messages are transmitted among processes either with shared buffer segments and/or dynamic message segments attached to the process.

iv. True, parallel processing can be accomplished with shared program and data segments. (Fast, synchronization primitives at the user level are required.)

10. We believe it has better support for languages than current machines by:

A. More data types and consistent operators for them: bytes, 16-, 32-, and 64-bit integers, floating, double floating, and byte (character) strings.

B. Generalization of the existing registers (i.e. floating point) to provide 16, more generally available registers. This permits better code generation by compilers.

C. Some dedicatedness of 2 additional registers (the argument and local pointers) to improve I-stream size, and enforce coding conventions along modules.

D. More support of R6 for better operation of the stack--many languages require this...particularly the more modern (e.g. APL, PL/1, PASCAL).

E. Really good procedure calling instructions.

11. It provides better (faster) and cleaner control of context by:

A. Mapping state into a single register array which is only

16 X 32 versus 1 X 16 + 8 X 16 + 6 X 64.

B. Correct operation on segment faults.

C. Instructions for rapid context switching together with implementations which provide additional sets of registers.

D. A single address space (see 9C, D, E).

12. It has better control of I/O for the Operating system and the user. A user program can directly control an I/O device with complete protection. Interrupts return (almost) directly to the user program.

SUBJ: VAX RATIONALE

DATE: 05-28-79
FROM: GORDON BELL

The 11VAX is an upward compatible 11, but it is another machine to compete for resources.

0. Already, it is clear that it is difficult to get resources for 11VAX. The study group had nearly all the resources it could manage--still, we lacked resources to develop a general operating system philosophy (or design), coding conventions, and general software plan as part of the architectural design.

1. It is a fourth machine requiring software (beyond 8,10,11).

2. It has to eventually replace 11's in production, otherwise it makes no sense as a fourth, supported machine.

3. Much current 11 software would be either brought over directly (e.g. compilers, utilities) or convert (e.g. monitors) to operate on it.

4. Much new software will be written for both 11 and 11VAX for sometime...until we can bring out the low end part and replace the 11.

5. It looks feasible to write software which will run both on 11 and 11VAX, but given our previous record of compatibility, one should be extremely skeptical! The only way this can be accomplished, I believe (GB), is through the use of an implementation language which has code generators for 2 machines, and even then common programs will sacrifice 11VAX or 11 space and time. We have not used implementation languages such that our software engineers are convinced they work!

What's different between 11VAX and an 11 (or closer relative)?

1. It increases the virtual address space for a single task to 29 bits--well beyond that provided by all competitors. This in turn causes a significant perturbation in the software, since many programs use the stack for addresses, hence all accessing is modified.

2. Its goal is to provide a significantly greater performance, price, and price/performance range than 11's. It (hopefully) does so by:

   A. Providing a bigger VA, thus permitting a larger address range (hence implementation range).

SUBJ: VAX RATIONALE DATE: 05-28-79
FROM: GORDON BELL

B. Using a highly variable instruction length.

C. Using several bus schemes for interconnection of components (11's could too).

D. Multiprocessors (11's could too)...but no use w/o bigger VA.

E. Using a segmentation scheme such that demand segmentation and/or paging can be used to provide large VA function on small physical memories (11's could too).

3. A closer relative has problems:

A. There are only a few opcodes left in the 11, hence no practical way to extend it. Any form of escape would increase the instruction stream to be non-competitive.

B. The alternative VA machanisms are too segmented and would require much software for efficient utilization.

C. A more radical change, such as 11VAX is probably inevitable.

4. A strict 11 might be more efficient, given the small addresses, but we would have to convince our users that a large VA is not what they need...something we've been unable to do so far.

5. A "closer 11" would be easier to maintain programming compatibility, but since the larger address perturbs all programs, it isn't clear that we gain by not taking a bigger step in the instruction-set. What we are most likely to lose with 11VAX is the ability to write new programs that run on 11's. We can only do this effectively by using a higher level language with separate compilers!

What's different between 11VAX and a 10-based system?

1. A 10 has some advantages:

A. It could be extended downward and provide many of the advantages that 11VAX would.

B. It has much software--much of which is written for larger memory configurations.

C. We understand it, and have solved many problems which have to be carried over into the 11VAX.

D. The memory required to express an algorithm is relatively small. The 10 is particularly efficient at encoding FORTRAN.

E. It's here.

F. There's a cult.

2. The 10 does have disadvantages:

A. The software and its documentation is tuned to the knowledgeable user.

B. It is not based on an 8-bit byte, even though it processes it.

C. It is unfamiliar to the people who would market, sell, and service it...and buy too.

3. The 11VAX is later, hence can learn from the past.

A. It is "familiar" to 11 users since all 11 instructions map into 11VAX.

B. Generally, there are minimal compatibility constraints, as compared with the various 10 models.

C. There is better sharing facilities with the memory management and subroutine calling mechanisms.

D. Having a "standard byte" could simply standards including text, communications, etc.

4. The 11VAX will have to span a significantly greater range than the 10. The minimal 10 is 32K X 4 or 128K bytes versus the 8K byte 11. While we could rewrite the 10 software for a low end 10, there is some concern as to whether we could get down to the small size currently provided on an 11.

5. There is some relatively high risk associated with achieving the 11VAX range goals with compatible user level software. In this, we end up with another large machine, requiring a very large software investment which replicates some of the 10. The 11 would still remain in production

with software being written for it.

What have we learned from the 11 that is useful to 11VAX?

To a certain extent, nearly all we know appears rather trivial post facto, but it should be noted, as it impacts 11VAX.

1. The address space of 16 or 18 bits was clearly tiny, given that the system range in 1969 was 4Kw-16Kw or 13-15 bits. Only 3 bits expansion were provided and with a conservative 26% to a more rapid 41% growth/year, then only 6 to 9 years growth or until 1975 to 1978--assuming an 18-bit max. Each extra bit provides 2 to 3 years of growth (depending on whether we assume 40 or 26% growth). Hence, assuming we are out of memory space now with 16 bits, the extra 13 bits provide 26 to 40 years of growth. (By contrast, a 10 has 20 bits per user--enough to carry it until at least 8 years beyond an 11.)

2. The range of models and options as we have implemented, create numerous software problems:

   A. All CPU arithmetic options require totally different software--for EAE, EIS, FIS, and FPU. The best solution would have first defined the registers and opcodes. Machines not having a hardware feature, would trap, and execute the instruction in a common way. This would have avoided the plethora of different operating systems, run time and compiler options.

   B. By putting in high complexity options in only 1 model (e.g. I+D-space), it gets no support.

   C. In not having a complete protection system, e.g. KT11, adds overhead in general operating systems in order to do the mapping (e.g. RSX-11/D).

   D. It is too early to tell about the multiple ASCII consoles.

3. Devices of the same class differ widely among direct options--(e.g. the COMM options). As a result, device support is varied. This problem could be solved in numerous ways:

   A. Planning--"someone(s)" watches it all.

   B. Funding--the device funds operating system support.

   C. A software device driver standard--low level software

would bring device to a standard internal operating system level.

D. Devices would be more standard across option types (e.g. 360).

4. UNIBUS address assignment is a mess. Since every computer we build is essentially unique with differing options, and since all options take up more than the 4K of address space, there is a conflict in the naming of I/O devices. There are several solutions:

A. A rom mapping.
B. Automatic relocation of devices.
C. Take a larger address space for I/O.

5. Multi-computer systesm. Two UNIBUSSES have an inherent deadlock problem. That is A and B make simultaneous requests, A can't get B's UNIBUS, nor can B get A's to complete the transfer.

In the case of the 11/85, requests are separated from access, and requests can be passed among busses without deadlock.

6. Multiprocessors on a single bus or multi-port memories. CSS has built some. Here we intend to provide for them.

7. Timers would be useful.

8. Software coding conventions would increase the ability to share software.

9. There are many operating statistics regarding frequency of use. These statistics have been employed in the design of the new instruction-set. Some of these include:

A. More generality of operators and data-types. This will simplify code generation and also aid the user in remembering the machine.

B. Addressing of floating point array elements by context. This eliminates the ASL's to get the index right so it can be added. This gets 10%-20% on some benchmarks.

C. Dedication of 2 more registers with support by modes to access the stack and arguments with shorter addresses.

D I G I T A L INTEROFFICE MEMORANDUM C1585

PAGE 1

SUBJ: VAX ARCHITECTURE (HRDWR/SOFTWR) DATE: 07-28-75
FROM: GORDON BELL
EX: 2234
MB: ML12/A51

TO: FILE

We have produced much documentation on the hardware architecture. Enough so that implementers can start to work so that we can interact with them. The software architecture is marked with great gobs of milling inactivity. The hardware architecture is described in terms of surrounding goals, constraints, and the technology environment for the 1975-1985 time scale. Most of the instruction set is completed and encoded, and the virtual addressing use and mechanism, though designed, is about 2 weeks away from description for review. We held our first corporate-wide (35 people for 3/4 day) design review (hearing).

So far, we appear to get a 1/3 reduction in code size and running time as compared with a comparable PDP-10 and 40% to 50% reduction over a PDP-11 for FORTRAN, while giving the user 29-bits of memory address space. While these measures are relatively spectacular for an instruction-set, note that if we didn't build the machine, and used a PDP-10 instead, technology evolution would give us the same gain 2 years delayed.

As an architect, I'm helping provide the best 11 follow-on machine that is similar to an 11 so that a user recognizes it as such.

As a business person, I'm terrified at the amount we'll spend in getting a 3rd machine to support beyond 10 and 11--also the risk is enormous. The 11 software support is thin and this will further stress it.

As a user, I doubt if I'll turn in my PDP-10 account # for a a number of years. ALGOL, APL, BASIC+, COBOL, DBMS,...SIMULA plus lots of applications are most important to me and I don't see 8-bits versus 9-bits, or any OP-code at all except a language's. We're dead if I'm anywhere near a typical user who just wants to get work done and not bit hack.

As head of development, I see 4 years of shear hell ahead for us all, and I expect super-human support.

GB:mjk

~~Copy + return both.~~ ✓

VAX

To: Daley, Keating, Vlack, Mikelsi, Stannard, Stewart, Hoffman, A. Miller, BJ, Hastings, Snyder, Conklin, Cutler, Cardinal

An example of using goals/constraints etc in the Requirements setting phase (6).

May 1975

gBell (marked up 8/80).

SUBJ: C. 11 CONSTRAINTS, GOALS, IMPLICATIONS AND SOME ALTERNATIVES [Rev.1]

To: VAXB

The following memo gives our view of the 11 VAX project definition in terms of the goals and constraints for it. For the memo the following definitions are useful:

I'd like to see this kind of document be kept up to date during a project. (I used it in 40 of VAX/VMS.)

1. Constraint(C): an absolute limit that a design can't exceed (e.g. current 11 user mode programs must run in new machine).

2. Goal(G): a value that a design will attempt to achieve or exceed (e.g. the machine should be implementable over a wide range).

2A. Non-goal(NG): not specifically precluded, but either difficult or impossible to achieve.

3. Implication(I): given a goal or a limit, one of the consequences. In the case of an implication from a constraint, it becomes a decision (there has to be a mode for interpreting current, 16-bit instructions). In the case of a goal, an implication is a likely decision if the goal is adhered to.

4. Decision(D): a value of a design parameter that has been selected (see example of implication).

5. Remark(R): comment on a statement.

6. Alternative(A): some possible choices (implication for satisfying goals). There may be multiple implications on non-exclusive basis.

7. Fact(F): almost synonomous with external constraint (e.g. a 32 K byte memory will sell for $600 in 1980.)

7A. Conflict(CF): specific goal/constraint/implication/fact conflicts with another.

8. 11 (16-bit): the current user machines.

9. 11VAX and extended architecture, whose most prominent characteristic is a greater than 16-bit virtual address and that will run 11 programs with a "high degree" of user compatibility.

10. User mode: the environment provided by an operating system for user level programs (e.g. RSX-11/D/M user mode tasks).

The following format is used:

A set of goals and constraints are given for 11 VAX architecture, market environment, user environment, implementation, etc. Following these goals and constraints is a list of remarks and facts which may lead to various implications (and decisions).

General Design Practices

G0 Provide cost/effective computer systems.

G0.1 Minimize memory space requirements (through various mechanisms like better instruction-set encoding and more operators)

G0.1.1 (IS) the new ISP should encode algorithms in the same sapce as an 11, while providing for 32 bit addresses (instead of 16).

G0.2 Minimize processing time requirements for various overhead functions (e.g. process context switch, memory management, better instruction set.

G0.3 Minimize cost of various mechanisms while satisfying functional requirements.

G0.4 Tend toward general mechanisms, while providing specific use. There is a tradeoff between highly specific, non-general schemes which satisfy a particular use and very general systems that permit many uses. We should tend toward providing general mechanisms which do not preclude eventual use; however, each mechanism must be applicable to our specific, immediate use.

G0.5 Minimize programming time. By 1980, people will be just about priced out of the market, relative to machines. A software support specialist will go for $500/day (now $320/day), and an extra 4K of ROM will sell for $150 or less.

I0.1 Tradeoff in nearly all cases will be to more hardware in order to save programming time (and space). Currently microprogramming bits (ROM) cost about 4-8 * conventional bits (actual cost (*2) * increased inefficiency to express same algorithm (*2-4). As a by-product, the algorithm can be interpretted as a factor of 2 to 10 faster, and use less Mp bits. A small implementation on LSI-11 might require 2K words X 2W2 bits (as opposed to 1K now).

G0.6 Provide a machine in which there is significantly more code sharing than any other system on the market or in existence.

11VAX ARCHITECTURE GOALS/CONSTRAINTS

C1 Extend the user VA to at least 24-bits:

The user mode machine should be extended to provide at least 24-bits, of direct, linear addressing. We are assuming the address to be less than or equal to 32 bits.

I1.1 This provides an environment for running larger programs that easily access global variables (e.g. FORTRAN, COBOL).

I1.2 Registers which manipulate addresses will have to

be made available with >16-bits.

I1.3 Instructions to operate on these new data-types, i.e. long addresses are necessary.

I1.4 We are not looking to automatically (in hardware) map an entire file into a user's address space. This would require >32 bits (4 billion bytes) over the machine lifetime

R1.1 There are several implementations that provide large user environments (e.g. 32-bits), but it is segmented into either:

A. 13-bits--now, but could be extended to operate dynamically as originally proposed, hence a program would change the segment registers..
B. 16-bits--a proposal by C. Mudge (program must change the segment registers).

R1.2 At the current time it appears that changing the address space to 32 bits affects nearly all software. The degree is unknown and appears to be quite independent of the extension method.

G2. Long life expectancy, with an implementation now and in 1980:
(In general, the goal is to not limit the life in arbitrary ways by addressing, and other decisions (e.g. using all OP codes).)

F1. Lower System Prices
The system price of a 1 megabyte machine is likely to be <100K in 1980. The system price of a 32Kbyte machine is likely to be $3K in 1980.

F2. Block oriented memories such as CCD and/or magnetic bubbles may be available for the first implementation at costs below random access.

I2.1 Yes--there are hardware implications vis a vis context switching, page faults, etc.

G3. Compatibility across a range of 1000:1.
The architecture should be cost-effective for implementing over an extremely wide range of sizes.

I3.1 Several different bus types will be required to handle range of memory sizes and processors.

I3.2 Multi-Pc's are one way to provide for the extra performance needed at larger memory configurations (e.g. 1 Mbyte/$100K).

I3.2.1 (VA) Multi Pc's used for parallel processing (parallel tasking) requires intercommuni-

cation via data and program sharing (via pages and/or segments).

A. Consider these alternatives:

STRUCTURE	PC	Pc.Perfs (in Mips)	Mp.size (K bytes)	Mp Implied PcPerf. (in Mips)	1980 System Price(K$)
LSI-11	WD	.05-.1	8	.008-.016	.8
LSI-11	WD	.05-.1	65	.06-.13	6.5
Unibus	?	.2	32	.03-.06	3.2
Unibus	?	.4	256	.25-.5	25.6
11/85 Bus	1 Pc	1.0	1000	1-2.	100
11/85 Bus	4 Pc	4.0	4000	4-8	400

Assumes: 1 inst/sec requires 1 to 0.5 byte of memory.

I3.3 (ISP) The ISP should be sufficiently applicable to both small and large machines. Various techniques may assist implementation at the low end: mapping of a single register array versus 2 now; more consistent definition of condition codes; and, in general all the things that imply ad hoc logic.

I3.4 There must be room to expand such that high end can be increased with more specialized operators. See I6.2

G4. Programs written on a large memory machine shall run on small memory machines provided the options and file space are adequate...and visa versa.

have accomplished running large LISP programs in 256 Kbyte Nebula.

This is an unusually ambitious goal for the system range we are considering.

I4.1 A user of a small system (i.e. an 8K word LSI-11 with a floppy and CRT) would be able to do program, checkout, and run a large program. Generally, the program would be executed at the facility; but the user could at his choice, move to any system in the range where the performance, cost, or cost/performance was better for the problem.

I4.2 A user of a large machine could move to a smaller machine and obtain the same results but not in the same time.

I4.3 All machines would be available with all OP codes and registers. In machines like LSI-11, the extended OP codes registers, and other features defining the architecture could be in memory, and the extended OP codes would be hardwired or executed either with microcode or software.

we didn't do, but had the idea that the military used in theirs.

I4.4 When running a program requiring a small part of the system's functional capabilities, it should do so

at minimal loss of performance. In this way we are not forcing a performance--function tradeoff to always favor function at the low end.

I4.5 (VA) Demand paging or demand segmentation are required.

MARKET SPECIFIC ENVIRONMENT GOALS/CONSTRAINTS

G5. Significantly improve real time response:

A1. Provide faster context switching (e.g. multiple register sets).

A2. Provide cleaner context, hence, easier to switch.

A3. Provide faster response to user I/O.

I5.1 Direct (protected control of I/O). This might be done as: new devices, a central map, special I/O segments, or special-shared exec procedures which deal with physical addresses at lower (protected) levels.

✔ I5.2 (VA) Minimize context switching.

I5.2.1 (VA) Map operating systems with user process.

G6. Enhance machine ISP for our varied markets.

✔ I6.0 (ISP) There are some general functions that would greatly improve ISP for all markets, e.g. better support of languages via a calling mechanism. These are considered highest priority.

cobol emulation — trying now to enhance for highly vector processing tech. mkts.

I6.1 Provide instruction-set data-type space for additional operators (e.g. strings, long integers, decimal arithmetic).

I6.2 (ISP) Leave room to handle at least 25% more data-types instructions.

(note G+H)

G7. Improve RAS over current structures

✔ I7.1 Provide for redundant single computer systems (e.g. multi-memories, multi-Pc's, and multiport I/O).

I7.2 Provide for interconnected systems with redundancy of path and components.

I7.3 Significantly better enforced protection among parts of operating systems and user programs to encourage information hiding. Protection should exist to the point of preventing faulty arguments from destroying parts of the system, without explicit checking by the program.

I7.4 The direct manufacturing cost in general, will increase for no increase in benchmark performance. However, total cost, including warranty may be less.

I7.5 There are more implications for each implementation.

I7.6 We should attempt, where possible, to protect against common forms of faulty programming. For example, we could provide undefined values of a variable (like the current floating point), such that attempts to process (i.e. read) undefined values would trap. calls within a process and among processes.

USER-LEVEL COMPATIBILITY

C8. No Apparent Customer Training For 11VAX:

No apparent training will be required for customer programmers to take advantage of the extended functions of 11VAX.

I8.1 11VAX Instruction-Set Processor (ISP) will have at least the same registers, the same instructions, and the 11(16-bit) ISP.

I8.2 New instructions will be of the same style, format, and permit the same type of programming.

I8.3 An assembly program coded with minimal, straightforward conventions will execute on either 11 (16 bit) or 11VAX with reassembly.

I8.4 (ISP) ISP elegance is needed such that no apparent increase in understanding is required for increased data-types and operators.

ISP elegance is a combined quality of instruction formats relating to mnemonic significance, operator/data-type completeness for all data-types, consistent method of address formation, etc. Lack of modal idiosyncracies, ad hoc registers, etc. all assist in minimizing instructions and aiding the ability to "know and remember" a machine-- paramount to using it effectively.

I8.5 (VA) The mapping scheme should be understandable (actually a goal). Mapping and process is probably the most complex part of total system architecture. The parts provided in the ISP architecture have been traditionally simple, and the complexity is usually buried in the operating system architecture. With advent of larger memories, better software management and faster memory management hardware are required. This in turn increases the lower level complexity.

G9. Run 16 bit User Mode Machines Defined by Operating Systems (e.g. RSX-11) Interface:

Unmodified object programs for the 11, 16-bit user mode interface that do not reference the I/O page will run.

I9.1 16-bit binary compatibility mode must be provided

in hardware in a KT11-like environment i.e. 8 segments, user mode interface, current 11 Op code decoding). Segments need not be as small as 32 words.

I9.2 Operating systems will provide for the "11(16-bit)" environment, and since the machine is to provide 11VAX, both environments must be provided.

I9.3 11VAX user machines: Users will "be able" to write user mode programs that will run on both 11VAX and older 11,16-bit user mode environments. (See I8.3).

I9.4 Object level compatibility will exist for RSX-11M to RSX-11D.

I9.5 (VA) Provide same RSX capabilities in subsequent systems. Data and program sharing at the user level are explicitly provided and used.

G10. The 11 VAX user mode environment will be a proper subset, albeit one which requires reassembly of the 11,16-bit user mode interface:

Programs written following suitable coding conventions for either user environment shall run with reassembly. The 11VAX environment will have larger addresses. Such an approach would be used for writing libraries utilities, etc. which have to run in both environments.

G11. The system software will support existing tasks with new (or modified) tasks which use extended addressing and other new features.

I11.1 11VAX and 11 (16 bit) subprograms cannot be mixed within a single task. Parameter passing (of addresses) is fundamentally incompatible.

I11.2 The operating systems will have to be modified, not just converted, to support both compatible and enhanced environments.

Constraint

↓

R. There are many problems--we don't know how now.

C12. Hardware peripheral compatibility with current 11's:

11-peripherals shall run with no hardware changes. Customer and DEC could carry peripheral designs to 11VAX. Software changes will, in general, be needed.

I12.1 Unibus and Massbus are correct main interfaces. (Sercon and Serial)

I12.2 Some kind of connection and mapping mechanism will be required to interface to a processor or computer bus in the high end implementations for Unibus devices.

I12.3 LSI-11 peripherals will be used in low end systems.

G12A. Minimize Excluded Hardware Features:

The hardware being omitted at this time should be stated: I and D space; concept of previous process; UNIBUS map as implemented in 11/70 (some form is necessary); and multiple registers ala 11/45.

DEC INTERNAL GOALS/CONSTRAINTS

G13. Existing Privileged Programs (Operating Systems) should run with minimal modifications.

I13.1 They will, with the limits of previous constraints.

I13.2 Obviously, there are more capabilities, larger addresses, and possibly a different memory management method, hence, there must be changes. Simply extending all addresses to 32-bits, without other changes (e.g. memory management) will be unworkable.

CF/I13.3 (VA) Conflicts with I5.2.1, while an existing operating system can be recoded to run, the memory management designs we have considered provide a system-wide VA for a process (or all processes). This permits an operating system and user process to cohabit a single VA space.

CF/I13.4 (VA) It will be difficult to decide on whether we re-write an existing operating system, or provide an improved operating system.

NG14. Existing Operating Systems will be recoded to provide only 11(16-bit) user mode environment.

R. Although one in a steady state should use an 11(16-bit) instead, there many times when only the limited (16-bit) environment need be provided.

G15. Phaseover from 11 to 11VAX must be minimized.

If we go ahead with 11VAX, we must minimize the perturbation as we move to provide a new user environment. The key to the project is the degree to which compatibility is provided.

I15.1 The 11VAX and 11(16-bit) user environments would be clearly defined, and the number minimized. (The RSX and RSXVX would be proposed as a starting point.)

I15.2 Operating Systems under development would be designed to operate in both the 11(16-bit) and 11VAX environments with minimal change.

I15.3 Coding conventions would be established for transportability to both 11's and the new 11VAX to insure transportability at the assembly language level. All code would first be executed and sold

on 11's, although testing for 11VAX would be impossible.

R. A higher level language (beyond the assembler) could be used to ease transportability.

R. (See I13.3; I13.41)

G16. Maximize the runnable diagnostics with no reassembly.

R. No work so far.

GB:mjk [6/4/75, Version 2, GLSCON.VAX]

SUBJ: RATIONALE OF MAJOR DESIGN CHOICES

This section will include various sections which outline the rationale for various choices that have been made in the design.

The various sections include:

1. Dealing with compatibility models.

2. ISP

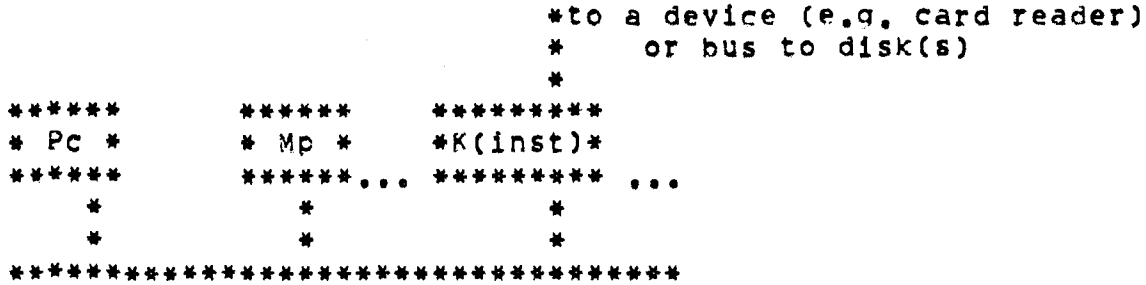
3. Virtual addresses, memory management, and protection.

GB:mjk [6/4/75, MAJORD]

SUBJ: INTER-PMS COMPONENT TRANSFERS ON DEC COMPUTERS:
RATIONALE, EVOLUTION AND RECOMMENDATION FOR A POLICY

This memo describes the philosophy that has been used for controlling the transmission of data among the various components within a computer (and especially at DEC). The method has remained relatively constant for about 15 years. As technology has changed recently to offer low cost, fast read only memories, it is time to update the position. We are to the point where nearly all controllers for larger devices can include their own computer which can interpret a program and have the capability of at least current device drivers. This memo will describe the past philosophy and posit, what I believe is, the right way to go in future systems. Recommendations will be given first, followed by the problem, and the alternatives that determine the solution framework.

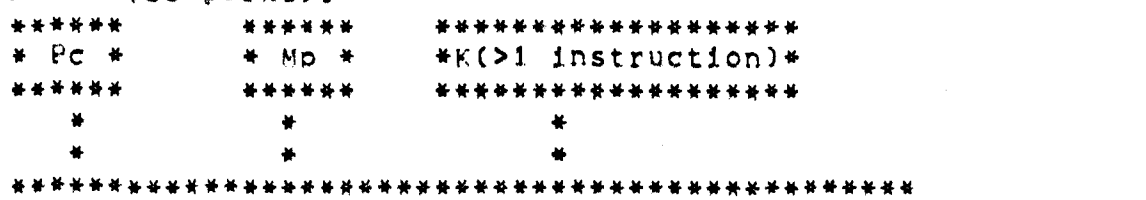
Given the current, 1 central processor UNIBUS system with primary memory modules, and simple controllers, K's for devices; a controller K, may directly transfer data to Mp or it may interrupt Pc. Pc can communicate with K for data and/or control information.



Recommendations (the solution)

0. Define an IO process level interface which is at least as capable as current I/O drivers. Current hardware engineering would be responsible for developing systems and diagnostics to this level. Implementation would be by any of the techniques listed below ranging from totally programmed as with our current systems to separated IO computers with their own microcode programs.

1. Adequate instruction buffering in K. We must add sufficient command buffering in each K, such that a device can operate at its full speed (subject to poor payoff from a cost/performance viewpoint).



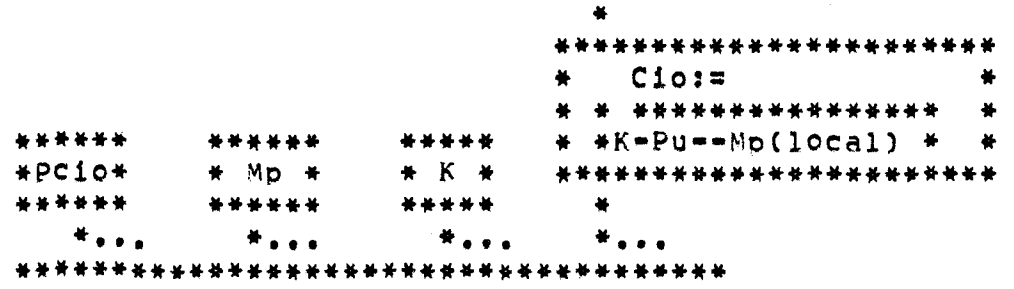
2. 1 instruction interrupt in Pc plus better I/O instructions--Pcio. We can, by a small change to current interrupt vectors, permit a single block-oriented data transfer instruction to be executed at interrupt time. The additional instructions we need are:

A. Block I/O, Byte/word, IO-device-address, word-count and transfer-address. Input/Output a byte/word according to a control word which has a word-count and transfer address. At termination of block, cause a conventional interrupt.

B. Decrement a word in memory and interrupt IO.

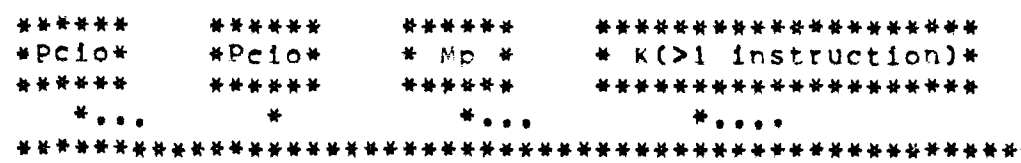
C. Block I/O with character translation. The communications group should specify the operation.

3. Add a fully programmed microprocessor Pu with its local primary memory, Mp (local), which forms a small, fast stored program, computer, Cio. Cio is connected to a control, K, or is part of a control K. With this scheme, IO processes will correspond to at least the current IO device driver level. In essence, Cio will operate on a data structure specifying a job(s) to be done. The program in Cio is fixed. We are currently building controllers of this type for communications.



4. Examine the feasibility of using the small, Cio's, i.e. Demons, on the UNIBUS generally for specific control purposes (e.g. disk management, communications).

5. A multiple processor structure to increase reliability and performance.



The Overall problem

A computer consists of a number of PMS components and the design task is to interconnect them in a "cost/effective" way. This implies:

1. there is a physical structure that permits information to be transmitted among them. The UNIBUS is the most general way.

2. A process (program) in the computer system has to tell the various components that the transfer must take place... i.e. control.

3. According to good engineering principles, the system should be cost-effective:

A. the cost of the transfer, in terms of the resources it uses, must be small.

B. The overall system cost must be small. This can best be accomplished by leaving out components.

4. The overall throughput must be high, which in the case of I/O means greatest concurrency (parallelism).

5. The devices must operate at their own speed unless this cost does not increase the cost/effectiveness.

6. In some applications, it is important to have a minimal time between when an event is signalled until when a response is given by the program. (This also gives high throughput.) This, in effect, minimizes the interrupt response time.

Controls (K), processors (P), and computers (C)

A control (K) is the simplest form of finite state machine. It is given an input (1 or more instructions), it executes them and stops. In our systems, a control is given 1 instruction at a time by a processor (Pc), it executes the instruction (e.g. move a disk arm, print a character, transfer a block of data on Ms.disk to Mp).

A processor (Pc) picks up its own instructions from a list in a primary (program) memory (Mp). It has a program counter, which points to the instruction it is executing (or going to execute). The act of fetching and executing instructions is program interpretation. Thus to give a task to a processor to execute, requires giving it a program...i.e. specifying "how to do it."

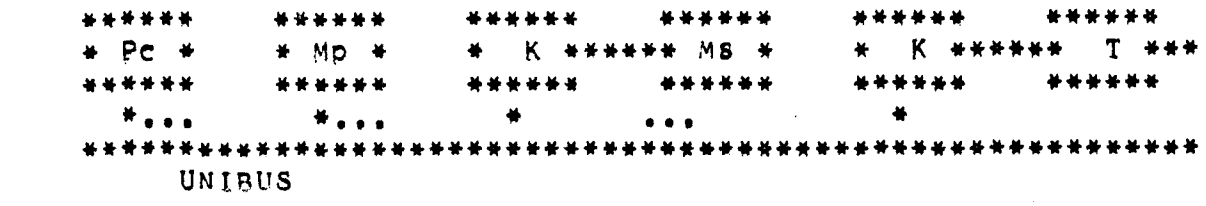
A computer (C) is a Pc-Mp pair with a program(s). In the case of Cio, specifying a task requires giving Cio tabular information (data-structure) about the task...i.e. specifying "what to do", not "how to do it." The assumption is that a program in Cio "knows" about the data structure and knows how without being told.

Pio and Cio are analogous to a procedure-oriented and a report generator-type program language (e.g. COBOL and RPG). In the former, tasks are specified by lists of instructions to carry out the task. The later accepts a template of the result (report) and then proceeds to achieve the goal by extracting the appropriate information from the data.

The Physical Structure Problem

The UNIBUS is the most general interconnection scheme to interconnect PMS components because it permits any device to communicate with any other. It is an obvious solution once the problem is formulated in its most general form.

The general structure is:



There are several kinds of traffic which the UNIBUS (or any other bus structure) carries:

1. Central processors (Pc) to primary (program) memory (Mp)-- in a stored program as the processes are being executed, each processor must access its program and data.

2. Primary memory (Mp) to secondary memory (ms), e.g. disk, via controller (K). In nearly all computers even beginning with Whirlwind, programs exceeded Mp.size that Pc could execute from. It is necessary to move programs and/or data between secondary (backing) memory, Ms, and primary memory (Mp).

3. Non-memory transducers (T), e.g. typewriters, communicate with a program.

4. General control to cause transfers (2 and 3) and generally synchronize with them.

Historical Solutions to the Physical Structure and Control Problem in Terms of Processors

There has been an evolution in structure and in the way initiation and synchronization have taken place with Pc. This has been governed by technology, and has followed this path:

1. Very simple controllers (K's). With the processor stopping to help control each transfer. This also simplifies programming because everything is sequential.

2. Adding interrupts, and more complexity to each control (K) so that it could proceed in parallel. Interrupts were "invented" to synchronize completion with the processor without requiring Pc to wait or poll.

3. Direct memory transmission (DMA-NPR) of information between Mp and Ms (or other device which required very high data rate transfers).

4. The addition of an io processor, Pio (IBMeze=channel) which executes a stored program. Pio has instructions to initiate the controllers, and nominally spends its time passing data from the controller it initiates to Mp. at the completion of data block transmission, it fetches another instruction from its own program. I've been traditionally against this approach because:

   A. It is most costly. (The initial channels on the IBM 7090 were really bad, because they were non-multiplexed; hence only 1 device (e.g. a 150 lin/min) printer could

run at a time. The 360 selector channels are just as bad.

B. They add logical and physical complexity without much payoff. The controller is doing the real work, because of device idiosyncrasies, and all Pio does is buffer and pass data from a control to the memory...something that a buffer and wire will also do reliably and cheaply.

C. As a somewhat (not terribly) intelligent device, they require more communication because they are somewhat smarter than a dumb controller (notice the nice analogy with people). They must be told how to do a job.

D. Since a Pio has the same complexity as the central processor, one might as well use just the central processor. The central processor is the cheapest device in the system because it is already there, and the only time that Pc is expensive to use is when its at full load (i.e. there is no spare capacity).

However, when this happens, the nicest alternative is to merely add a second central processor to do the IO task (and any other tasks). Note the cost is no worse than in the case where we required both an arithmetic and IO processor.

E. There is system cost to have another component type which has to be stocked, diagnosed and programmed. The central processor has to have a program waiting for the IO processor, or has to compile one, or insert one in the Pio's table structure.

Note that when all the work has been done by the central processor, the only remaining work is actually handing the commands to the traditional, low level controller.

One should ask, why have a complex middle-man to which you hand commands, that merely hands commands off to someone else. Why not have Pc just hand commands to K when they're generated? It's clear to me since each small set of commands (a channel program) generates an interrupt back to the CPU, nothing has been gained since the Pc does the same (or slightly more work)...

5. The IO computer Cio. This has been used effectively by CDC in the 6600-7600 series, and we have done this to a certain extent in the PDP-10 and in large PDP-11's where a certain high level function is being performed by a totally separate program and complete process. The control activity is in Cio's memory, and it is told what to do--i.e. transfer a block, not how to do it (e.g. move arm + search + transfer + check).

Notice, the complexity is bounded; we have come full circle, once a Cio is formed. A Cio is precisely a second computer just like the starting point of the most primitive computer (i.e. K simple-Pc), but it is split apart for the sole purpose of I/O task management.

6. Single instruction execution interrupt level. Improved instructions in Pc to handle IO transmission. This was done in the PDP-10 and in PDP-8 for communications I/O, such that instead of executing a program at interrupt time, a single instruction is executed. This slipped by me in the initial implementation of the 11 and should be included at this time. The most conventional use is to interrupt, and then execute a single Block Transfer In/Out instruction. The instruction transfers one word under control of a word count, and location pointer in memory. It has been used extensively byour competitors--the most notable has been Interdata, who added instructions to input characters from communications lines and perform translation, and store them in memory.

7. Specialized processors which interpret programs for a particular task. The GT40 is a good example of this. Because the instruction-rate is high, a complete processor is required. A typical instruction draws a character or line, or manipulates a list data structure defining the picture.

Evolution of Controllers (K)

While the above section discussed the evolution of the concept, and location of control, device controllers have varied considerably. Controller complexity has been influenced by technology, thus "control" can be distributed among hardware in a processor, a processor and a program, a specialized processor, or completely in an autonomous controller.

Our controllers have evolved to the execution of single instruction (e.g. find a disk block, transfer a block from disk to a block in memory, output a character on a given LA36). I believe that controllers have and will evolve along the following lines:

1. K(simple). Simplest control where the CPU or a program handles most of the device control. In essence, all that K has are buffers to staticize information and convert signal levels according to the device's needs. Input converters sense the device's output and read them into another part of C.

2. K(1 instruction) Current controllers can execute a complete instruction on a data type that is known to the device being controlled (a character on the LA36, a line on a line printer, or a block on a disk or tape).

3. K(> 1 instruction) Current controllers but with sufficient command buffering (>1 instruction) such that the devices will operate continuously. We may have been minimizing controllers to the extent that system performance is degraded. For example, since a disk is usually the limiting component, and we use alternate blocks, the transfer rate is limited to be 1/2 the maximum, or conversely, the throughput is down by a factor of 2.

4. K's formed as K-Pu-Mp(local). This corresponds to at least the device driver level task. In this case, a special IO computer, is formed by a microprogrammed processor, Pu, which has a local primary read-only memory. In haveing a

program, there are several possible uses of the increased complexity:

A. The control program formerly in Pc-Mp or Pio-Mp can be located in Mp (local).

B. K is told what to do, not how to do it--it knows by interpretting its own program in Mp(local).

C. The control program can diagnose the device.

D. The control program can fetch a data structure (task) from Mp (global), and manage buffers, do error control, etc.

E. More optimization of device control (e.g. disk transfers based on min. latency via a queue of jobs) can be done.

Summary

The dimensions of control choices evolves along these lines for a controller (K) and the corresponding control in the processor (P).

*	K simple	*	* Pc (with embedded K)
*	K (instruction)	*	+ * Pc (interrupts + I/O programs)
*	K (>1 instruc)	*	* Pcio (1 instruction at interrupt)
*	K-Pu-Mp(local)	*	* Multi-Pcio
	(note equivalent		* Pio (e.g. channels0
	to K simple +		* P special
	Cio)		* Cio (separated computer with local Mp for I/O process control

GB:mjk [6/4/75 Version 2, PMSSTR]

SUBJ: THE PMS STRUCTURE OF 11VAX MODELS

Introduction

Although the implementation of a computer is presumed to be independent of its ISP (often called the architecture), it is clearly not, since one must worry about how something is to be built, when a specification is made. This document describes various bus structures which we currently support within the DEC environment, and makes various assumptions and recommendations about their future; thus we can proceed to define the architecture assuming them and their idiosyncrasies.

We must also make certain assumptions about the kind of structures we expect to build from these busses, in order to proceed with the architectural definitions 11VAX goal, G12 discusses the need for hardware compatibility with existing 11's via the UNIBUS.

Proposed Use Policy for Corporate Busses

Fundamentally, we will continue to use the Q-, U-(UNIBUS), and B-busses (alias the Unicorn bus, alias synch. backplane interface).

1. Q-bus (LSI-11)

   Policy: Very low end, tightly coupled, bounded system bus trades off for cost.

   Supports: Serial line, parallel interface, lots of Mp types, RX01.

   Future Support: line printer, special a/d/a, digital I/O (IPG), scan graphics, (LDP) sync and SDLC communication, IEC (Instrument bus), IPL?, RSL, RK05?, TS03?, serial bus? May get more COMM for front end work.

   Problems: VAXable? Can we not evolve it for all purposes?

2. U-bus (mid-range 11VAX)

   Policy: Cost/performance mid-range system, open-ended, used for mid-range memory bus for Pc-Mp, Mp-Ms, and all other peripheral transmission. Will continue as main workhorse.

   Supports: all devices.

   Future: remove low speed devices to S-bus.

   Problems: Is it reliable enough in >1978? Move to multi-processors

3. B-bus (for high end 11VAX)

   Policy: max. performance system, bounded as a computer memory bus for Pc-Mp and Mp-Ms. Non-Sercon-bus and non M-bus peripherals will use U-bus.

   Supports: Sercon-bus; M-bus; U-bus.

   Future: S-bus and less U-bus

   Problems: Will we run into a ports number limit at very large Mp.size? Is it fast enough? How do we get more speed in >1980 w/o redoing everything?

4. S-bus

   Policy: all low speed I/O. Low cost, very reliable, bus for interfacing to system, especially independent of distance. Wait for operation and special LSI chip. Move to put all devices in low end when successful. Should be lowest cost scheme for interconnecting low speed peripherals.

   Supports: IPG equipment, cluster for EIA terminals?, computer-computer?

   Interfaces: U-bus, Q-bus?, and B-bus?

   Propose: all low speed equipment (paper tape, cards, line printer, TS03, RX01, all terminals).

   Problems: schedule, will it really ease software support?

5. M-bus (Massbus)

   Policy: use for high speed disk and tape for U-bus, and B-bus.
   Problems: how much does this cost us in peripherals.
   Use: controllers for U-bus, B-bus.

6. Sercon-bus (RK06 disk bus)

   Policy: RK06,..., RSL?
   Use: controllers for U-bus, B-bus, Q-bus?

General 11VAX structures
---

0. All addresses in Q-, U-, and B-bus system are physical addresses. May eventually use virtual addresses in controllers when they have stored programs for control to cope with complexity.

1. There are instructions to get and transmit physical addresses when accessing I/O device address registers. With more busses, computers, and more controllers for each, we can get into incredible support difficulty vis a vis different devices and their names. This problem includes the dynamic device assignment which the UNIBUS currently has. It also has to include the problem that a device may actually be controlled (by hardware) differently on different busses.

Also, controllers will evolve to have more complexity.

The control (and device) naming problem probably has to be solved by a ROM device directory such as proposed by Stambaugh et al. Note, that such a scheme has to be field retrofitable, and it also permits us to send matched software with hardware; hence, we can have a policy which supports our own hardware at a different price than that of the add-on people.

2. Controllers are evolving in complexity and behaving like special, IO computers. This can only be supported by having a standard internal software interface for all peripherals which is IO process structured at a level which includes the current IO drivers.

3. As we interconnect system busses B-bus to U-bus for peripherals, there has to be a standard method to avoid the proliferation problem mentioned above.

4. Current multiprocessors are predicated on a single bus structure (i.e. Q-, U-, and B-bus). If we build multi-processor systems with cross-point switches (e.g. multi-port memories), some additional problems must be solved (e.g. only certain processors can field interrupts or run particular I/O devices, cache update).

5. Coupled computers using methods like the UNIBUS window can be built easily in the case of B-bus systems since a large physical address space exists and the B-bus breaks requests and response into two transactions.

UNIBUS Enhancements for VAX

Since our mid and large machines depend on the UNIBUS, we must examine the needs. Some of the problems we have to address are:

1. Device assignment as mentioned above.

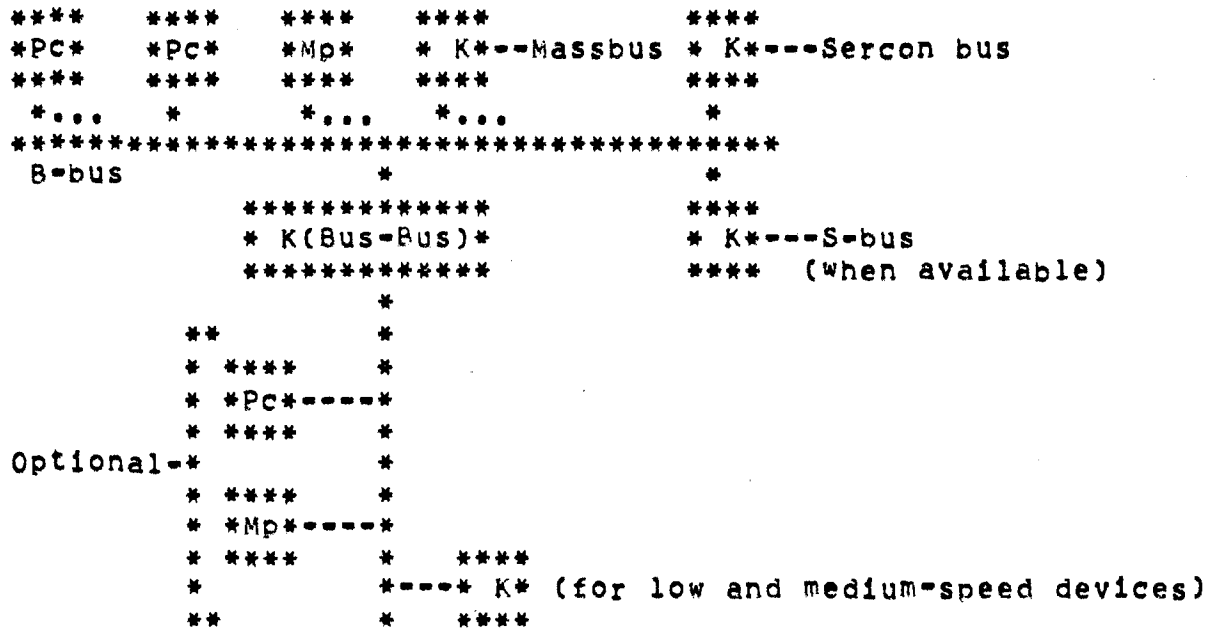
2. With paging systems, it is necessary to transfer a full segment to Mp easily. Here, we assume that Mp fragmentation occurs; a segment of up to 2^16 bytes is contiguous on Ms.disk. This can be solved by a modification to the U-bus repeater to include a map of 2^9 entries for each page in the 2^18 bytes of the U-bus address space.

In this way, each page of the second UNIBUS is mapped into an arbitrary page of the first bus.

3. There will be multi-Pc UNIBUS systems.

The Structure of B-bus Systems

0. The general system structure will be:



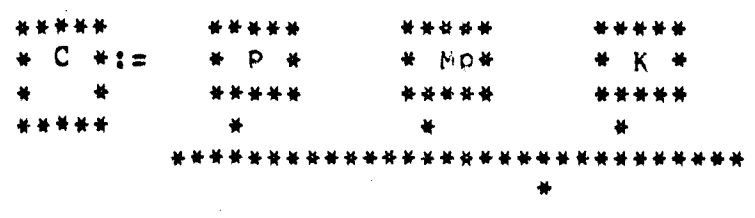
A large system would be:

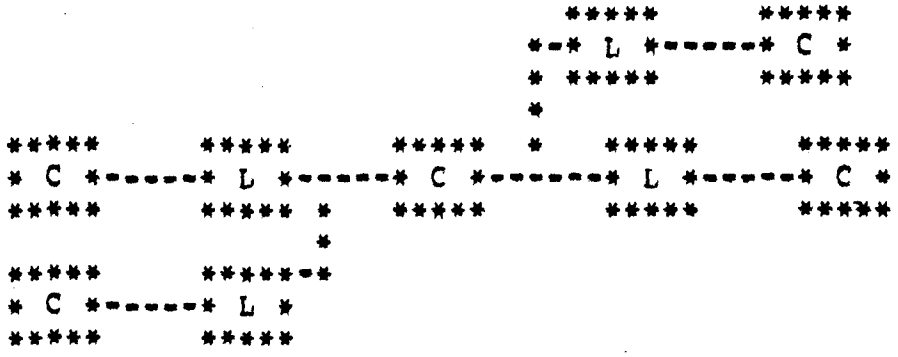
4 Pc, 4 Mp, 4-6 M-bus, 2 U-bus,

Here, we assume the B-bus peripheral K's may have multi-instruction buffering, and may possibly fetch a data structure from Mp.

1. Note, that by having system-wide unique address for all physical components, computers can be coupled almost arbitrarily, and messages will be stored and forwarded.

A computer is:





2. K (bus-bus) B-bus support of U-bus peripherals.
There is a fundamental problem of physical address space management because each U-bus has addresses $0<2^{18}-1$, and B-bus has addresses $0<2^{24}-1$. Assume each UNIBUS is mapped into some part of the B-bus address space. There are two cases of interest:

   A. The UNIBUS has no computer:=(Mp-Pc). In this case, the purpose of K(bus-bus) is to make all K's on the UNIBUS appear exactly as though they were attached to B-bus. This undoubtedly requires a high level of control in K:

      i. A few U-bus addresses for the devices are mapped into B-bus space. Pc accesses devices exactly in the way it is used to.

      ii. Interrupts from U-bus are re-mapped back into B-bus space.

      iii. NPR devices on U-bus require extra address bits in order to access all of B-bus space. Hence, physical address bits are concatenated with those coming from U-bus via a map.

   The purpose of K(bus-bus) is to make the mapping be transparent to B-bus. Therefore, we might assume there is some level of complexity in K to deal with mapping and extending addresses, command and data buffering and interrupts.

   B. UNIBUS has a computer (i.e. Mp-Pc) attached to it. With this structure, 3 modes of operation are useful:

      i. K's are on B-bus as indicated above, but much of the K(bus-bus) control complexity is handled in a program on the UNIBUS-based computer. This mode is necessary, if NPR devices are used, so that data can be transferred directly to B-bus memory.

      ii. A task is executed in the UNIBUS computer, and a resident program transfers data to B-bus Mp. This may entail buffering in both U-bus and B-bus memory.

      iii. A task is executed in the UNIBUS computer, and the appropriate B-bus computer accesses the data (usually a buffer) directly. Note, that this avoids any redundant buffering.

Scheme B, is the most general, though more costly (>Mp+Pc). It does, however, give additional processing capability, and decreases the interrupt requirements for the B-bus Pc. A design should be carried out for both schemes. One would expect the following mechanisms in K(bus-bus) for scheme B.

i. Each unibus is mapped into a B-bus space of 2^18-1 when accessed by Pc(B-bus).

ii. The UNIBUS configuration would have relatively small memories, and an unused memory UNIBUS address space would be mapped on a page-by-page basis into B-bus address space. The map could be manipulated by either processor, since it occupies a fixed position in the B-bus and U-bus space. In this way, NPR transfers could be made to either memory.

iii. Interrupts would undoubtedly be processed by the Unibus computer. In principle, certain BR lines could be routed to the B-bus, or a program in the UNIBUS computer could re-route interrupts to B-bus via the K(bus-bus).

GB:mjk [6/4/75, PMSVAX]

SUBJ: A BYTE ORIENTED ISP PROPOSAL FOR THE EXTENDED PDP-11 ARCHITECTURE

INTRODUCTION

This memo documents still another ISP alternative for the extended-11 architecture. Like the others, the byte oriented ISP preserves (and extends) the -11 data types, operators, and addressing modes. Superficially, it appears somewhat different than the current-11 in that the instructions are multiples of bytes rather than 16 bit words. However, stylistically, it is most consistent with the basic -11 architecture in that operators, data types, and addressing modes are independent, and in that each operand of an instruction is specified in a general way.

There are a number of objective and subjective goals for any new ISP. These include:

1. "Cultural" compatibility with the current -11--a PDP-11 programmer should be able to program efficiently in the new ISP with a minimum of relearning and changing of programming style.

2. Bit efficiency--compared to the current -11, algorithms should be encodable in the same or fewer number of bits. The efficiency should not be based on tricks and fully useable by high level language processors.

3. Elegance--the ISP should be clean, simple, and systematic.

4. Extensibility--room should be left in the ISP encoding so that additional data types and operators can be included in a manner consistent with the originally defined data types and operators.

5. Implementability over a range--the ISP should be effectively implementable over the range of systems envisioned for the extended -11 architecture.

It is the opinion of the author, that of the alternatives presented, the byte oriented ISP best meets these goals.

ADDRESSING

In the byte oriented ISP the virtual address is 32 bits. As in other ISP proposals:

1. The general registers are extended to 32 bits,

2. Index constants and indirect words are 32 bits (although a short form is defined),

3. All address arithmetic including PC incrementing, auto incrementing/decrementing, stack pushing/popping, and indexing is done in 32 bit mode, and

4. The size of the PC saved and restored in subroutine and trap sequences is 32 bits.

Specific to the byte oriented ISP:

1. Short form index constants are 8 bits, and

2. There are no short form indirects (because virtual addresses are a full 32 bits).

DATA TYPES

The data types defined for the byte oriented ISP are the same as those in the alternative proposals (although the notation is slightly different):

1. byte (B) - 8 bit byte
2. integer (I) - 16 bit integer
3. long integer (L) - 32 bit integer
4. quad integer (Q) - 64 bit integer
5. floating (F) - 32 bit floating point
6. double floating (D) - 64 bit floating point

It is assumed that decimal arithmetic will be supported by a pair of instructions which convert from a decimal ASCII string to quad and the reverse.

PROCESSOR STATE

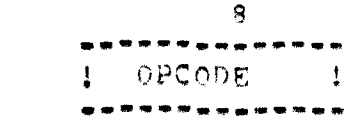
The processor state is represented by 16 32 bit general registers and a 32 bit PSW (which is composed of 16 bits of the current 11 PSW and a 16 bit register status word. See W. D. Strecker, Subroutine Call and Return Mechanism, 5/23/75). For arithmetic operations on 64 bit data types, pairs of adjacent registers are used. Specifically, if register R is specified in a 64 bit operation, registers R and R+1 mod 17(8) are used. For addressing purposes the registers are divided into two classes: index registers and base registers. The index registers are R0 through R7, while the base registers are R10 through R17. The type of register is implied by the addressing mode. Hence, only a 3 bit register field in an instruction is required. By convention, certain of the registers have predefined meanings:

R17 - program counter (PC)
R16 - stack pointer (SP)
R15 - argument pointer (AP)
R14 - local pointer (LP)

INSTRUCTION FORMATS

There are three basic instruction formats:

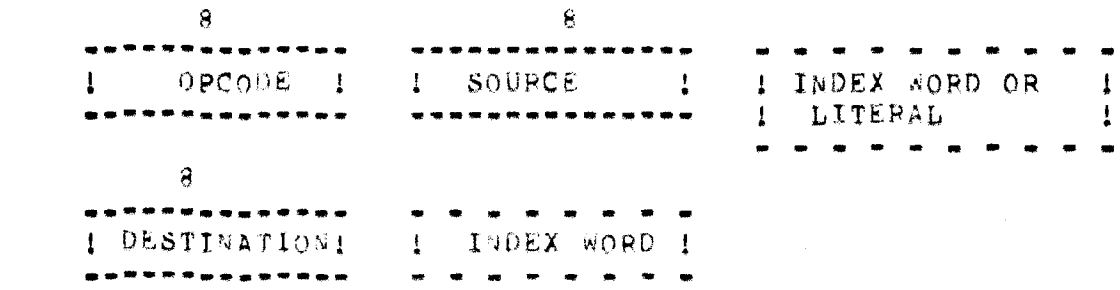
1. Zero operand



2. One operand

or

3. Two operand



Hence the instruction length (without index words or literals) is 8, 16, or 24 bits as opposed to a fixed 16 bits on the current -11.

SOURCE AND DESTINATION SPECIFIERS

The formats of the source and destination specifiers are:

1. Source

where M' is an extended mode field. The modes encoded in M' are:

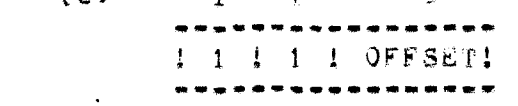
M'	TYPE	MODE
0	BASE	R+10(8)
1	"	(R+10(8))
2	"	(R+10(8))+
3	"	@ (R+10(8))+
4	"	- (R+10(8))
5	"	@- (R+10(8))
6	"	X1 (R+10(8))
7	"	@X1 (R+10(8))
8	"	Xs (R+10(8))
9	"	@Xs (R+10(8))
10	INDEX	X1[R] context
11	"	(Xs(LP))[R] context
12		(not yet)
13		(defined)
14		( )
15	"	R

The notation X1 and Xs stands for long (32 bit) and short (8 bit) index costants respectively.
Context means that the index register is IMPLICITLY presnifted by the size of the operand (i.e. shifted left two for single precision floating operands). The definition of mode 11 will be clarified in another memo [W. D. Strecker, The Skeleton Call Instruction, 5/21/75]
For floating point operations mode 0 specifies one of eight 64 bit floating point registers.

(b)

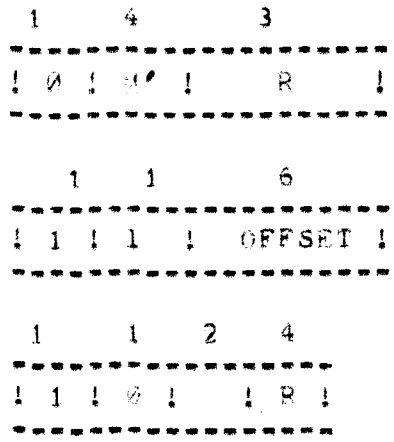
where the literal field is a six bit two's complement integer.

(c)



where the offset field is an unsigned integer offset off the LP>
Again refer to the 5/21/75 memo for clarification.

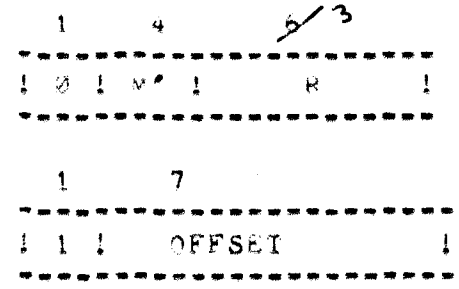
2. Destination



This format fills the position taken by literal in the source specifier. There is a certain (desireable) assymetry between memory-to-memory and memory-to-register operations. For example, a 16 bit add integer memory to memory clearly generates a 16 bit result and appropriately sets the condition codes. An Add 16 bit integer memory to register (32 bits) needs specification as to the size of the result, and how the condition codes are to be set.

When operands are less than 32 bits, accessing a register destination by mode Ø or mode 15 sign extends the operator through 32 bits, generates a 32 bit result, and sets the condition codes on that result. When an operand is less than 32 bits and a register destination is accessed by the above format, the register is assumed to be the size of the operand, the result is the size of the operand, and the condition codes are set on that result size.

3. Branch Destination (Destination form used by branch and JSR instructions.)



where offset is a signed byte offset off the PC.

INSTRUCTION SET

1. Two operand format — OPCODES

(ADD)	( )		
(SUB)	( )		
(MUL)	( )		
(DIV)	(B,I,L,Q,F,D)		30
(CMP)	( )		
MOVEXP	(MOVE EXPONENT)		1
MOVA*	( )	(MOVE ADDRESS)	
BIC	( )		
BIS	( )		
BIT	( )		
XOR	(B,I,L,Q)		44
ASH	( )		
LSH	( )	(LOGICAL SHIFT)	
ROT	( )	(ROTATE)	
IMUL	( )		
IDIV	( )		
MOV*	( )		

*MOVAF = MOVAL
MOVAD = MOVAQ
MOVF = MOVL
MOVD = MOVQ

(MOD) (F,D) — 2

MOVC (F D) (MOVE CONVERTED)
(D F)
(B F) — 18
(F B)
(B D)
(D B)
(I F)
(F I)
(I D)
(D I)
(L F)
(F L)
(L D)
(D L)
(Q F)
(F Q)
(Q D)
(D Q)

JSR*
*uses branch destination and any source (except literal) for linkage)

INCR* (B,I,L,Q,F,D) (increment and compare) 6

*uses two destination fields. Increments first destination and compares it to second: used for loop control.

2. One operand format (destination) OPCODES

(INC)	( )	
(DEC)	(B,I,L,Q,F,D)	30
(NEG)	( )	
(ABS)	( )	
(TST)	( )	
(CLR*)	( )	
(ROL)	( )	
(ROR)	( )	
(COM)	( )	
(SXT)	( )	
(ADC)	(B,I,L,Q)	36
(SBC)	( )	
(ASL)	( )	
(ASR)	( )	

what about shifts

*CLRF = CLRL
CLRD = CLRQ

RTS (see definition of JSR) 1

SWAB 1
SWAI (SWAP INTEGER) 1

SCC* (set condition code) 1
CCC* (clear condition code) 1

*uses destination field as source

3. One Operand instructions (branch destination) OPCODES

(BRANCH)
(CONDITION) 15

(Note that conditional branches have either a 7 bit offset or a full MR destination. Hence, there is no jump instruction and conditioned boundries can reach any address in the vertical address space.

JSR (JSR PC,) 1

4. Zero Operand Instructions

RESET	1
WAIT	1
NOP	1
HALT	1
RSB (RTS PC)	1
COROUTINE (JSR PC, @(R6)+)	1
RTI	1
RTT	1

	196

This leaves 60 opcodes for completion (a few necessary operators are not yet defined) and future extension.

WS:mjk [6/4/75 Version 2, BYTEIS.WS2]

SUBJ: SUBROUTINE CALL AND RETURN MECHANISM

In an earlier memo (THE SKELETON CALL INSTRUCTION, W.D. Strecker, 21 May 1975)a simple skeleton call and return instruction were described. It was pointed out in that memo that those instructions did not represent a complete subroutine calling mechanism and that such a mechanism would be defined shortly. This memo documents that mechanism. The complexity and power of the mechanism documented here should be compared against the earlier, simpler call and return instructions.

ISSUES ADDRESSED IN THE CALL AND RETURN MECHANISM

1. In the skeleton mechanism, the argument list had to be generated explicitly by software. The current mechanism has an instruction to generate argument lists.

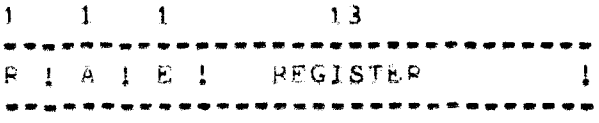
2. In the skeleton mechanism, caller registers were explicitly saved by the caller even if the callee would not destroy those registers. In the current mechanism, registers are saved only if the caller wishes them saved and the callee intends to use them.

3. In the skeleton mechanism, the callee had to explicitly allocate local storage on entry and de-allocate it on return. The current mechanism provides an entry instruction to allocate, and de-allocation is automatic.

THE REGISTER STATUS WORD

In order to implement the call and return mechanism, it is necessary to introduce a 16 bit extension of the PSW called the register status word (RSW). Effectively, the RSW keeps track of registers which have contents which callers have asked to be saved and which callees have not yet saved (or used).

The format of RSW is:



where:

1. R is the register flag.
2. A is the arglst flag.
3. E is the enter flag.
4. REGISTER is a field of bits:
   One for each register R0 - R14(8). [This implies that registers R15(8) and R17(8) are always preserved. R16(8) is never preserved.
   The register conventions are:

   R17 - program counter (PC)
   R16 - stack pointer (SP)
   R15 - argument pointer (AP)
   R14 - local pointer (LP)]

The use of the R, A, and E flags will be explained subsequently.

THE ARGLST INSTRUCTION

ARGLST is an instruction used by the caller to copy a compact, precompiled argument list template into the standard argument list format. ARGLST also allows the caller to save any registers which contain arguments in such a way that the callee's RETURN instruction will restore them.

It is the purpose of ARGLST to handle the passing of formals, static, and local (including stack) arguments. The argument list may be placed in the stack or in static storage. If in static storage, any of the arguments pointers may be pre-compiled into the static argument list and ARGLST will skip over these. Thus ARGLST can be used for FORTRAN as well as other languages. The use of ARGLST is optional for the caller and does not affect the callee.

The form of ARGLST is:

```
ARGLST template, arglist
```

where:

1. Template is a general byte ISP destination specifying an argument list template, and

2. Arglist is a general byte ISP destination specifying the location of the last word of the generated argument list.

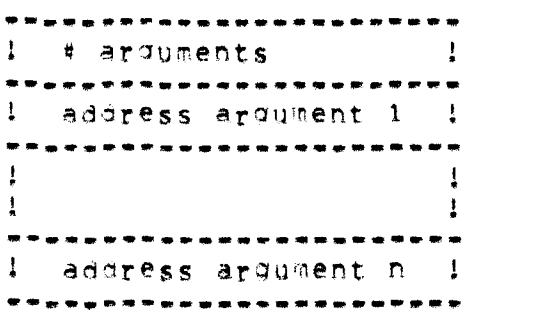
The form of the template is:

1. 16 bit argument in register mask,

2. 16 bit argument count, and

3. general byte ISP source for argument n, ..., general byte ISP source for argument 1.

The form of the argument list created by ARGLST is of the standard form:



Note that the argument specifiers in the template are stored in the opposite direction from the arguments in the argument list. That is because the template is processed as a pseudo I-stream, whereas as the argument list is stored using the stack convention. In order to allow the destination argument list (when in

static storage) to contain pre-compiled addresses. ARGLST interprets a short literal in the template as a no generate.

The size of the template entries are just those of the general byte ISP operand specifiers:

	Argument type	Specifier Size
1.	Local storage	1 byte (sometimes 2 or 5 bytes)
2.	Static storage	5 bytes
3.	Formals	2 bytes (sometimes 5 bytes)
4.	No generate	1 byte

The operation of ARGLST is as follows:

```
        BEGIN
        temp1:=EA (Arglist);
        temp2:=EA (template);
   11:  temp3:=PC;
   12:  PC:=temp2;
   13:  temp4:=(PC)+;
   14:  IF temp4 THEN RSW:=(RSW.OR.100000(8).AND.(.NOT.temp4))
   15:  FOR i:=0 STEP 1 UNTIL 14(8) DO
        IF temp4 <i> THEN -(SP):=Ri;
   16:  IF [temp1 implies SP] THEN RSW:=RSW.OR.040000(8);
   17:  temp 5:=(PC)+;
   18:  FOR i:=1 STEP 1 UNTIL temp5 DO
        BEGIN
        temp6:=EA ((PC)+)
        IF [temp6 does not imply short literal]
        THEN (temp2):=temp6;
        temp2:=temp2-4;
        END;
  19:  (temp2):=temp5;
       IF [temp1 implies SP] THEN SP=temp2
  110:  PC:=temp3;
        END;
```

In the preceeding EA(X) means effective address of X, X<i> means the ith bit of X and (X) means contents of X. Line 11 saves the real PC while line 12 establishes a pseudo PC for argument template processing. Lines 13, 14, and 15 pick up the argument in register mask and, if it is non-zero, sets the R flag in RSW and saves the indicated registers on the stack. Line 16 checks to see if the argument list goes on the stack. If so, it sets the A flag in RSW. Line 17 picks up the count of the number of arguments. Line 18 moves through the argument template and generates the argument list. Line 19 puts the argument count in the argument list while 110 restores the real PC.

THE CALL INSTRUCTION

The form of the call instruction is:

```
    CALL argptr, procedure, mask
```

where:

1. Argptr is a general byte ISP destination which specifies the location of the argument list,

2. Procedure is a byte ISP branch destination which specifies the entry point of the callee, and

3. Mask is a 16 bit literal which specifies the registers of the caller which are to be preserved across the call.

The operation of the CALL instruction is:

```
    BEGIN
    temp1:=EA (argptr);
    temp2:=EA (procedure);
    -(SP):=PSW;
11: PSW:=(PSW.OR.mask). AND. 017777(8)
    -(SP):=AP;
    -(SP):=PC;
    AP:=temp1;
    PC:=temp2;
    END;
```

Note, that in the line labeled 11, the PSW is or'ed with those registers the caller wishes saved and the E flag is cleared. Registers which have already been saved by ARGLST should not be included in mask, since they are already saved.

THE ENTER INSTRUCTION

Sometime after the callee is entered and before the caller alters the contents of any registers (other than PC of course) the callee must execute the ENTER instruction. (Actually, if the callee is not going to modify any registers, or will explicitly save and restore any registers to be modified, the ENTER instruction need not be executed.) The form of the ENTER instruction is:

```
ENTER mask, size
```

Where:

1. Mask is 16 bit literal which specifies, which registers the callee intends to use and,

2. Size is a general byte ISP source specifying the size in bytes of local storage to be allocated.

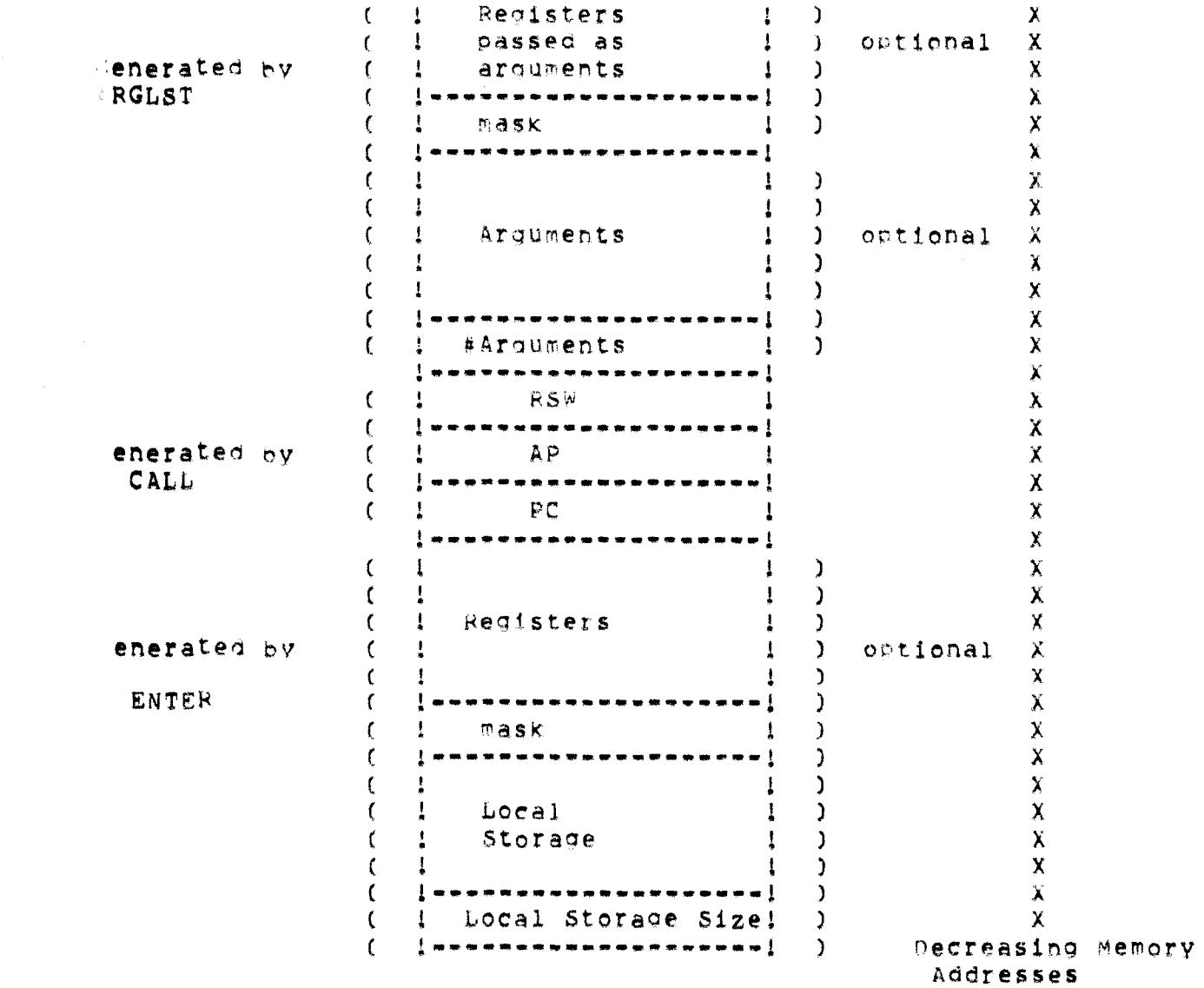
The operation of ENTER is as follows:

```
        BEGIN
   11:  temp1:=PSW.AND.Mask
   12:  PSW:=(PSW.AND.(.NOT.Mask)) .OR.020000(8)

   13:  FOR i:=0 STEP 1 UNTIL 14(8) DO IF temp1<i> THEN
        -(SP):=Ri;
   14:  -(SP):=temp1;
   15:  temp1:=(SIZE + 3)/4;
   16:  SP:=SP-temp1*4;
   17:  AP:=SP;
   18:  -(SP):=temp1;
```

```
END;
```

In line 11, temp1 contains a mask of the registers which previous callers have asked to be preserved and which the callee intends to use. In line 12, RSW retains a mask for the remaining registers which callers have asked to be preserved and the flag is set. In line 13, registers masked by temp1 are saved, while in 14 temp1 is saved. Lines 15, 16, and 17 convert the byte local storage allocation to the nearest 32 bit word allocation and allocate that amount of storage on the stack. In line 18, the size of the local storage allocation is saved.

After the ARGLST, CALL, and ENTER instructions have been executed, the stack frame looks as follows:



THE RETURN INSTRUCTION

The RETURN instruction undoes the entire stack frame just described. The form of the RETURN instruction is:

```
RETURN condition
```

where: condition is a general byte ISP source which specifies

how the four condition codes are to be set on return.

he operation of return is as follows:

```
        BEGIN
        temp3:=condition
   11:  IF RSW.AND.#20000(8) THEN
        BEGIN
        temp1:=(SP)+;
        SP:=SP+4*temp1;
        temp1:=(SP)+;
        FOR i:=14(8) STEP-1 UNTIL 0 DO
        IF temp1<i> THEN
        Ri:=(SP)+;
        END;
   12:  temp2:=(SP)+;
   13:  AP:=(SP)+;
   14:  RSW:=(SP)+;
   15:  IF RSW.AND.#60000(8) THEN
        BEGIN
        temp1:=(SP)+;
        SP:=SP+4*temp1
        END;
   16:  IF RSW.AND.#10000(8) THEN
        BEGIN
        temp1:=(SP)+;
        FOR i:=14(8) STEP -1 UNTIL 0 DO
        IF temp1 <i> THEN
        Ri:=(SP)+;
        END;
   17:  RSW:=RSW.AND.#60000(8);
   18:  condition-codes:=condition;
   19:  PC:=temp2;
        END;
```

ne line 11 checks if the E flag is set and, if so, de-allocates
ocal storage and restores the registers saved by enter.
ne lines labelled 12, 13, and 14 restore RSW, AP, and retrieves
C. The line 15 checks if the A flag is set, and if so, pops
ne argument list from the stack. Line 16 checks if the R
lag is set, and, if so, restores registers containing argu-
ments from the stack. Lines 17, 18, and 19, clear the R and A
lags, set the condition codes, and restore the PC.

D:mjk [6/5/75, pII41.V01]

SUBJ: SKELETON CALL INSTRUCTION

ntroduction

his memo describes what is termed a "skeleton CALL instruction," nd describes how certain instructions and addressing modes efined for the byte ISP relate to the conventions established y the CALL instruction. The reason that it is termed a skeleton CALL instruction" is that it is by no means a omplete subroutine calling mechanism. Such a mechanism will be efined shortly, and as much of that mechanism as possible will e integrated into the CALL instruction.

ssues Addressed in the Skeleton CALL Instruction

alling a subroutine involves the following activities (not ecessarily complete or in order):

. A list of arguments is prepared;

. Relevant state of the caller is preserved;

. Control is transferred to the callee; and

. Local storage specific to the invocation of the callee is defined.

nce the callee is entered, mechanisms must exist for the callee o access:

. Its own local storage; and
. Arguments passed by the caller.

he function of the CALL instruction and related byte ISP nstructions and addressing modes, is to provide fairly fficient and general support for points 1 through 6.

he CALL Instruction

he form of the CALL instruction is:

```
CALL argptr, procedure, mask,
```

here:

. Argptr is a general byte ISP destination which specifies the location of the argument list;

. Procedure is a byte ISP branch destination which specifies the entry point of the callee; and

. Mask is a 16 bit literal which specifies the registers of the caller which are to be preserved.

It is assumed in the following discussion, that the byte ISP as been modified to include 16 "general" registers, denoted R0 - R17. R16 through R17 are analogous to P0 through R7 on he current 11.)

ıe CALL instruction implicitly recognizes three registers R14,
16, and R17. Specifically:

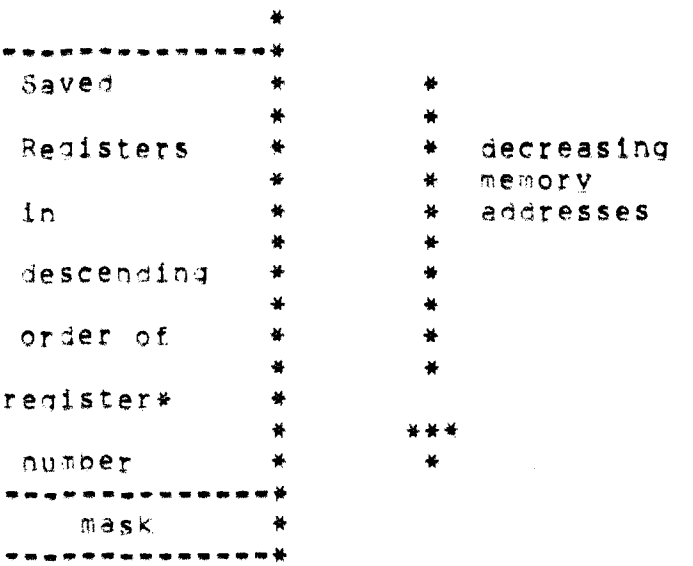
, R17--program counter (PC);
, R16--stack pointer (SP); and
, R15--argument pointer (AP).

ı ALGOL-like notation, the operation of the CALL instruction
ı as follows:

```
   BEGIN

   temp1:=EA (argptr);
   temp2:=EA (procedure);

   FOR i:=17(8) STEP 1 UNTIL 0 DO
   IF mask<i> THEN
   -(SP):=Ri;
   -(SP):=mask;
   AP:=temp1;
   PC:=temp2;

   END;
```

ıere EA (X) means effective address of X and X<i> means the
th bit of X.

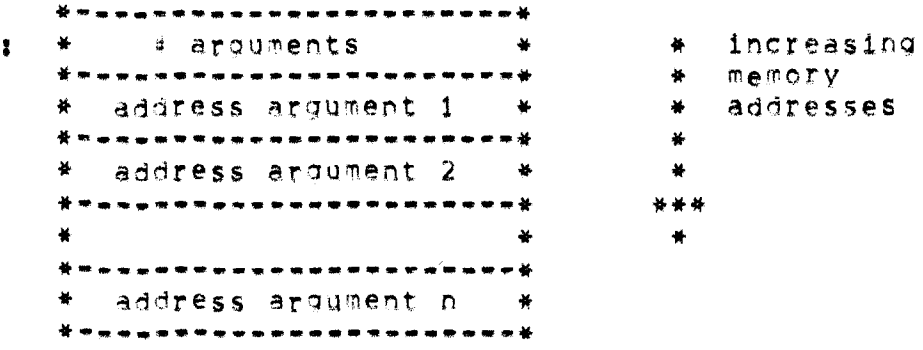
ter the CALL instruction is executed, the stack looks as
ollows:



rgument List

is implicitly assumed in this memo that arguments are passed
reference. The argument list contains the addresses of
guments. (There is nothing inherent in the CALL instruction
ıat precludes passing arguments by other means (i.e., value,
ame, or string); however, both the caller and callee would
ıve to agree on the interpretation of the argument list. This
ıterferes with the goal of general usability of software
odules.)

ie form of the argument list is as follows:



gument lists generally appear in one of three places:

. In a fixed area of memory;
. In line with the CALL instruction; or
. On the stack.

ie CALL instruction supports 1 and 3 directly:

```
CALL A, ...
```

ere A is the location of the argument list, or:

```
CALL (SP), ...
```

ere the argument list is on the stack. Note that the argument ist must be pushed on the stack in reverse order. To support the following is used:

```
   CALL L1, ...
   BR   L2
L1:
   argument list
L2:
```

ie use of either 1, 2, or 3 is transparent to the callee.

eparing the Argument List

en the arguments are in fixed locations in memory, the argument ist can be generated at compile time; otherwise, it must be nerated at the time of the call. A byte ISP instruction, MOVA* ove address), greatly facilitates this. The form of the instruc- ion is:

```
   MOVA source, destination, where:
```

1. source is a general byte ISP source, and
2. destination is a general byte ISP destination.

ctually, because of the context (operand size) dependent addressing odes, the move address instruction exists in four forms:

1. MOVAB

2. MOVAI
3. MOVAL = MOVAF
4. MOVAQ = MOVAD

ıe operation of the instruction is simply:

```
BEGIN

destination:=&A (source)

END
```

ıe MOVA instruction can be used to readily generate argument
ists containing addresses of variables in dynamically allocated
ocal storage and of saved register images on the stack.

ocal Storage Addressing

: is the responsibility of the callee to define its own local
torage. This is typically defined statically (i.e., FORTRAN)
: on the stack (i.e., ALGOL). A single register R14, called the
local pointer" (LP), is, by convention, at the base of local
torage. In the static case, the callee would execute:

```
MOVA    A, LP
```

ıere A is the static base of local storage, while in the stack
ase the callee would compute the amount of local storage
eeded (say, in R0) and execute the following:

```
SUBL    R0, SP
MOVL    SP, LP.
```

ocal variables are addressed as positive offsets of the LP.
ıe byte ISP provides three lengths of operand specifiers
or operands in local storage:

- Xs (LP) -- 2 bytes
- X1 (LP) -- 5 bytes
- Xvs (LP)-- 1 byte

vs is a 6 bit offset specified in the source or destination
yte, which is implicitly shifted left twice, to make it a long
ıteger offset.

rgument Addressing

calar arguments are accessed indirectly through the AP. Thus,
o pick up the ITH argument, the addressing mode:

```
@ 4*I (AP)
```

ould be used. The byte ISP provides two forms of this address
ode:

- @ Xs (AP) 2 bytes
- @ X1 (AP) 4 bytes

orm 1 handles up to thirty-one arguments.

requently, the argument list contains the address of the base
f some data structure (i.e., an array), and it is desirable
o have an efficient way of accessing arguments within that
tructure. Ideally, a logical offset within that data structure
ould be computed, and a single addressing mode would combine
he offset and the base address in the argument list and access
he desired data structure element. However, is it frequently
onvenient to modify (bias) the data structure base as specified
n the argument list, so as to simplify computation of the
ffset. This modified data structure base would typically be
laced in local storage. Assume that the logical offset has
een computed in R* and the biased data structure base is in
ocal storage location Xs relative to LP. Then the addressing
ode:

```
(Xs (LP))[R]**
```

ould access the desired argument data structure. This operand
n the specifier is 2 bytes long.

By logical offset, it is meant that to obtain the iTH long
integer in a data structure, the offset contains i (and not
4*i, as in the current 11).

*Brackets [] denote logical or context indexing.

he Return Instruction
---

he RETURN instruction assumes that the SP points to the
egister mask. The operation of the RETURN instruction is
s follows:

```
BEGIN
temp1:=(SP)+;
FOR i:=1 STEP 1 UNTIL 17(8) DO
IF temp1<i> THEN Ri:=(SP)+;
END;
```

S:mjk [5/21/75, SKELCL.WS3]

UBJ: PENDING ISSUES FOR BYTE ISP

- Exact format of registers and addressing modes.
- Assignment of data-types to address boundries.
- Can we define a range of address sizes in order to reduce program size. (What is efficiency of 11VAX ISP?)
- New operation codes and data-types (string, decimal, vector, list, bit field.
- General operation codes: exchange, block I/O, block I/O with translation, register save, more physical address for I/O
- Uninitialized value traps.

B:mjk [6/4/75, BYTISP.GB11

* * * C O M P A N Y C O N F I D E N T I A L * * *

An Address Mapping Proposal for the Extended-11 Architecture

W. D. Strecker
T. N. Hastings
Digital Equipment Corp.
21 May 1975

Goals and Features

1. Work for smallest (RT systems), medium (RSX) and large (new operating system).

2. Add minimal hardware state information to a process, so that context switch time is minimal.

3. Most shared segments have system-wide virtual addresses which are always present but may be protected, (good for interprocess communication and system-wide shared data bases).

4. Be reliable--no need for share counts or list structures for keeping track of which processes are accessing which pages. Need segment share count only for per-process region sharing.

5. Work well for the classic 2-domain (exec/user mode), 8-segment (KT-11) systems with minimal overhead caused by the unused generality. No risk involved for converting existing systems.

6. Provide 8 access modes with controllable access capabilities per process. Used in the new system to eliminate the monolithic operating system.

7. Be simple, but general. A single mechanism controls segment access, device access, and domain access.

8. Segments are even multiples of the page frame size. Segments can be paged or not.

9. Provide hardware validation of argument addresses passed from lower to higher privileged acess modes.

10. Permit argument addresses to point to segments that have different access mode protections from each other. E.g., access mode 5 calls access mode 2 which calls access mode 1 with addresses in access mode 5 and access mode 2.

11. Inter and intra segment argument referencing be the same. Callee can pickup address of an argument in a register and continue to have address validation occur.

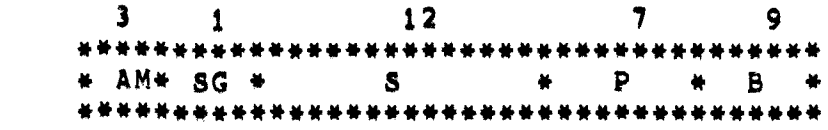
12. Addressing and protection work the same for all instructions, including the new class of instructions which work on address data types.

13. Separate segment number from access mode number, so that two argument pointers can point to the same segment but have different protection.

14. Provide more powerful global sharing and access control than the competition.

Virtual Address

Assuming the byte oriented ISP, the processor generated virtual address is 32 bits. For mapping purposes this address consists of five fields:



where:

1. AM is the access mode,
2. SG is the segment group,
3. S is the segment in SG,
4. P is the page in S, and
5. B is the byte in P.

The three fields S, P, and B are just those found in the classical form of segmented/paged addressing. With such a form of addressing, a central problem is how single copies of procedures and data can be shared by multiple processes. The solution to this problem basically takes one of two forms:

1. Linkage Segment--each shared procedure has a per process linkage segment (or portion thereof) thr which all non local references are made. This allows shared procedures and data bases to be placed arbitrarily in a process virtual address space.

2. Shared Unique Address--shared procedures and data bases have unique virtual addresses which are known to all processes. (This is a special case of the situation where all^H^H^H___ virtual addresses in the system are unique.)

Of these two approaches the shared unique address is the simplest and is the one adopted for this proposal. All system wide shared procedures and data bases are placed in the system^H^H^H^H^H^H_____ segment^H^H^H^H^H^H^H_______ group^H^H^H^H^H_____ and permanently linked togeth From the standpoint of an individual process these shared procedures and data bases have unique virtual addresses and occupy half of the virtual address space. The other half of a process virtual address space is the process^H^H^H^H^H^H^H_______segment^H^H^H consists of addresses which are unique only to that process or are shared by a subset of the processes.

The PR field is 3 bits, so that 7 useful access mode are provided. (We already have uses for 3 useful access mode, kernel, supervisor, and user). Given the discussion above the S, P, and B fields are defined on the basis of arbitrary and pragmatic considerations, and could be changed. The size of the B field defines the page size of a pDp-11 disk block or 512 bytes. Thus the B field is 9 bits. Given the desireability of paging page tables, a page table should fit in a single page. Considering the information needed, a 32 bit page table entry is required. This implies that 128 can fit in page and hence a 7 bit P field. The remaining 12 bits then defines the size of the S field.

Virtual to Physical Mapping^H^H^H^H^H^H^H^H^H^H^H^H^H^H^H^H^H^H^H^H^H^H^H^H^H

Mapping from virtual to physical addresses logically in volves four steps:

1. The contents of the SG field are used to select the system segment table (SST) or the process segment h itable (PST).

2. The contents of the S field are used to select a segment table entry (STE) which describes a page table (PT).

3. The contents of the P field are used to select a page table entry (PTE) which describes a physical page frame.

4. The contents of the B field are concatenated with the page frame to from a physical address.

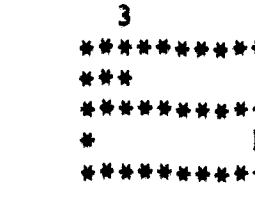
The f the S field are used to select a segment table entry (STE) which describes a page table (PT).

3. The contents of the P field are used to select a page table entry (PTE) which describes a physical page frame.

4. The contents of the B field are concatenated with the page frame to from a physical address.

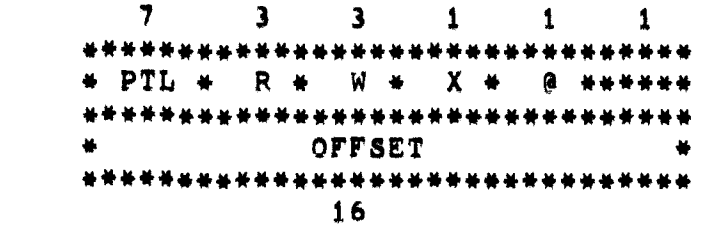
The descriptions and formats of the tables, and table entries actually involved in the mapping follow:

1. There are two hardware registers called the system segment table base (SSTB) and the process segment table base (PSTB). The contents of the PSTB is part of process context and changed on each process context switch. The contents of the SSTB is unchanged on a process context switch. The format of the SSTB and the PSTB is:



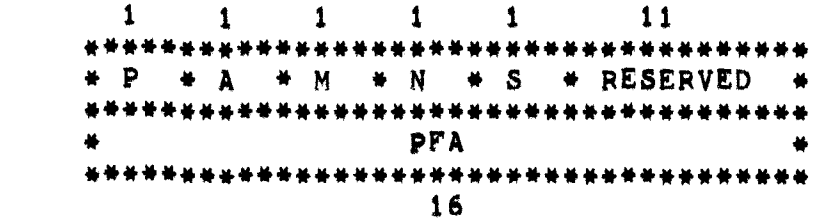
"Hwhere STL is the length of the segment table in pages and PFA the physical page frame address of the segment table. This restricts segment tables to be in the first 2**25 bytes of physical memory.

2. The format of a STE is:



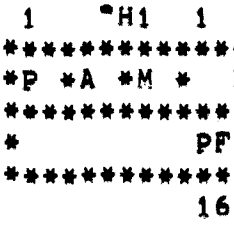
Where PTL is the length of the page table; @, R, W, and protection fields whose use will be defined later; and OFFsET is an offset in a data structure called the page table table (pTT).

3. The reason for introducing the PTT is to facilitate paging of page tables. In order to do this efficiently there must be only a single physical pointer to a page table. This pointer exists in the pTT. The base of the PTT is pointed to by a hardware register termed the PTT base (PTTB). The format of a PTT entry (PTTE) is:



Where P is the present bit (i.e., in memory) and PFA is the physical page frame address of the page table. When P is zero (i.e., not present) the remaining 31 bits is normally used to contain the disk address of the page containing the page table. The N bit is used to indicate that the segment is non-paged. In that case PFA is the page frame address of the beginning of the segment. Non-paged segments are generally used to avoid the spatial overhead of page tables tables for very short segments. They also might be used exclusively in some implementations that do not have paging. For non-paged segments the A bit, when set, indicates that the segment has been accessed, and the M bit when set, indicates that the segment has been modified (written). The S-bit indicates that this segment is a per-process stack segment.

4. The page table entry is similar to a PTTE:



Where P is the present bit, A is the accessed bit, M is the modified bit, and PFA is the page frame address. The A bit when set indicates that the page has been accessed, and the M bit when set indicates that the page has been modified (written). Again when P is zero the remaining 31 bits normally contains the disk address of the page. The reserved field of a PTE (and PTTE) can be used to extend PFA in systems with greater than 2**25 bytes of physical memory.

Protection H H H H H H H H H H

Protection is enforced by eight modes of processor operation. These modes are analogous to the two or three modes of operation defined for current memory mapped -11 processors. In view of how these modes are to be used, access mode is a more suggestive word than mode, and will be used in the discussion. The access mode are numbered 0 to 7 with access mode 0 the most privileged. Since access mode 0 can read, write, or execute all segments in the address space, operating systems would typically use only access modes 1-7. Access Mode 1 corresponds to kernel mode on the current -11. When the effective address has been computed. The Final Effective Access mode (FER) contains the access mode number to be used for the protection check in the appropriate STE.

Each segment table entry defines protection for an individual segment. The protection for read, write, and execute is specified separately. These are the 3-bit R, W, and the 1-bit X fields in the STE. Every time the processor wishes to access memory it presents the memory management unit with a 32-bit virtual address (VA) which includes the PR field. In addition the processor specifies the reference type (R, W, or X). The memory management unit maintains a copy of the highest (least privileged) access mode encountered during an effective address calculation. This register is called Final Effective Access Mode (FEAM). Each time a 32-bit address (including PR field is presented to the memory management unit, it compares PR with FER. If PR is greater than FER, the memory management unit copies PR into FER. The memory management unit uses FER for performing access checking in the SST or PST. For the address mapping to proceed, FER must be less than or equal to the value in the appropriate R, W, or three bit function of the X field in the STE (this function has the value of 0, if X=0, or the R field, if X=1). If it is not, a memory management trap occurs and the reference is aborted.

The current access mode of execution is contained in the PR field of R7. This will be written as R7<PR>. As the effective address is computed, the effective access mode is also computed. The hardware guarantees that the final effective access mode (FER) will be greater than or equal to the current access mode as specified in R7<PR>. Thus FER is initialized to R7<PR> at the beginning of each effective address computation. On the PDP-11 each address generated by the CPU specifies an Rn. A full 32-bit add is performed using the 32-bit contents of Rn including the PR field. The resulting PR field is compared with FER. If PR is greater than FER, PR is copied into FER, thus increasing the access mode number which may lower the privilege. This is repeated, if indirection is specified.

EXAMPLE: MOV Rs,@X(Rd)

Registers used:

VA = 32-bit virtual address register

R0 - R7 where R7 = Program Counter

INST = Instruction register

FER <2-0> = Final Effective Access Mode

T = temporary interval register

MV = Virtual Memory

1. Instruction fetch

```
VA<31-Ø> <--- R7<31-Ø> ; FER<2-Ø> <--- VA<PR> ;

INST <--- Mv(VA) if execute^H^H^H^H^H^H^H_______ ok for segment VA<28-16>
                    from access mode FER ;
```

2. Source fetch

```
T <--- Rs<31-Ø> ;
```

3. Destination fetch

```
VA<31-Ø> <--- R7<31-Ø> ; FER<2-Ø> <--- VA<PR>

VA<31-Ø> <--- X + Rd<31-Ø> (32-bit add)

FER<2-Ø> <--- max(FER, VA<PR>)
```

3.1 Indirect store

```
VA<31-Ø> <--- Mv(VA) if read^H^H^H^H____ ok for segment VA<28-16>

                              from access mode FER.

FER <--- max(FER, VA<PR>)

Mv(VA) <--- T if write^H^H^H^H^H_____ ok for segment VA<28-16>

                     from access mode FER.
```

New instructions^H^H^H^H^H^H^H^H^H^H^H^H^H^H^H^H________________

The Move Address MOVA instruction computes the effective source address like any other instruction. However at the end it replaces source VA<31-29> with FER<2-Ø>, so that the address moved is validated (i.e., access mode field is equal to or greater than R7<PR>). Actually four forms of MOVA are required: B, W, D, and Q so that the index register can be appropriately shifted.

A single CompareAddress instruction CMPA compares the contents of the source word <29-Ø> with the contents of the destination word <29-Ø>. The PR field is not included. The 29-bit quantities are sign extended to set the condition codes.

The CALL instruction sets PR field of the argument pointer to the access mode of the caller. Thus the caller is garranteed to reference the caller's arguments at the caller's access mode level (or higher). The CALL instruction also ensures that the stack pointer is pointing to a stack segment as specified by the S bit in the PTTE. A return instruction returns from a CALL.

An ENTRY instruction allocates stack storage and makes sure that SP remains in as STACK segment.

Stacks

Only one stack is needed per process. It will have its S bit set and will have the access mode fields set to 7. Asynchronous software interrupts (ASI) will use a new stack as set up by the software in a different segment. Examples are Control C interrupt, EOF, etc. To protect the interrupted stack the software will raise its protection to access mode 0 until the ASI is dismissed. Synchronous hardware faults (SHF) will use a separate system-wide stack (as on current -11s). Examples are page fault, floating overflow, stack invalid, etc. Such fault routines will run to completion in a short time or will copy the process status onto the process stack.

Indirect Segments

Because protection is applied at the segment level, the smallest protectable entity is the smallest segment, or a page. It is very often useful to protect an entity smaller than a page: for example, a unibus address or an executive procedure entry point. To do this in the obvious way would require a special segment whose length could be specified explicitly down to the byte level. This is a reasonable solution but requires new table entry formats and mapping procedures. It also conflicts with the goal that in a paged system all memory management is done at the page level. This address mapping proposal contains an alternative solution.

A fourth protection field, the indirect field (@) is specified in an STE. When the processor generates an indirect reference through a virtual address in a segment whose STE has the @ bit set, a special form of protection check is performed. The processor is assumed to specify the ultimate use of the indirect address (i.e., R, W, or X). The R7<PR> field is checked against the STE protection based on the ultimate use of the indirect address. A segment with the @ bit set may not be directly read, written, or executed (except in access

To illustrate the use of this assume that it is desired to give a process running at access mode 4 access to a single unibus address. A virtual page in a segment with R=W=X=@=Ø protection would be created mapping into the unibus address space. say address A in this page corresponds to the desired unibus address. Another virtual page (the indirect page) in a segment with @=1, R=W=4, X=Ø protection would be created. Say the contents of address B in the page contains #A (the address of A). Then a processor running in access mode 4 could reference the byte at the unibus address by:

MOVB (B), destination.

The indirect segment also allows protected entry to lower access mode procedures. This is effected through the CALL instruction. The format of the instruction includes a destination. To transfer to a destination with lower access mode protection the destination must be indirect through an indirect segment (if it is not, an illegal instruction trap results.) R7<PR> is set to the lower of R7<PR> and the R field of the indirect segment.* The RETURN instruction restores R7<31-Ø> from the stack. However R7<PR> is not set lower than R7<PR> at the time of execution of RETURN.

Argument passing across access modes is essentially similar to an ordinary procedure call. Generally the calling procedure provides a list of argument addresses to the called procedure in some way defined by convention**. The PR field of the argument pointer forces the access to arguements to occur at the caller's access mode level.

Compatability Mode

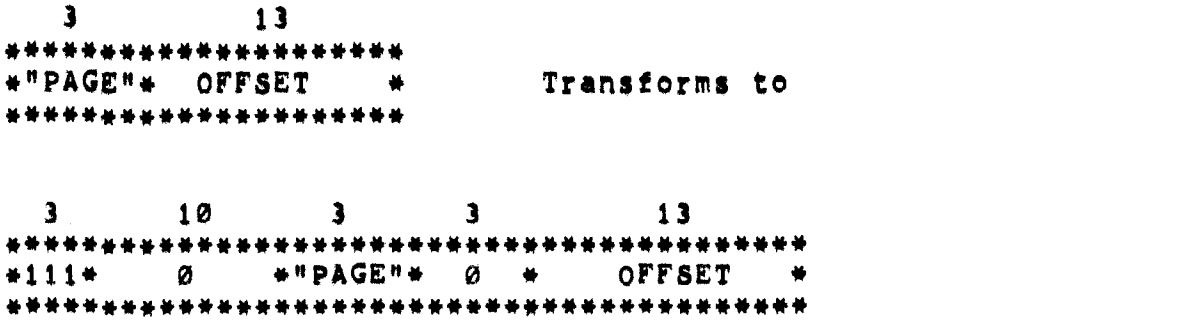
When operating in compatability mode, the address mapping must simulate the behavior of the KT-11. This effectively requires eight variable length "pages" on 8K byte virtual boundaries with individually specifed protection. Since the address mapping scheme described above does not provide protection below the 64K byte segment level, there is no contiguous portion of the virtual address space which has the necessary attributes to simulate the KT-11.

The solution to this is to properly specify the transformation of a 16 bit virtual address generated in compatability mode to a

---

* Raising R7<PR> by a CALL through on indirect segment is not permitted.
**The conventions are built into the CALL instruction.

32 bit virtual address and then to map that address normally. The transformation is:



The KT-11 "page" boundaries are now on segment boundaries where protection and length (in 512 byte increments) can be specified arbitrarily.

Loose Ends and Problems

1. Problem - the segment size should be the same for all processes.

   Solution - move PTL field to PTT.

2. Do we need a 3 bit X field?

3. Do we need restrictions as to which access mode can enter a protected entry point?

4. Problem--need a way to prevent a CPU from accessing a segment and/or page while it is being swapped in. Solution - add a light in PTTE and PTE which says page in transit.

5. Problem - it is highly desirable to have argument lists distinguish between input and output arguments.

   Solution - add another bit to access mode field and by convention access modes occur in pairs. The higher privileged access mode in each pair can write.

6. Problem - IO cannot go into stack segments.

7. Problem - callees may change access mode and/or SG field during address computation.

8. Indirect segments seem to be powerful. There are two proposals. One in this document and one in the following one (some extensions and enhancements to the indirect segment concept).

Rejected Ideas

1. A full capabilities scheme was considered. It was rejected

because of the uncertainty in a new system (despite its straight forward use in RT and RSX-like systems). Also the inter domain call mechanism appeared to be very complex. See Address Translation, T. Hastings/D. Nelson, 5 May 75.

2. We considered making access mode 0 be lowest privilege and 7 be highest so that it works the same as interrupt levels. This was rejected because an argument list with 0 access mode # seems likely where the caller does not intend to lower the privilege below his access mode level.

3. We considered having I/O use the mapping mechanism. It was rejected because of the high hardware complexity and low software gain.

WS:cw:mjk [6/5/75, WSP0B1]

SUBJ: PROCESS STRUCTURE

There are two very different process structure proposals. They are:

1. Process structure — Dave Nelson
2. A Higher-level Interrupt Proposal — Rich Lary

1 is very general and 2 is an extension of current hardware interrupt mechanisms. Perhaps both will be included. However, there is the hope that the overhead of 1 can be reduced sufficiently to be practical for handling all I/O.

We must also address:

1. Message transmission among processers.

2. Other inter-process communications.

3. Overhead (performance in time,space) and computer-to-computer links.

4. Multi-processors interrupts, I/O connection including Cio's.

GB:mjk [6/4/75, PROSTR,GB1]

Process Structure

Dave Nelson
June 4, 1975

The notion of a process, and with it the mechanics of an interprocess communication, are fundamental concepts found in modern operating system designs. The goal of process structuring is to define a set of primitive data structures and operators related to processes, which are sufficiently powerful to make their inclusion into the hardware worthwhile, and yet sufficiently general that they will be useful over the range of software systems to be implemented on the machine architecture.

A process is the logical progression, of a program; it is the forward motion, the advancement from one logical state to another. In computer systems, pro- cesses are realized by data structures which include programs and process context. By context, we mean the dynamic state of the process, generally referring to machine registers and other data required to distinguish processes.

Motivation

Early computer systems were characterized by a single processor executing a single program, which explicitly defined the entire functionality of the system. Input and output instructions were imbedded in-line to the program causing extensive delays when synchronized to slow devices. The advent of interrupt hardware invoked a degree of parallelism in that programs associated with I/O functions could be separated from mainline programs, and run asynchronous to the mainline program, but synchronous to the peripheral (under interrupt control), thereby making optimal use of processor resources. Some elementary logic was required to synchronize the interrupt routines with mainline programs, particularly when exceptional events occurred such as the reception of a carriage return from a terminal. But for the most part these mechanism were ad hoc and informal.

Modern concepts of a system architecture considers, in retrospect, the interrupt driven I/O functions as separate processes which cooperate with the mainline process. To some extend, process structured architecture is a natural extension of interrupt structures.

Quite apart from I/O functions, operating systems have evolved from a simple single user control program to more complex multiprogrammed systems. More recent operating

systems have formalized the notion of independent but cooperating programs each of which shares processing resources in accordance with some criteria such as priority. Programs are perceived to execute under control of a supervisory control program which is designed to provide an illusion of independent, parallel processors with the cost of only a single processor which is controlled to multiplex itself across all runnable program.

Following these lines of evolution, process structured architecture now acquires a form that:

1. recognizes I/O functions as first class processes which periodically have to wait for slower peripherals, and

2. exploits the separation of programs from processes, hithertofor done to software, to its logical generality which allows a rather arbitrary relationship between processes (programs) and processors (hardware).

The essential characteristic of a process, therefore, is that it conceptually has a processor all of its own, and that the state of its processor is more or less independent of all others. The physical allocation of processors is unimportant, and since there are usually more processes than there are processors, the system architecture must be designed to multiplex a fixed number of physical processors, thereby creating an illusion of a larger number of conceptual processors, each assigned to a process. The multiplexing of physical processors (scheduling) is a major portion of this section.

Occasionally, processes are required to interact to an extent predicated by their function. Since all processes are conceptualized owning their own processor, and that the processors run asynchronously, there can be no assumptions made as to the time progression of individual processes. Time ordered relationships must be handled explicitly by means of hardware functions. These hardware functions which allow independent processes to cooperate and synchronize are also a major portion of this section.

Process States

In general, a process can be in any of four states:

1. Running--process is running on a processor.

2. Runnable--process is runnable and waiting for a processor.

3. Waiting--process is waiting for an event.

4. Disabled--process is in a state unknown to the hardware environment.

As system execution proceeds highest priority processes found in the runnable state are moved into the running state when a processor is assigned to its execution. Processes continue execution until either a higher priority process becomes runnable, or the current process enters a waiting state. The logical structure which maintains process states consist of a series of process control blocks linked together in three distinct queues (running, runnable, and waiting). Processes which are not contained in either of these queues are assumed to be disabled and under the management of higher level software.

Process Structure Mechanics

The mechanics of process structured architecture can best described in terms of (1) its data structures, and (2) its operations. The data structures are comprised of three fundamental entities: first, Process Control Block (PCB) contains the entire context for a process; second, a structure of queues, or link list, define an ordered set of PCB's which describe the state of the process being either running, runnable, or waiting; third, a data structure called a semaphore is used to provide interprocessor synchronization and control. The operations which characterize the process structure mechanics are: first, functions of the CPU which invoke process scheduling; second, primitives which operate on semaphores for process synchronization; third, interrupts and interprocessor communication.

Process Control Block

The process Control Block contains the following general information:

LINK
PRIORITY
PROCESS STATUS WORD
CONTEXT/RESOURCE MASK
MAP POINTER
GENERAL REGISTERS

Process State Queues

Process state queues consists of process control blocks linked together in some fashion so as the described the state of all the processes in the system. All three queues (running, runnable, waiting) are managed by equivalent logic using the LINK portion of the process control block field. Process states are characterized by which queue contains them. When a process changes its state, it is remove from its current queue and entered into the queue describing its new state. The following is a brief description of each que.

The running queue contains all PCBs which are currently executiong (on a single processor system, this queue trivially reduces to a pointer to the current PCB). The order of PCBs within the running queue is by decreasing priority, the lowest priority process appearing first in the list. The list is equal in length to the number of CPUs currently operating.

The runnable queue contains a table of pointers, one for each priority level. Each pointer contains the address of the last PCB of the next higher priority level queue. Further, all priority level queues are also linked together to form one single runnable queue. In this way, processes which become runnable having a specific priority can be linked into the runnable queue by their priority, in an efficient manner. The following describes the runnable queue:

LINK TO PRIORITY ZERO OR HIGHEST RUNNABLE
LINK TO PRIORITY ONE OR HIGHEST RUNNABLE
LINK TO PRIORITY TWO OR HIGHEST RUNNABLE
etcetera

Processes which are neither running nor runnable are contained in waiting queues. Waiting queues are located by entries in semaphore data structures, described below. Events which occur on these semaphores invoke dequeuing from the waiting que and an enqueing into the runnable queue at the processes priority.

Semaphores

Semaphores are data structures used to provide interprocess communication and sychronization. The format of the semaphore is as follows.

COUNT
LINK TO FIRST PCB
LINK TO LAST PCB
CURRENT OWNER
MAXIMUM LIMIT

The count is an indication of the number of processes currently linked to the waiting queue. The first and last link fields point to the beginning and end of the que respectively. The owner field points the PCB which was at the head of the queue prior to the most recent signal. The limit field contains the maximum value of COUNT.

Semaphores are used to generally synchronize and control inter-task activities. They can be used to lock out critical sections (test and set), they can be used to provide time ordered sequences of processing among groups of cooperating processes. The pimitives which operate on the semaphores are described below.

Process Structures Operations

The following describes the functional operations which act upon the previously described data structures. These include priority scheduling, interprocessor comunication, interprocess communication, and peripheral interrupts.

Priority Scheduling

As previously described, the process structure system is characterized by a set of independent processes each of which is logically perceived to proceed independently and in parallel. Processes, being logical artifacts, require processors for their execution. The relationship of which processors are executing on which processes is controlled by the priority scheduling operation.

Priority scheduling is invoked whenever:

1. the currently running process executes a wait primitive (described below), or

2. an external signal is received from

    a. another CPU,
    b. a peripheral controller.

In the case of (1), the processor links the currently running process to an appropriate WAIT que, and in the case of (2), the processor places the current PCB at the top of the runnable queue and dequeues the PCB from the appropriate wait queue if the external signal was generated from a pripheral controller (i.e., an interrupt), placing this PCB on the runnable queue. At this point, the processor reschedules itself by selecting the highest priority runnable process, comparing the resouce mask of each PCB against the processor's own resources.

Interprocessor Communications

In multiprocessor systm, a hardware mechanism must be provided to enable one CPU to interrupt another. As processes change state (i.e., enter WAIT state), or when peripheral controllers generate interrupts, it becomes necessary for processors to reschedule themselves depending on their current priority in relation to the set of runnable processes such that, or an n processor system, the n highest piority processes are executing at any given time. In order to guarantee this condition, the interprocessor signalling mechanism must be invoked according to the following algorithm. Whenever a processor reschedules intself by selecting the highest priority PCB from the runnable queue, it determines the priority of the next runnable PCB and

compares it against the priority of the first PCB in the running queue: if greater, an interprocessor signal is generated to the CPU currently running the PCB (note that the running queue is ordered by reverse priority, so that the first entry in the running queue is the lowest running process). Any processor receiving such a signal will reschedule itself performing the above function as well, thereby propagating interprocessor signals until the condition that all processors are running the highest priority processes is achieved.

Interprocess Communication

In a process structure system, each process logically proceeds in parallel. The relative speed of progress of the collection of processes is undetermined and the programmer must provided explicit synchronization mechanisms. These synchronization mechanisms must include the capability of waiting for events, and functions which allow processes to be created and destroyed.

The WAIT instruction examines the state of a semaphore and determines whether or not the process can proceed. WAITs are logically paired with SIGNALs in that in order for a WAIT to proceed a corresponding SIGNAL must have preceeded it in time. The relative excess of WAITs compared to SIGNALs is maintained in the COUNT field of the semaphore data structure. Accordingly, the operation of the WAIT instruction checks to see if the COUNT is greater than zero, and if so it decrements it and proceeds. If the count is less than or equal to zero, the current PCB is enqueued into the WAIT queue for this semaphore and a process schedule is invoked.

The SIGNAL instruction also operates on the semaphore and is the counterpart of the WAIT instruction. The SIGNAL instruction functionally opens the semaphore to allow one and only one WAITING process to proceed. If no process is currently waiting then the next process which performs a WAIT on the semaphore will proceed directly. The operation of the SIGNAL instruction is to simply increment the value of COUNT.

Semaphores must be properly initialized in order for them to function correctly. The equivalent of a CLEAR instruction can can be implemented by standard -11 instructions which initialize the semaphore data structure.

Additional instructions are provided to create and destroy processes. The arguement of the CREATE operator is the address of its PCB. The DESTROY instruction effectively extinguishes the current process.

Interrupts

Interrupts are effected in a manner similar to the operation of the SIGNAL instruction. The results of an interrupt transaction provides an address (trap address) which locates the contents of a process status word. The process status (PS) is encoded so that 11/45 interrupts will run in process zero, unaltered. Additional bits in the PS are used to define process structured interrupts, in which case the low order 16 bits (PC) are used as an address of a semaphore. Consequently, a process structured interrupt is simply the execution of a SIGNAL operation on a semaphore located by a word in the trap address space of the device. Following the operation of the "SIGNAL" operation the processor then dispatches an inter-CPU SIGNAL as previously described.

A Higher-Level Interrupt Proposal

I. Philosophy

This is not a "process-structure" proposal; it is a proposal for a slightly higher-level interrupt mechanism than exists on the PDP-11 today. Its goals are to provide "interrupts" which satisfy the great majority of the synchronization needs of an operating system without adversely affecting device service latencies, as well as "classic" PDP-11 style interrupts. It cannot be used for the hardware switching of user tasks in an RSX-like operating system; however none of the more sophisticated process-structure proposals claimed to do that either.

The operation of this scheme can be conceived of as a multi-priority version of the RSX-11M Fork Queue with selective enable/disable on each priority level to allow synchronization of data access among processes at different priority levels.

II. General Description

A System Process is a process which performs a function on some subset of the system data base and external world, without ever entering an internal wait state. The only event which can interrupt execution of a System Process is the invocation (synchronous or asynchronus) of a higher-priority System Process. All System Processes run in a common address space (namely that part of the address space described by the Shared Segment Table - see Memory Management document) and share a common stack. Their Process Control Block is exceedingly short (16 bytes) and mostly read-only and gives the initial state of the process only - process context is saved on the common stack in the case of preemption. For generality I will describe the scheme as working for 2**N priority levels where N is arbitrary - in practice N could be an implementation dependent parameter but would most likely be 3,4, or 5 (yielding 8, 16 or 32 priority levels). there are no limits on the number of processes, and indeed one of the virtues of this scheme is that storage requirements grow fairly slowly as you increase the number of processes in the scheme.

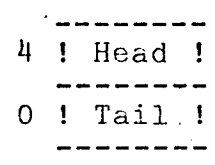
III. Hardware

The following registers are used by this scheme in addition to the PS ar 16 GP registers. Some of them (SCM,PMAX) are strictly performance-enhancing and could be eliminated, but it would be unwise to do so.

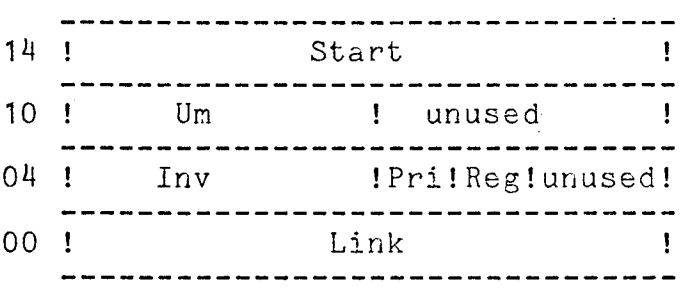
Register Name	Organization	Purpose
System Context Mask (SCM)	16 bits per register set	Used to intellegently save/restore registers on context switches in single or multiple register set environments.
Priority Level (PRI)	N bits	Priority of current software process. Levels 0 thru 2**N-1 are the system priority levels in ascending order. Levels 2**N-5 through 2**N-2 are associated with BR levels 4 thru 7.
Priority Mask (PM) and Priority Enable Mask (PEM)	2**N bits 2**N bits	PM is the mask of priority levels with non-empty run queues. PEM is always ANDed with PM for scheduling purposes.
PMAX	N bits	The number of the highest-numbered bit on in PM .AND. PEM. Automatically updated by the hardware whenever either register is altered.
Priority Run Queue Base Pointer (PRQBP)	32 bits	Points at the base of the array of run queues.
System Stack (SS)	32 bits	Hardware stack used to stack process context and PS,PC pairs on old-style interrupts. Similiar to the Kernel Stack on the 11/40.
Current PCB Pointer (CPP)	32 bits	Points to the Process Control Block of the currently running System Process.
Mode (MODE)	1 bit	This bit is part of the processor Status (PS) word and indicates whether SS or US (User Stack) should be used as SP. The primitives described in section V cannot be executed unless MODE=0.

IV. Control Blocks (in memory) manipulated by the hardware

The Priority Run Queue Table (PRQT) is an array of 2**N listheads of the form:



Which are either null (Head=0, Tail points to Head) or point at a list of Process Control Blocks (PCBs), which have the following format:



Where:

Link Pointer to next PCB in this run queue.

Inv 16 bits of invocation count for this process - initially -1 (dormant), it is incremented for every REQUEST to this process and decremented on every TERMINATE.

Um Use mask - 16 bits of "which registers are used by this process:.

Pri N bit priority of this process

Reg Register set used by this process

Start Initial PC of this process, including Current Ring in high 3 bits. Initial Local Pointer value for this process is #Start+4; i.e. a pointer to the first word after the PCB.

V. Operation

There are 6 primitives associated with this scheme. they are:

REQUEST(PCB) Increments the Invocation Count of the designated PCB. If the count goes from -1 to 0, the PCB is placed on a run queue according to Pri(PCB) and a context switch is performed if Pri(PCB) is greater than PRI. REQUEST can be executed as a machine instruction or by an interrupt vector of a special type.

TERMINATE(CT) Decrements the Invocation Count of the current process's PCB by CT. If the Invocation Count goes negative the process is terminated, and the previous processes context is popped from the stack and it is resumed (unless PMAX is greater than it's priority). If the Invocation count does not go negative the current process is restarted at it's starting address.

DISABLE(MASK) Bit Clears the Priority Enable Mask from MASK. In a multiprocessor system this will cause any other processor which is executing at a level which is now disabled to enter a subinstruction wait (spin) state until the disable is lifted.

ENABLE(MASK) OR's MASK into the Priority Enable Mask. If the new PMAX is greater than the current processes priority, a context switch is performed.

TRAP(NEWPS,NEWPC) Performs a classic PDP-11 style interrupt. SCM is set to all ones for the current register set, and PRI is set to the analogue of the priority field of the PS. TRAP can be executed synchronously (as the result of an instruction fault or TRAP instruction) or by an interrupt vector.

RTI() Returns from a TRAP type interrupt.

VI. "Formal" Description of primitives

The description given below is for the simplified case of one register set, however multiple register sets are a fairly small addition. There is one glaring inefficiency (in addition to the ubiquitous bugs)- namely TERMINATE sometimes pops the previous process's context from the stack only to find that there is a pending process of higher priority (i.e. PMAX > PRI), so it re-pushes everything. This was necessary because the saved registers (determined by (old SCM) .AND. Um(new PCB)) might be different. There is a way around this which is fairly efficient but I didn't have the heart to include it on this go-around.

```
REQUEST(PCB):

    Inv(PCB) := Inv(PCB) + 1
    If Inv(PCB) = 0 then
          Temp := Pri(PCB)
          If Temp > PRI then
                Switchto(PCB)
          else
                Link(Tail(PRQT(Temp))) := PCB
                Tail(PRQT(Temp)) := PCB
                Link(PCB) := 0
                PM(Temp) := 1
          end
    end
end

Switchto(PCB):

    PSTEMP := PS
    If MODE = 1 Then
          MODE := 0    ;switch stacks
          SCM := 177777
    end
    Temp := SCM .AND. Um(PCB)
    -(SP) := PSTEMP
    -(SP) := PC
    -(SP) := LP
    For i := 14 step -1 until 0 do
          If bit(i) on in Temp then
                -(SP) := Ri
          end
    end
    -(SP) := SCM
    -(SP) := Temp
    -(SP) := PRI
    SCM := SCM .OR. Um(PCB)
    PRI := Pri(PCB)
    LP := #Start(PCB) + 4
    PC := Start(PCB)
end

DISABLE(MASK):

    PEM := PEM .AND. (.NOT. MASK)
end

ENABLE(MASK):

    PEM := PEM .OR. MASK
    Checkpri()
end
```

```
TERMINATE(CT):

    Inv(CPP) := Inv(CPP) - CT
    If Inv(CPP) < 0 then
          PRI := (SP)+
          Temp := (SP)+
          SCM := (SP)+
          For i := 0 until 14 do
                If bit(i) on in Temp then
                      Ri := (SP)+
                end
          end
          LP := (SP)+
          PC := (SP)+
          PS := (SP)+ ;maybe switch stacks
          Checkpri()
    end
end

Checkpri():

    If PMAX > PRI then
          Temp := Head(PRQB(PMAX))
          Head(PRQB(PMAX)) := Link(Temp)
          If Head(PRQB(PMAX)) = 0 then
                Tail(PRQB(PMAX)) := #Head(PRQB(PMAX))
                PM(PMAX) := 0
          end
          Switchto(PMAX)
    end
end

TRAP(NEWPC,NEWPS):

    PSTEMP := PS
    PS := NEWPS ;maybe switch stacks
    -(SP) := PSTEMP
    -(SP) := PRI
    -(SP) := SCM
    SCM := 177777
    -(SP) := PC
    PRI := Pri(PS) + (2**N-8)
    PC := NEWPC
end

RTI():

    PC := (SP)+
    SCM := (SP)+
    PRI := (SP)+
    PS := (SP)+ ;maybe switch stacks
    Checkpri()
end
```

II. Enhancements

This scheme is capable of being extended to multiprocessor configurations. The registers PM, PEM, PMAX and PRQBP would exist in a central spot and could only be modified by a central arbitrator. In addition there would be a new register of 2**N bits called the Active Priority Mask (APM) which had bits set for every priority level on which some processor had a process initiated. PMAX would then be redefined as the highest numbered bit set in PM .AND. PEM .AND. (.NOT. APM). This would insure that no two processors could attempt to initiate processes on the same priority level. The primitive operations would have to lock the memory data structures they use (namely the Priority Run Queue array and the Invocation Counts and Links in PCBs) while they were executing.

This scheme cannot be directly extended to handle more complex processes (i.e. processes with internal wait states) but with the addition of instructions to load and save the processor context it is possible to provide such a process-structured environment in software at a time cost considerably smaller than current architectures.

VIII. Simplifications

There are a few ways in which this scheme can be made simpler (read cheaper) at some small expense in performance/flexibility.

The CPP register can be eliminated, since R15 is left pointing at the end of the PCB; processes would be constrained not to alter R15, or at least to restore it to it's original value before executing a TERMINATE.

2) The System Context Mask could be eliminated as well as it's associated logic. The purpose of SCM was to insure that no register ever was saved unless the previous process needed it and the new process was going to destroy it. Simply stacking the set of registers which the new process was going to destroy would be less logic at the penalty of somewhat more memory references per context switch.

3) The Priority Run Queues are currently FIFO. Since there are a large number of priority levels, it would not affect performance very much if they were LIFO. This would save one word per queue in memory, plus some logic and memory references in the insertion and removal of processes.

INTERRUPTS AND PROCESS STRUCTURING

6/16/75

T. N. Hastings
W. D. Strecker

1. INTRODUCTION

Contemporary operating system philosophy tends to formalize the notion of a group of user and executive computational entities as a set of processes. These processes synchronize their execution, control access to shared resources, and communicate with semaphores and messages. The goal of process structuring is to define a set of primitive data structures and operations related to processes, which are sufficiently powerful to make their inclusion in hardware worthwhile, and yet sufficiently general that they would be useful over the range of software systems to be implemented on the extended -11 architecture. The process structuring defined here is intended to be useful in real time, transaction processing and time sharing systems.

This paper represents the synthesis of four preceding papers ("Process Structuring", W. D. Strecker, 4/28/75; "Process Structure", Tony Lauck, 4/30/75; "A Higher Level Interrupt Proposal", Richard Lary, 5/23/75; "Process Structure", Dave Nelson, 5/20/75), and thereby supersedes all of them. It satisfies the different software approaches of procedure-based versus process-based operating system design. The scheme provides a single uniform mechanism which is good for both approaches without compromising either. These ideas were put together by a special task group including: P. Christy, T. Lauck, D. Rodgers, D. Cutler, S. Rothman, R. Lary, D. Nelson and the authors.

2. SUMMARY

This summary presents a short description of the major parts of the process structure, including the handling of traps and interrupts. The purpose of the summary is to give a quick survey of the forest before the reader is led through the trees in subsequent sections.

The two major goals of the process structure are (1) to provide hardware process synchronization and priority micro-scheduling for a wide range of systems from small real time to large transaction processing and time shared systems employing procedure-based and/or process-based software approaches and (2) to make single- and multi-processing hardware and software as similar as possible. The main

distinction between procedure-based and process-based software approaches occurs on the handling of user requests to the operating system. In a procedure-based system user requests are ordinary procedure calls which run as part of the user process in a higher Access Mode. These are synchronous requests since the user process waits if the operating system procedure waits. In a process-based system a request to the operating system sends a message to operating system processes called a system services. These are asynchronous requests since the user process can run concurrently with the system service.

Each CPU is always attempting to run a current process as described by a Process Control Block (PCB) at the lowest priority interrupt level, called BR 0. A CPU register contains the address of a PCB. The PCB contains the process priority level (0-15), CPU context mask, CPU context, address of semaphore if waiting or run queue if runnable, accumulated CPU time used, link to next PCB in a queue, pointer to process map, etc.

External interrupts may temporarily interrupt the process to run short Interrupt Service Routines (ISR's) at interrupt priority levels BR 1-7. The ISR quickly dismisses its interrupt and returns to the interrupted process. Thus a CPU is either at process level (BR 0) or interrupt level (BR 1-7). The hardware automatically uses a per-CPU stack-pointer register for BR levels 1-7. However, BR 0 level general registers must be saved and restored in order to switch processes. In order to speed process context switches (for system services), the PCB contains a context mask indicating which of the 16 registers the process intends to use.

Process level is used for user tasks and for system services. Operating system calls may run as part of the user process (procedure-based operating system) or as a separate process, called a system service (process-based operating system). In addition, ISR's may activate other system services to perform more lengthy computations than those at ISR level. In fact we expect less and less computation at ISR level and more and more at service level. The ideal ISR consists of a single SIGNAL followed by RTI. This type of ISR will be specified by a new form of interrupt vector. RSX-11M has implemented a single level service process mechanism (called fork level) at a cost of 150 micro-seconds per process schedule. Multiple levels would have taken 200 micro-seconds. The VAX hardware provides 16 process priority levels (all at interrupt BR level 0). In addition, the ISP provides primitives to synchronize processes using semaphores. A semaphore is an address of a Semaphore Control Block (SCB) in memory. The SCB contains information about the state of the semaphore, such as number of waiting processes, the address of the PCB which owns or last owned the semaphore, and a pointer to a

queue of waiting PCB's.

Each process can be in one of four states:

1. Waiting (for a semaphore)

2. Runnable (but not running on a CPU)

3. Running (on a CPU)

4. Deactivited

The hardware maintains 16 runnable queues of PCB's, one queue for each process priority level.

The 8 BR levels are in order of decreasing priority:

7	Alert, such as power fail
6-2	I/O devices
1	Inter CPU interrupts, context switching and process selection
0	Processes (user and service)

In addition each CPU has a mask register for enabling and disabling each of the 7 BR levels 1-7. Devices will interrupt all CPU's which are enabled for the level until some CPU clears the request flag in the device. It is anticipated that the software in multi-CPU systems will enable disJoint levels (except BR 7 and 1).

In order to provide a cheap, channel-like capability, devices will be able to interrupt the CPU and execute a single instruction in a vector. This is called Single Instruction Break. Typically, this will happen at a higher BR level until a count runs out. Then a lower BR level interrupt will be requested.

Traps occur in response to (synchronous) events in the CPU caused by the execution of the current instruction, such as floating point trap or memory parity error. It happens immediately, no matter what the BR level is. The BR level remains unchanged. Since the stack may be in error, the hardware uses the per-CPU stack pointer registers, as on current -11's instead of the process stack pointer. Traps at BR level higher than 0 are unusual, but will nest correctly using the per-CPU stack. Trap routines, like ISR's, cannot WAIT. The trap routine must run to completion quickly or move the contents of system stack to the process stack or SIGNAL another process.

The WAIT instruction allows a process (but neither an ISR nor a trap routine) to WAIT for a semaphore (or seize it if it is free). If the process must wait, WAIT moves the process to the end of the semaphore queue and interrupts the CPU at BR level 1. The ISR at BR level 1 consists of three

instructions: save old process, select new process, and restore new process. To allow fast pre-emption of one process by another, the BR level is dropped to 0 for the restore. The SIGNAL instruction allows a process or an ISR to wake up the first process in a semaphore wait queue. SIGNAL requests a BR level 1 interrupt on the CPU running the lowest priority process, if the awoken process is of higher priority. The interrupted CPU executes the same 3-instruction ISR as above. The DEACTIVATE instruction allows a process or an ISR to deactivate a process (itself or another) by removing it from whichever queue it is in. The REQUEST instruction allows a process to test a semaphore and own it if it is free, but not wait for it if it's already owned. The ACTIVATE instruction places a PCB in the appropriate run or wait queue. The CHANGE-PRIO instruction changes the priority of a process.

In order to implement process synchronization primitives, there must be a more fundamental mechanism for synchronizing CPU's. The memory system provides four primitives: read, write, read-lock, write-unlock. Most controller and CPU accesses are ordinary (possibly cached) reads and writes. The process synchronization primitives generate sequences of (uncached) accesses which begin with a read-lock and terminate with a write-unlock. During this time read-lock accesses from any controller or CPU are negatively acknowledged. These controllers CPUs will loop making read-lock requests until one is accepted or an interrupt occurs. While a read-lock is in force for one CPU, ordinary read and writes (from all CPU's and controllers) may proceed as normal. Since ISR need to do SIGNALS, once a CPU has been granted real-lock, it must not acknowledge any interrupts until after write-unlock is done.

3. GOALS AND IMPLICATIONS

G1. Provide process mechanisms for synchronization and micro-scheduling which will be efficient for a wide range of systems.

I1.1 Must be good for real time, transaction processing, and time sharing.

I1.2 Recognize that there are two divergent techniques for handling calls to the operating system:

Process based--a separate process or set of processes is invoked (RC4000, RSX11-D, CHIOS, CSTS).

Procedure based--a set of procedure calls is made as part of the user process (TOPS10, SNARK, MULTICS).

The hardware primitives should be equally efficient for both kinds of systems. Otherwise the hardware primitives will influence the software architecture.

G2. Provide a wide range of inter-process communication, synchronization and micro-scheduling primitives from simple, fast to general, lengthy.

I2.1 The speed of interprocess communication should be comparable in speed and space to procedure calls over the range of simple to general for each.

I2.2 Include message passing as well as synchronization.

I2.3 Provide same primitives for users.

G3. Process synchronization must be the same for single and multi-processor systems.

I3.1 Do not use priority for exclusion, only for urgency of execution. Use semaphores for exclusion.

G4. Include RAS considerations.

I4.1 Need a mechanism to time out indefinite WAIT.

I4.2 Need a mechanism to abort excessively long use of a semaphore.

NG5. Do not provide dead-line scheduling mechanisms.

4. TERMINOLOGY

Process--A virtual machine with a PC, registers and a map which can be executed by a CPU. A PCB completely describes a process. Processes run only at interrupt BR level 0.

Interrupt--A change in the instruction sequence in response to a request from an external, asynchronous device, controller, or CPU. Interrupts use an -11 like vector mechanism to get started. Interrupts run at interrupt levels BR 1-7. Interrupts are really cheap forms of processes which have no PCB and deactivate themselves at the end of execution. However to prevent confusion, they will never be referred to as processes.

ISR--An interrupt Service Routine.

Trap--a change in the instruction sequence of a process (or an ISR) due to an event caused by the execution of the current instruction in the CPU, e.g. floating overflow. There is no change in BR level when a trap occurs, so traps can occur at BR levels 0-7 (0 is most common). Thus a trap is like a special form of procedure call rather than a separate process. The only reason that different registers may be used is to handle errors (bad user stack) or improve efficiency. However, a trap routine may decide to SIGNAL another process to handle the work.

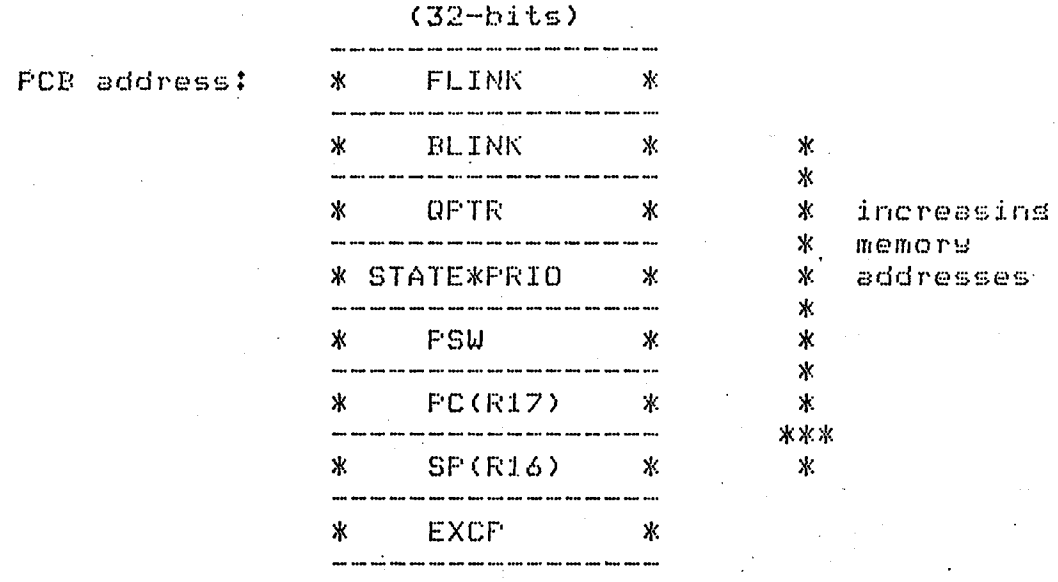
Pre-empt--A change from one process (BR level 0) to a higher priority process (BR level 0) on the same CPU. This is analogous to interrupts except that it involves two processes both running at the lowest interrupt level (BR 0).

Interrupt latency--The elapsed time from an external event to the first useful instruction in the ISR.

Pre-emption latency--The elapsed time from a SIGNAL instruction executed by a process, ISR or controller until the first useful instruction in the first waiting process.

5. PROCESS DEFINITION

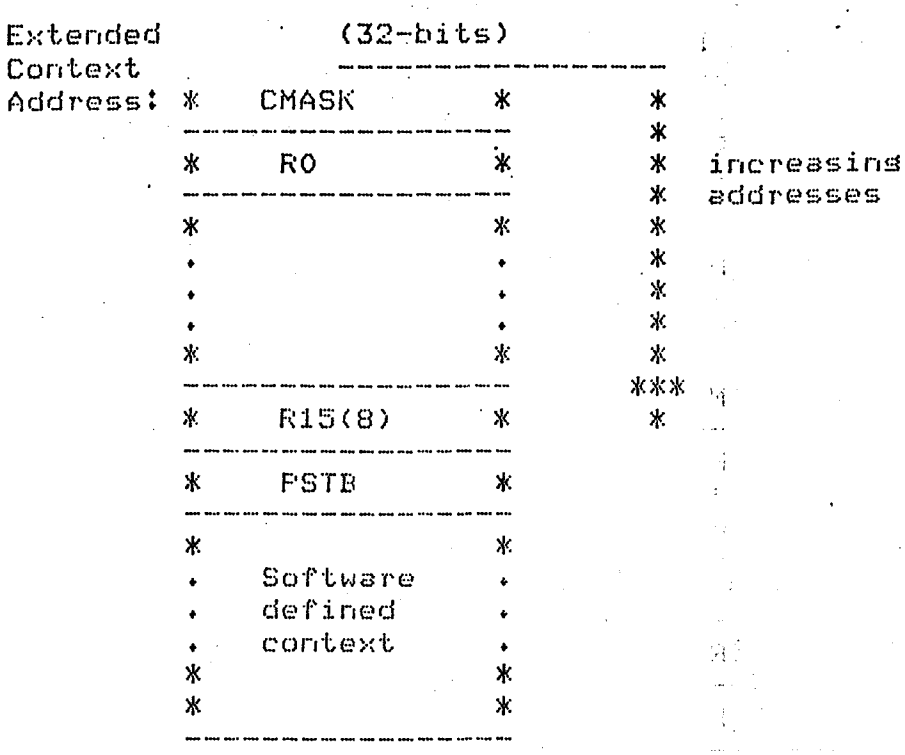
A process is defined by its process control block (PCB) which is an 8 word (32 bit) block in MP. The format of a PCB is:



The forward link (FLINK) and backward link (BLINK) are used to link the PCB in a doubly linked queue structure. The queue pointer (QPTR) points to the head of the queue in which the PCB is linked. The STATE field indicates the state of the process and will be defined later.

The priority field (PRIO) indicates the priority of the process. .Currently there are 16 priority levels, 0-15, with priority 15 the highest

The PSW, PC, and SP fields are the processor status word, program counter and stack pointer, of the process respectively. The extended context pointer (EXCP) points to additional process context. The form of this is as follows:



The context mask (CMASK) additional hardware context for the process. Bits 0 through 13 define whether registers R0 through R15(8) are defined for the process. Bit 14 of CMASK defines whether the process has a private segment table base pointer (PSTB). R0 - R15(8) and PSTB follow CMASK. Additional software defined context, if any, follows. The two part PCB means that each process known to the hardware need have only a minimal eight word block in memory. This facilitates swapping the rest of a waiting process out of memory.

6. PROCESS STATES

A process can be in one of four states:

1. Running - a process is running on a CPU.

2. Runnable - a process is runnable and waiting for a CPU.

3. Waiting - a process is waiting on a semaphore.

4. Deactivated - a process is deactivated.

There is a queue or set of queues for each of the process states 1 through 3. A process in state 4 is effectively out of the process structure and there is no specific queue for deactivated processes. States 1 and 4 actually have two substates each:

1a. Running

1b. Running with deactivate flag set.

4a. Deactivated from run.

4b. Deactivated from wait.

The reason for these substates will be clarified later.

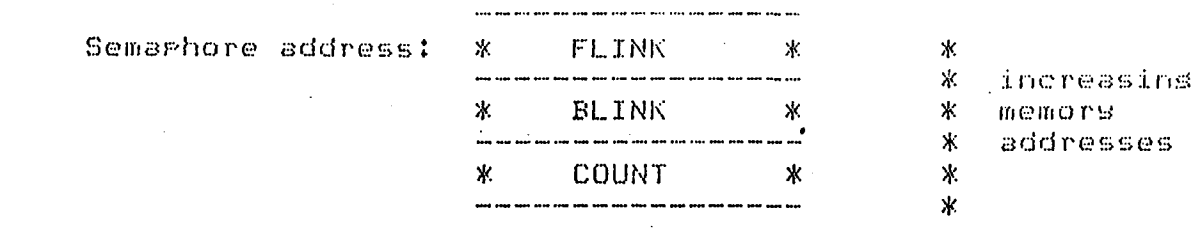
7. SEMAPHORES

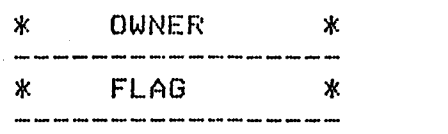
Processes synchronize their execution and control access to shared resources with semaphores. Three primitive operations are defined for semaphores: WAIT and SIGNAL and REQUEST. Informally, when a process performs a WAIT on a semaphore, the count field of the semaphore is decremented. If the decremented value is ≥ 0 the process continues and becomes owner of the semaphore; otherwise it is put into the wait state. When a process ISR or controller performs a SIGNAL on a semaphore, the count field of the semaphore is incremented. If the incremented value is ≤ 0, the process represented by the first PCB on the semaphore queue is put into the runnable state and becomes owner of the semaphore. A REQUEST tests the count field of a semaphore. If it is ≥ 1, the request operation effectively performs a WAIT which immediately makes the process the owner and always continues); otherwise the process continues.

8. PROCESS QUEUES

1. There is one currently running process queue (CRPQ) for each CPU in the system. This queue contains exactly one PCB which is the PCB of the process currently running on that CPU. The CRPQ is pointed to by the current process pointer (CPP) a per CPU hardware resister.

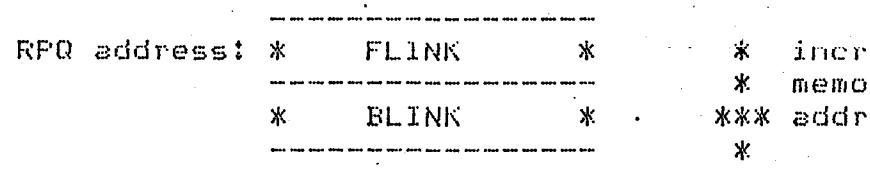
2. There is one semaphore queue (SQ) for each semaphore defined in the system. The semaphore is defined by a 5 word semaphore control block (SCB) which has the following format:





The FLINK field points to the first PCB in the semaphore queue while BLINK points to the last. If the queue is empty FLINK points to BLINK and BLINK to FLINK. The semaphore queues are accessible to all CPU's in a multiprocessor system. The COUNT field indicates the excess of SIGNALS over WAITS performed on the semaphore (the definition of SIGNAL and WAIT will be given later). The OWNER field points to the PCB of the process which owns or last owned the semaphore. If the count field of the SCB is zero, the owner field is not used by the hardware. The FLAG field is reserved for software and for recovering from excessively lengthy ownership.

3. There are 16 runnable process queues (RPQ0-RPQ15): one for each priority level. An RPQ is defined by a two word runnable queue control block (RQCB) which has the following format:



The FLINK and BLINK fields are defined as for an SCB. The RPQ's are accessible to all CPU's in a multiprocessor system. Since software never looks at this word, it may be stored in main memory or in hardware.

In order to facilitate scheduling of processes there is a 16 bit runnable queue summary word (RQSW). Bit i in the RQSW indicates whether RPQi is empty or not. RQSW is accessible to all CPU's in a multiprocessor system.

9. PROCESS SCHEDULING

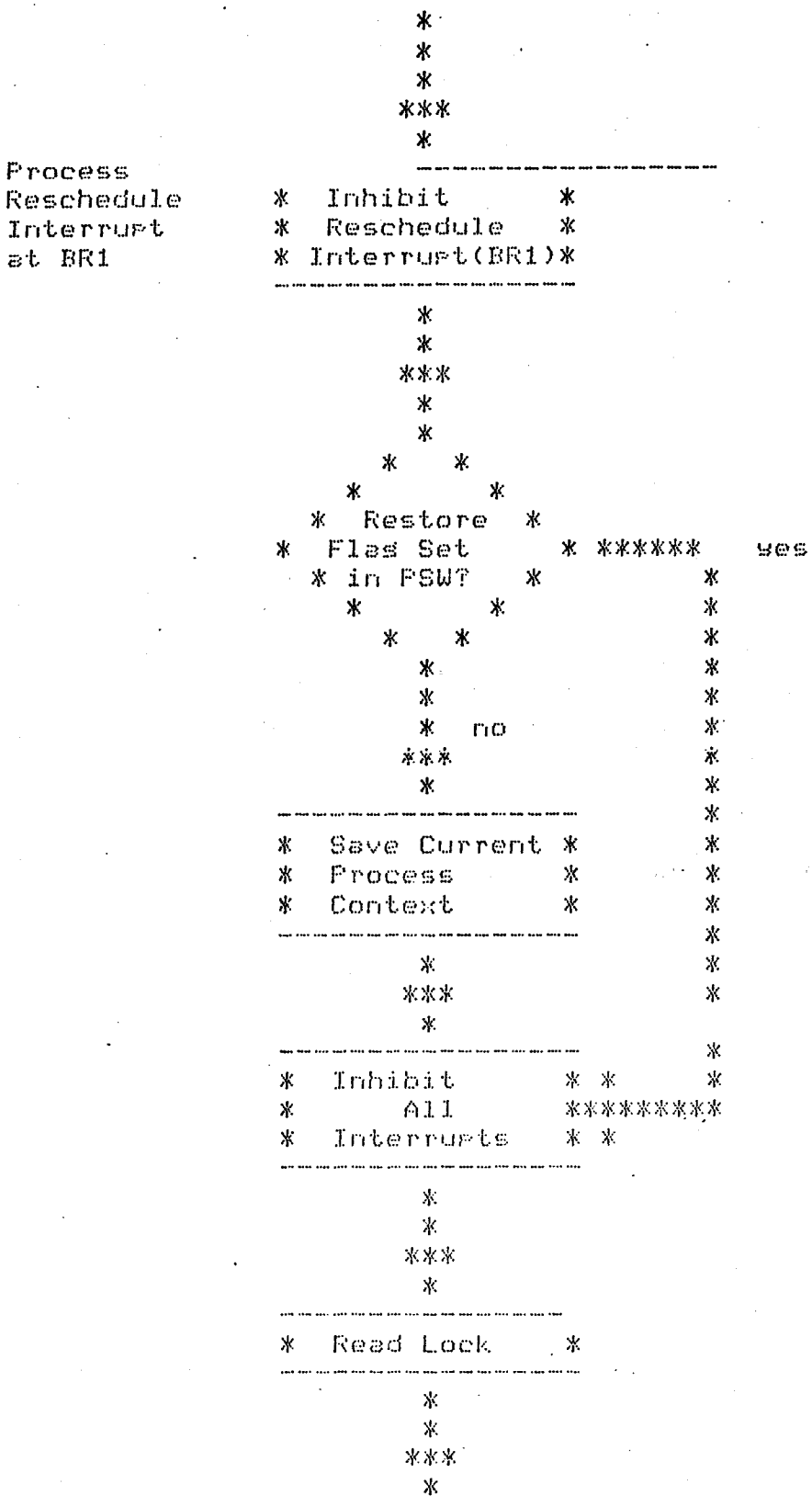
In an n-CPU system, the objective of process scheduling is to keep the n highest priority runnable processes running. To effect this, each CPU has process scheduling logic (i.e. microcode) and there is a specific inter-CPU interrupt mechanism. Whenever a CPU executes a process structure primitive (to be defined later) which requires a process rescheduling in order to meet the process scheduling objective, it examines the state of the process queues and the priorities of the currently running processes and decides which CPU should

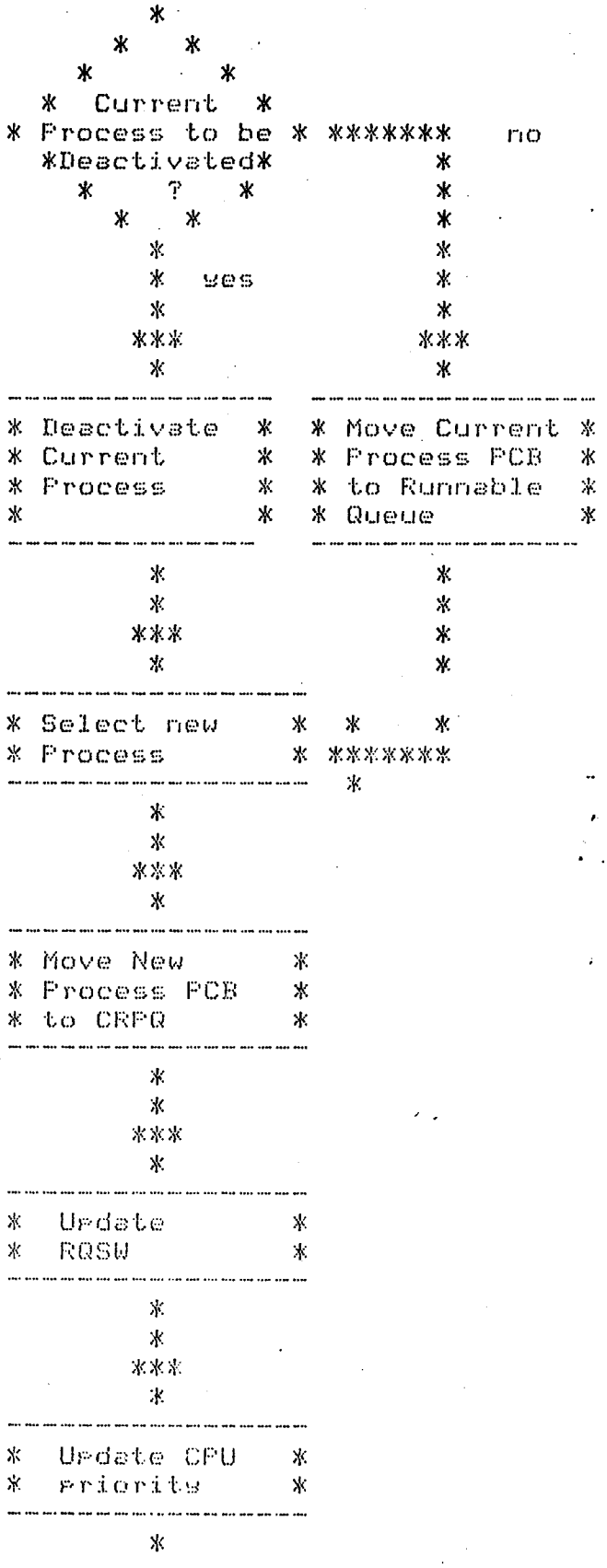
reschedule. It then generates a reschedule interrupt on that CPU. This interrupt occurs at a level (BR1) which is higher than the level at which processes run (BR0) and lower than the level at which I/O device initiated interrupt service routines run (BR2-BR7). A CPU may initiate its own reschedule interrupt.

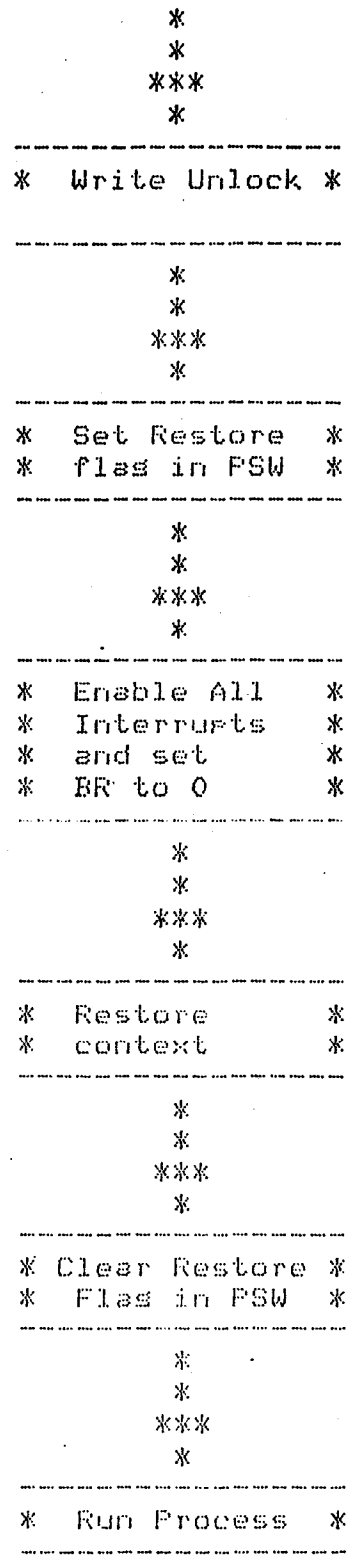
When a CPU receives a reschedule interrupt it effectively has been notified that it should be running a higher priority process than it is currently running. The interrupted CPU saves the context of the process it is currently running, examines the RQSW, selects a new process to run, restores its context, and proceeds to run it.

As noted earlier, the reschedule interrupt occurs at BR1. During the majority of the reschedule operation, context saving and restoring, interrupts can occur at higher BR levels. The operation of examining the RQSW and selecting a new process is necessarily run interlocked (i.e. using the hardware READ lock and WRITE unlock operations) and run with interrupts disabled (on the CPU doing the rescheduling).

In order to further enhance the scheduling objective all interrupts, including reschedule interrupts, are enabled during the context restore operation. Such a rescheduling interrupt could occur, if, during the restoring of new process context, a still higher priority process becomes runnable on the CPU. To handle this properly a restore flag in the PSW is associated with reschedule interrupt servicing.


Process
Reschedule
Interrupt
at BR1
Inhibit
Reschedule
Interrupt(BR1)
Restore
Flag Set
in PSW?
yes
no
Save Current
Process
Context
Inhibit
All
Interrupts
Read Lock


Current Process to be Deactivated ?
yes
no
Deactivate Current Process
Move Current Process PCB to Runnable Queue
Select new Process
Move New Process PCB to CRPQ
Update RQSW
Update CPU priority


Write Unlock
Set Restore flag in PSW
Enable All Interrupts and set BR to 0
Restore context
Clear Restore Flag in PSW
Run Process

10. PROCESS STRUCTURE INSTRUCTIONS

There are six process structure instructions defined:

1. ACTIVATE (PCB Address, mask)

2. DEACTIVATE (PCB Address, mask)

3. CHANGE-PRIO (PCB Address, mask)

4. SIGNAL (Semaphore Address, mask)

5. WAIT (Semaphore Address, mask)

6. REQUEST (Semaphore Address)

The ACTIVATE instruction takes a PCB which has been assembled by software and makes it known to the process structure hardware by inserting it into a runnable process or semaphore queue. This potentially causes a reschedule operation.

The DEACTIVATE instruction removes a PCB from a process queue. This makes the process unknown to the process structure hardware.

The CHANGE-PRIO instruction changes the priority of a process, and, if necessary, forces a reschedule operation.

The SIGNAL operation effectively gives up ownership of a semaphore. If there is a process waiting for ownership of the semaphore that process is made runnable. If there are several processes waiting for ownership of the semaphore, the first one asking for ownership is made runnable. Hence a semaphore queue is FIFO.

The WAIT instruction causes a process to wait for ownership of a semaphore. If the semaphore is free the process continues; otherwise it is placed in the wait state and its PCB is linked to the end of the semaphore queue. In the latter case, a reschedule operation is initiated.

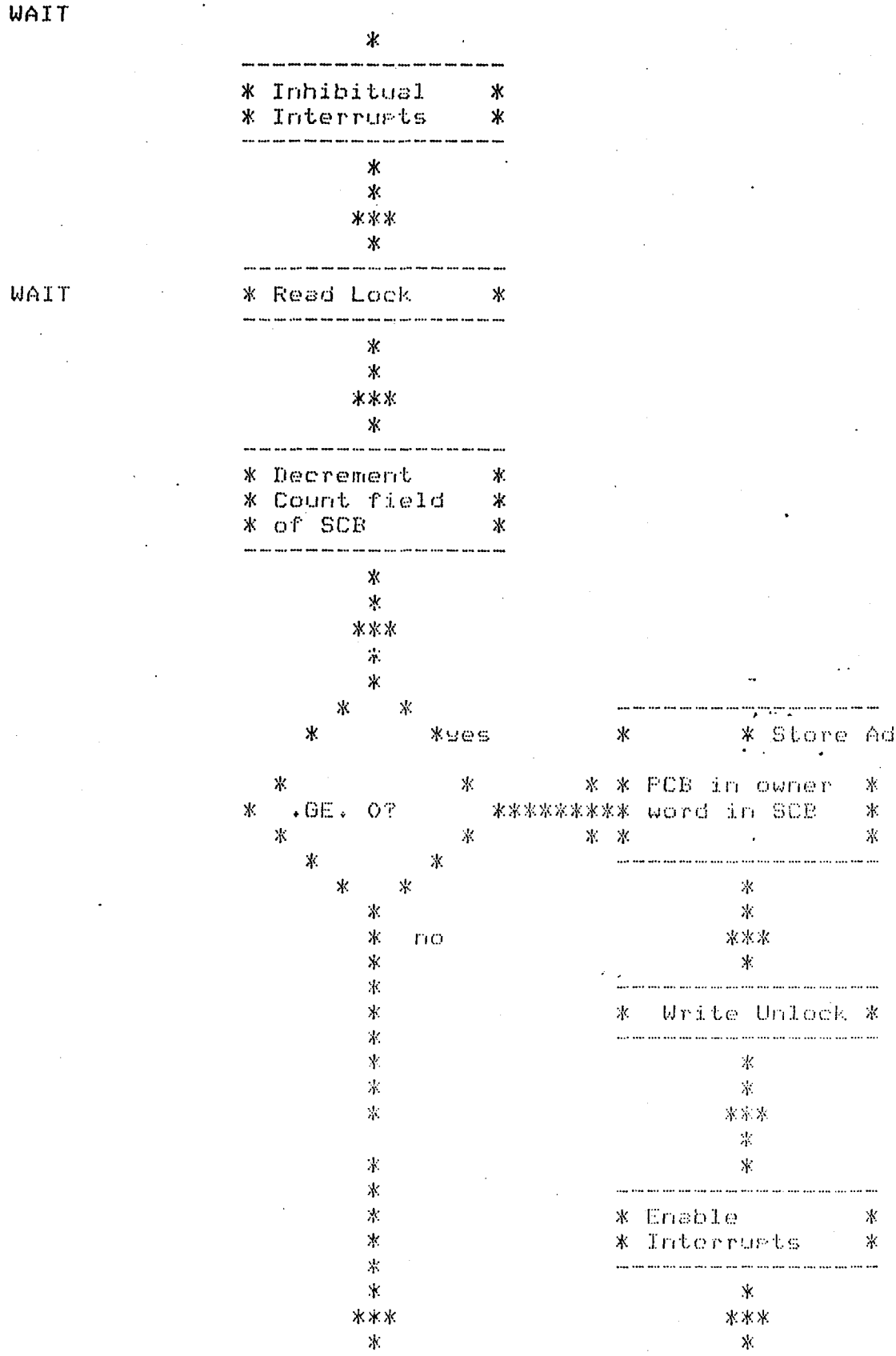
The REQUEST instruction requests ownership of a semaphore. If the semaphore is free, the process becomes the owner of the semaphore and continues; otherwise the process just continues. Which of those outcomes occurred is indicated by condition codes.

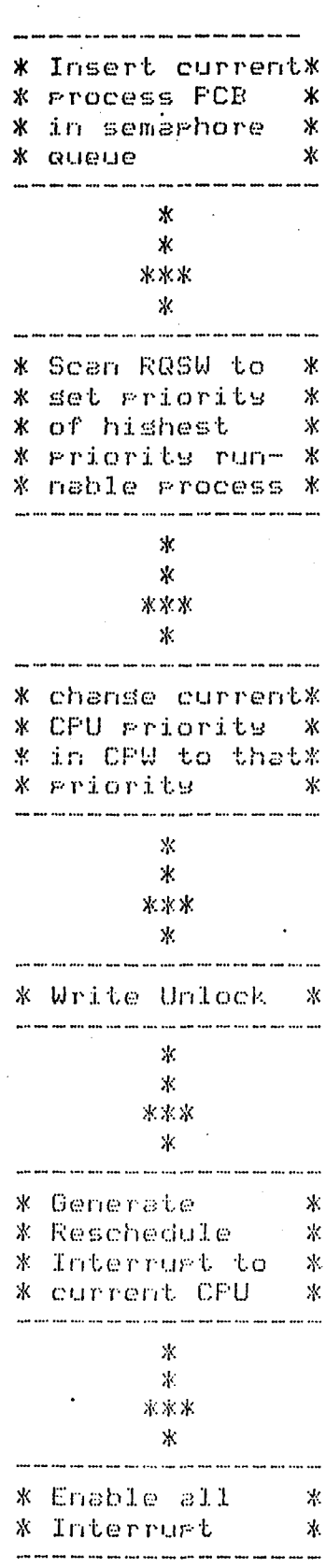
All instructions other than REQUEST potentially result in process rescheduling. The process executing the instructions may be removed from a run queue to a wait queue or runnable queue, or it may be deactivated. In these cases, its context must be saved. The 14 bits of

the 16-bit mask operand in the instructions indicates which of the general registers r0-r15(8) are to be preserved if the process is rescheduled. (Actually, the mask is and'ed with CMASK in the process PCB to determine which registers to save. The value of CMASK is not changed, however and is used to restore the process later).

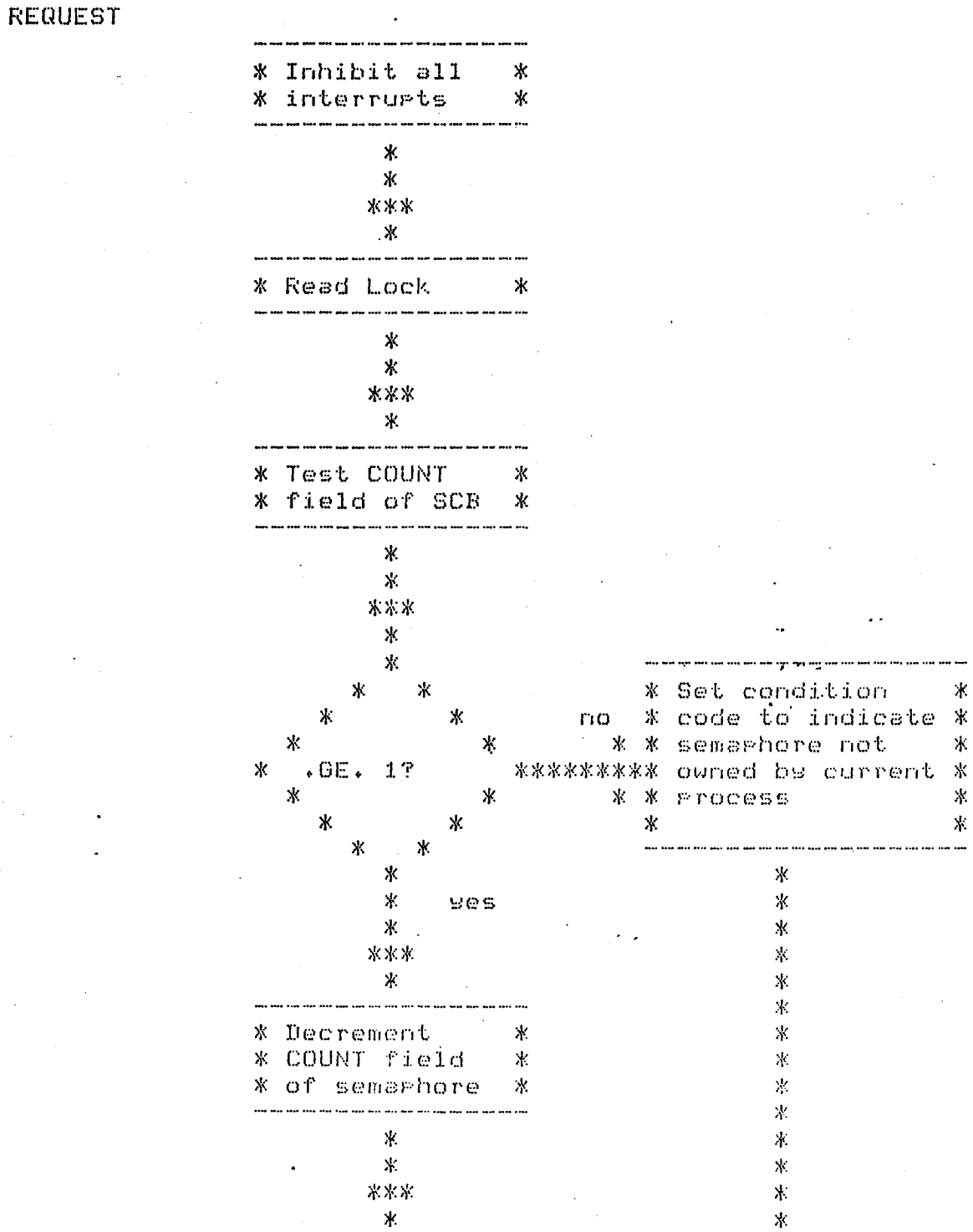
In the following sections of this paper, flowcharts of the process instruction instructions are given. In order to facilitate process scheduling, a 32-bit CPU prio word (CPW) is introduced. its format is:

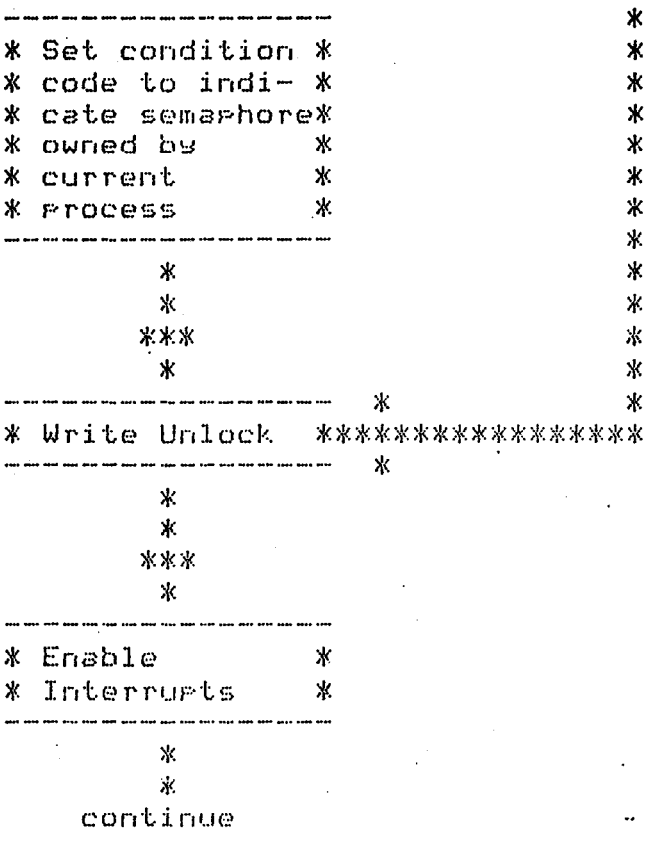
This word is accessible to all CPU'S in a multiprocessor system and it nominally contains the priority of the process currently running on each CPU. In order to facilitate process scheduling, it may transiently contain a higher priority than the priority of the process the CPU is running. However this occurs only when the CPU is about to reschedule.


WAIT
Inhibitual Interrupts
WAIT
Read Lock
Decrement Count field of SCB
.GE. 0?
yes
no
Store Adr. of PCB in owner word in SCB
Write Unlock
Enable Interrupts

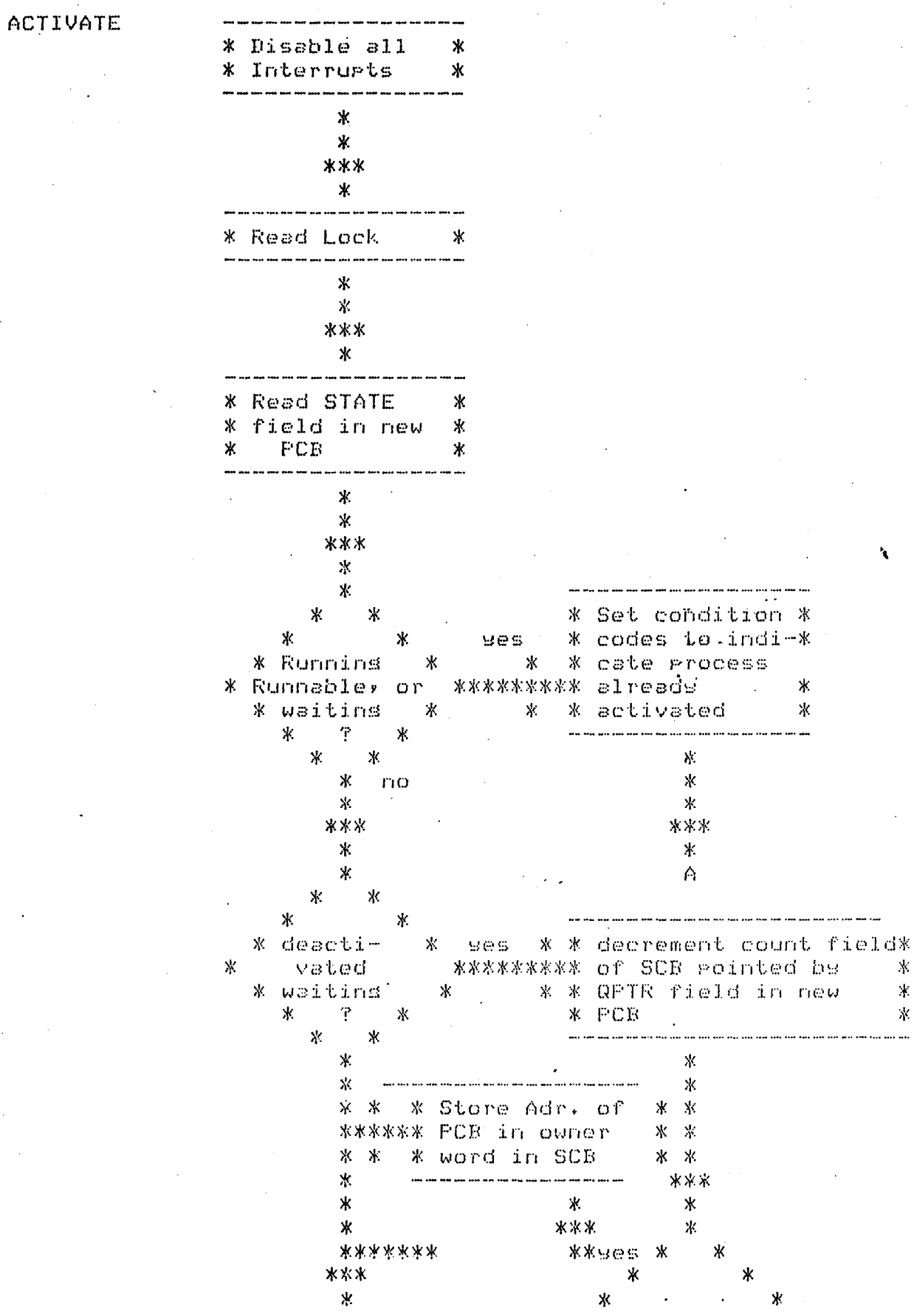

Insert current process PCB in semaphore queue
Scan RQSW to set priority of highest priority run-nable process
change current CPU priority in CPW to that priority
Write Unlock
Generate Reschedule Interrupt to current CPU
Enable all Interrupt

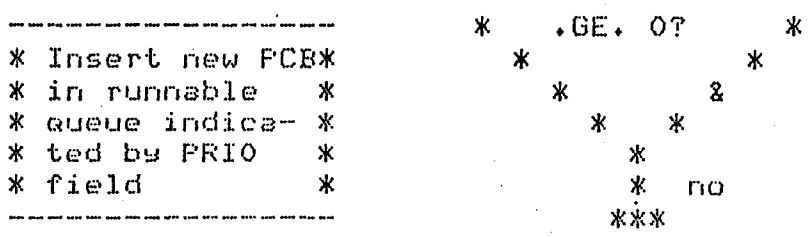
continue

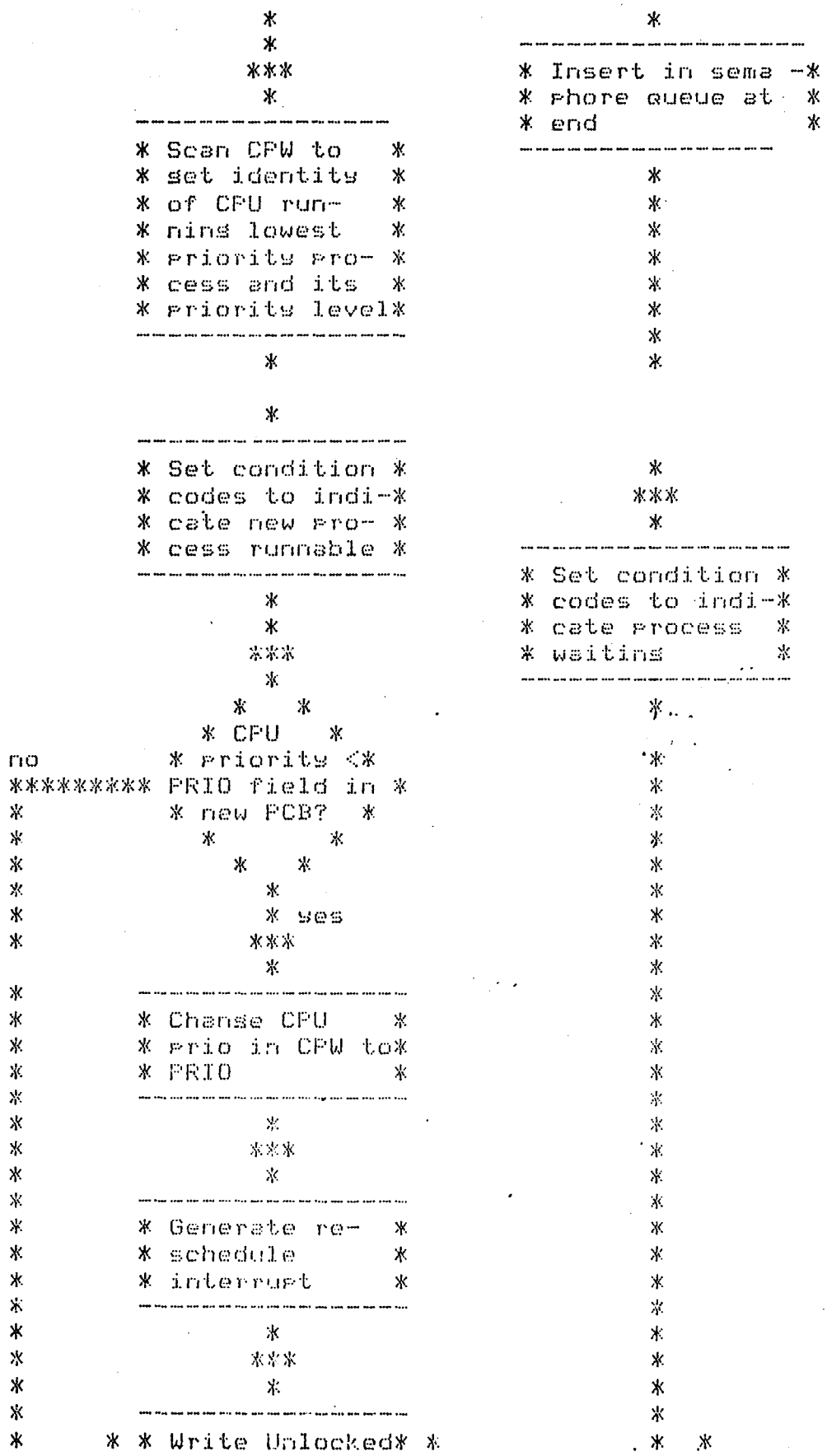

REQUEST
Inhibit all interrupts
Read Lock
Test COUNT field of SCB
.GE. 1?
no
yes
Decrement COUNT field of semaphore
Set condition code to indicate semaphore not owned by current process

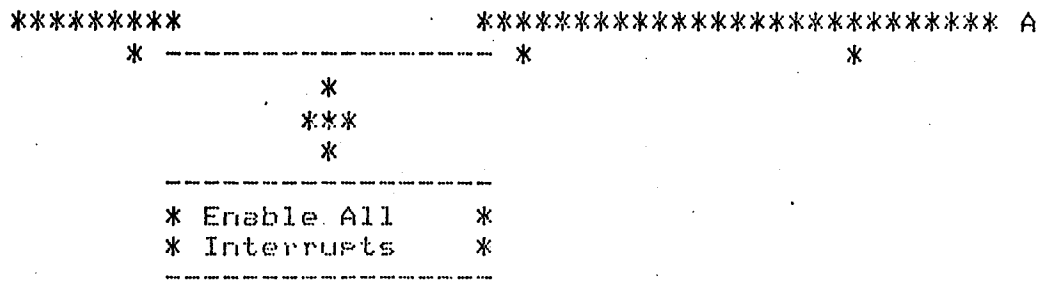

Set condition code to indi- cate semaphore owned by current process
Write Unlock
Enable Interrupts
continue

ACTIVATE

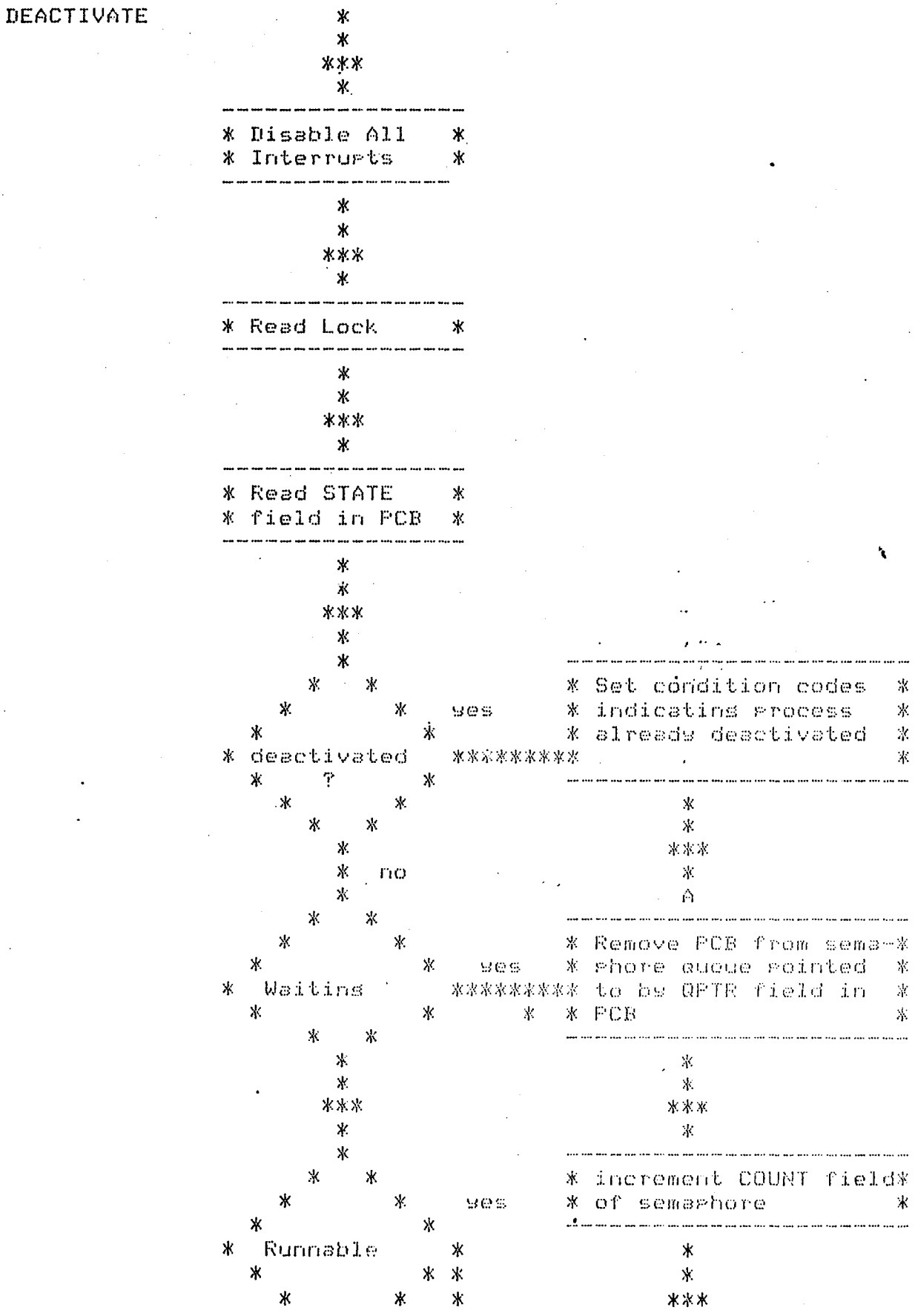

Disable all Interrupts
Read Lock
Read STATE field in new PCB
Running Runnable, or waiting
yes
Set condition codes to.indicate process already activated
no
deactivated waiting
yes
decrement count field of SCB pointed by QPTR field in new PCB
Store Adr. of PCB in owner word in SCB
yes

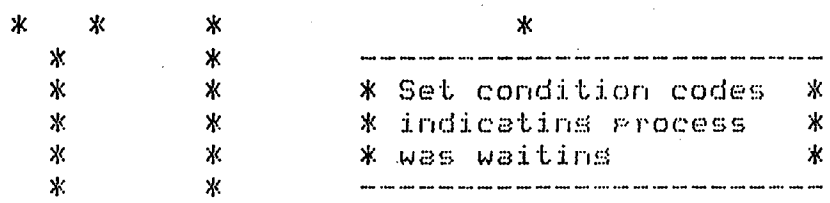

Insert new PCB in runnable queue indicated by PRIO field
.GE. 0?
no

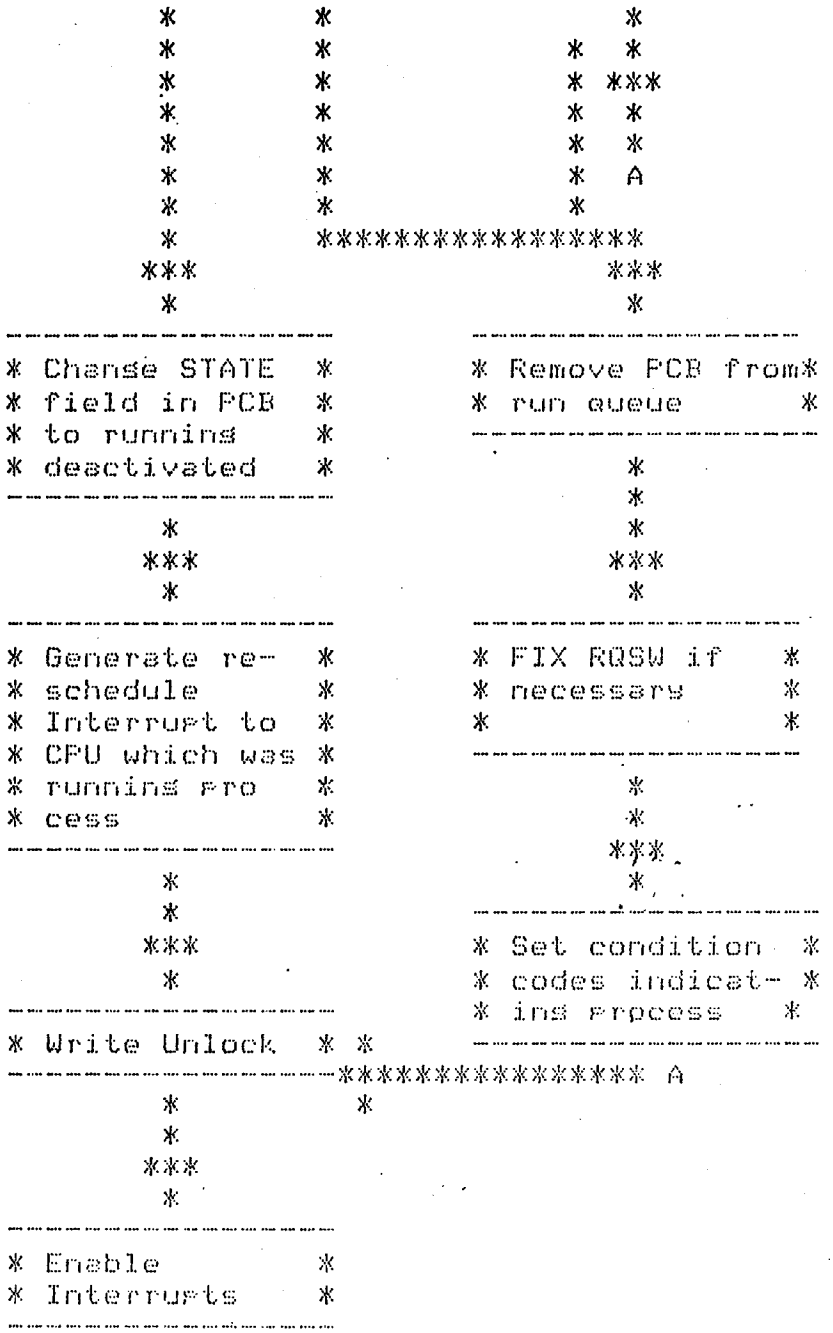

Scan CPW to set identity of CPU running lowest priority process and its priority level
Set condition codes to indicate new process runnable
CPU priority < PRIO field in new PCB?
no
yes
Change CPU prio in CPW to PRIO
Generate re-schedule interrupt
Write Unlocked
Insert in semaphore queue at end
Set condition codes to indicate process waiting


A
Enable All
Interrupts

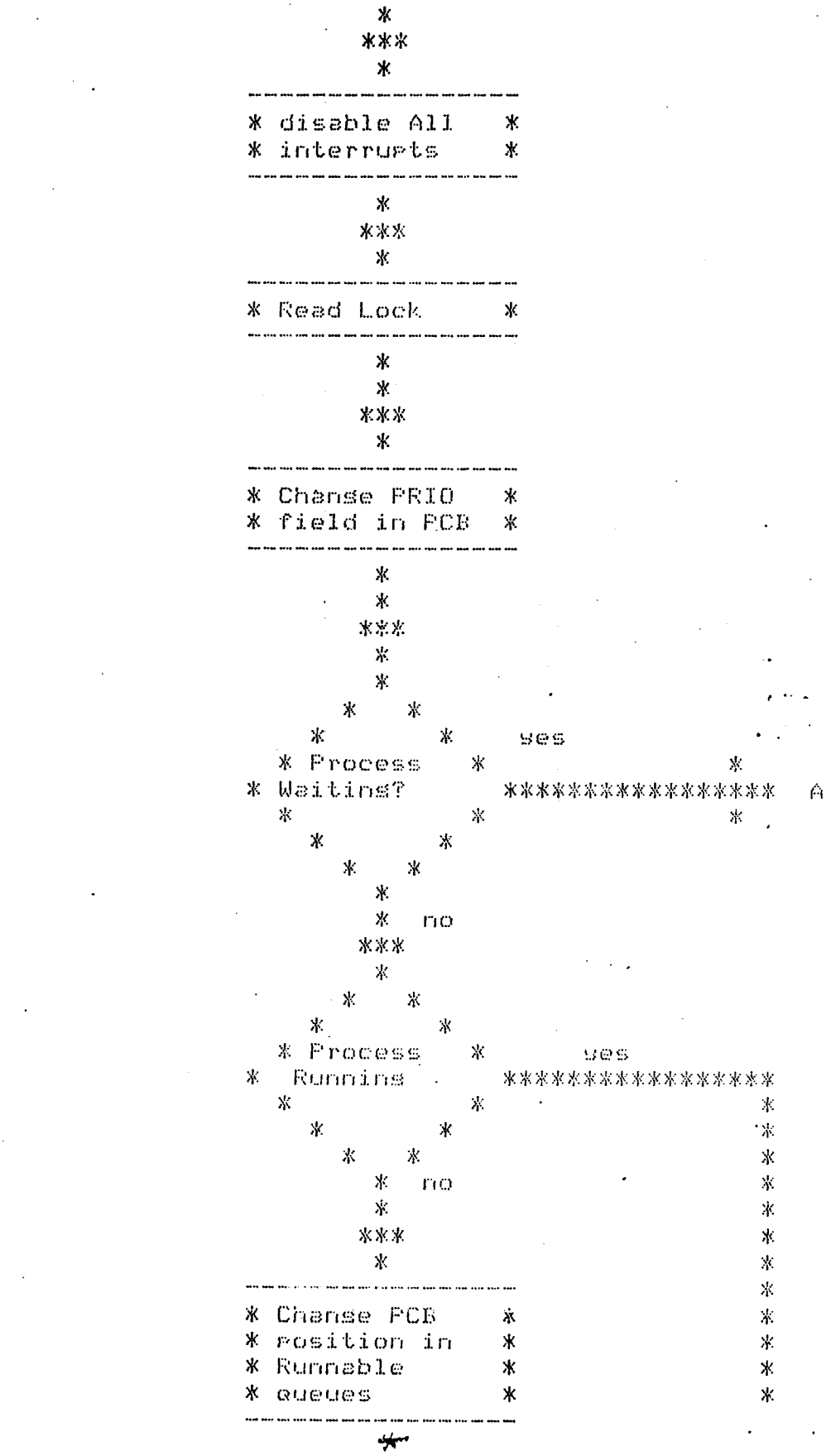
DEACTIVATE

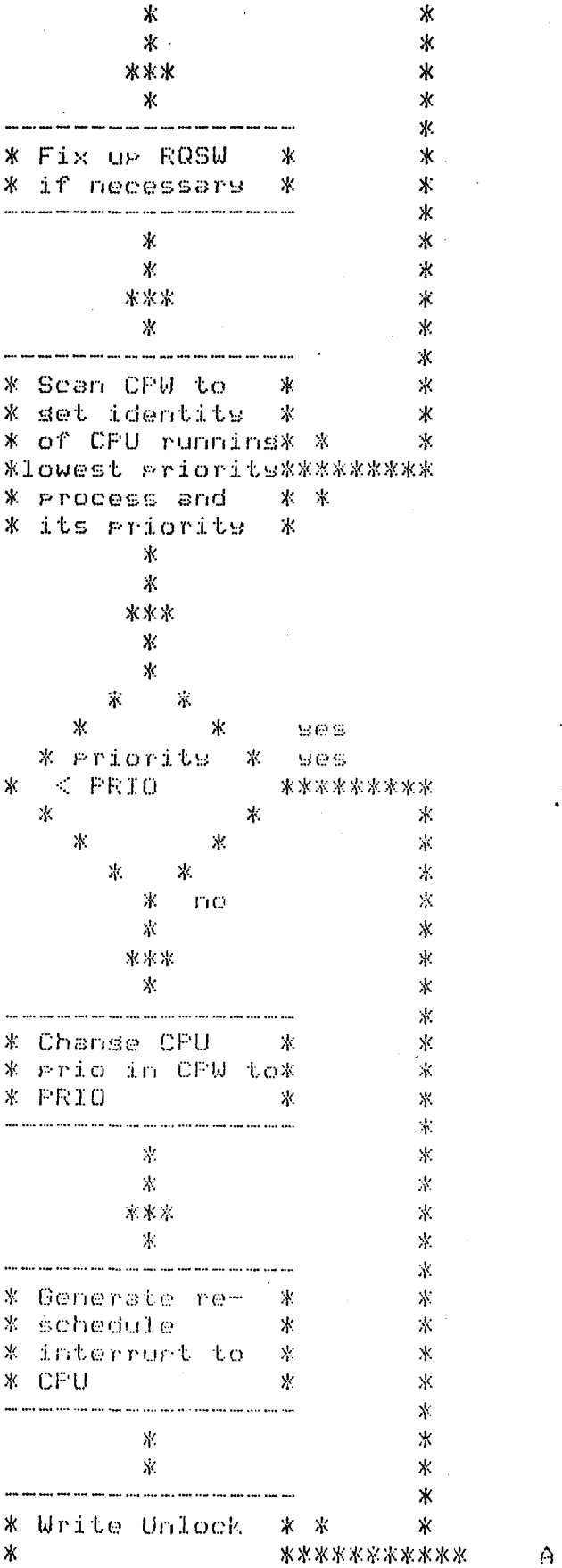

Disable All Interrupts
Read Lock
Read STATE field in PCB
deactivated ?
yes
Set condition codes indicating process already deactivated
no
Waiting
yes
Remove PCB from semaphore queue pointed to by QPTR field in PCB
Runnable
yes
increment COUNT field of semaphore

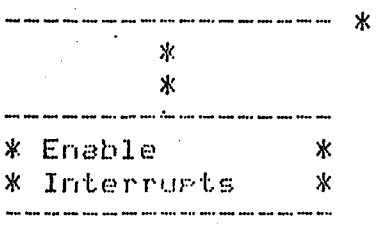

Set condition codes
indicating process
was waiting


Change STATE field in PCB to running deactivated
Generate re-schedule Interrupt to CPU which was running process
Write Unlock
Enable Interrupts
Remove PCB from run queue
FIX RQSW if necessary
Set condition codes indicating process
A
A

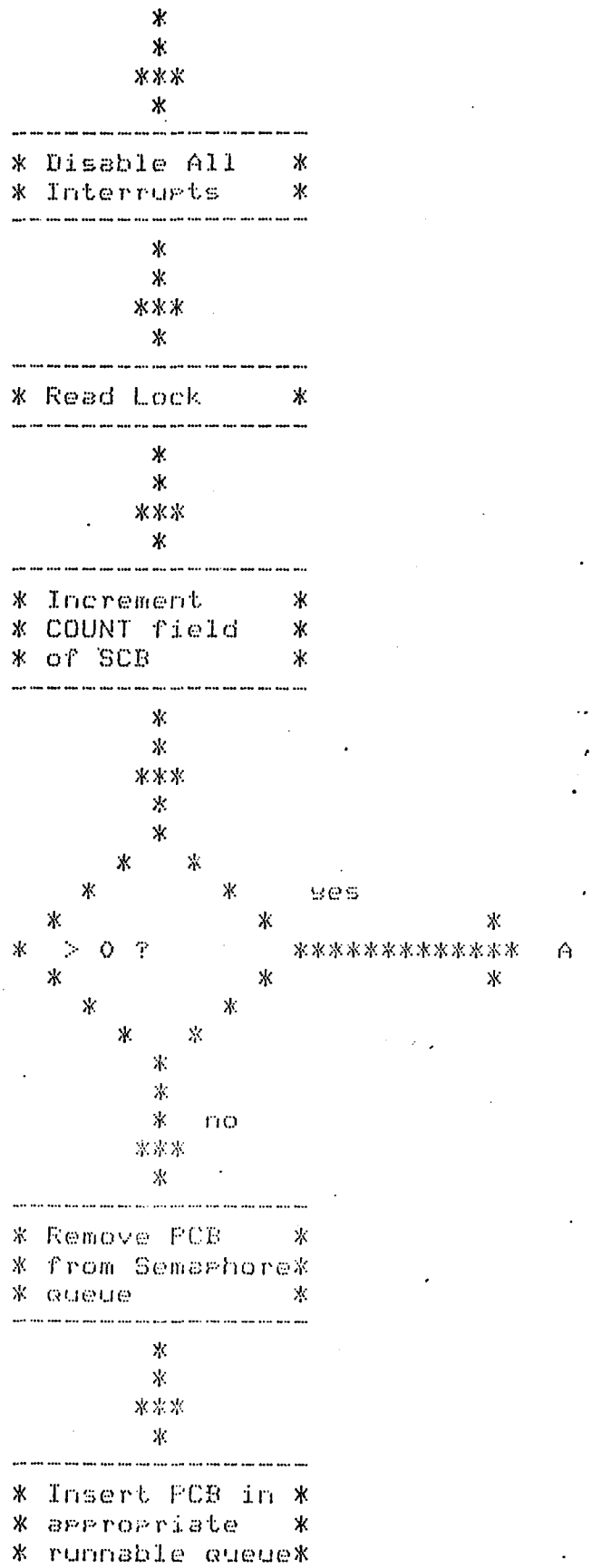
CHANGE PRIORITY

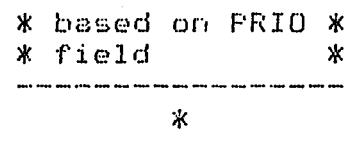


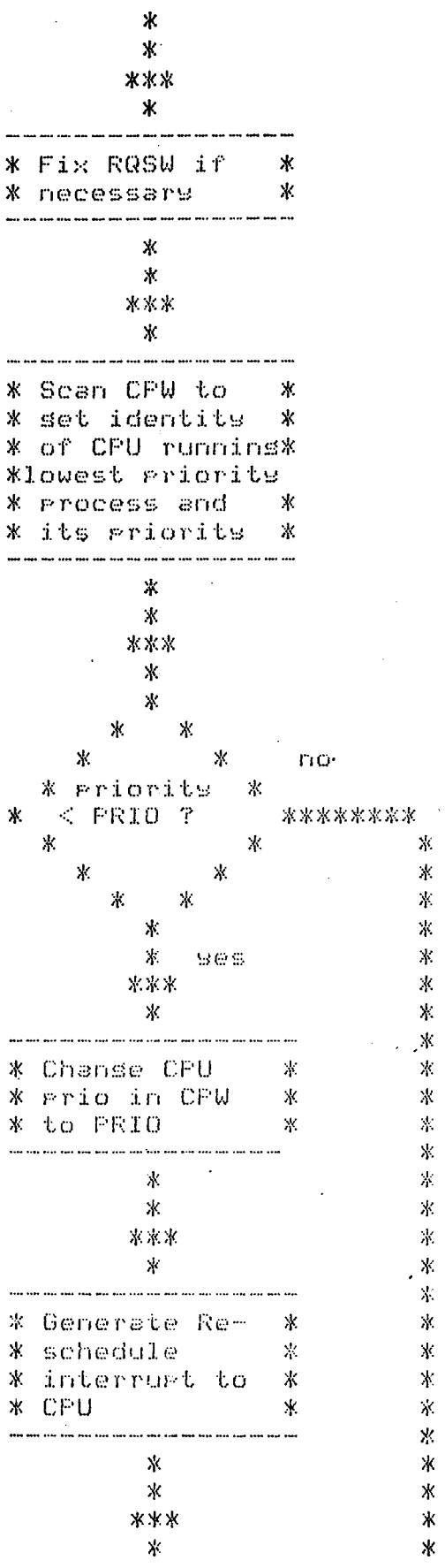

Fix up RQSW if necessary
Scan CPW to set identity of CPU running lowest priority process and its priority
priority < PRIO
yes
yes
no
Change CPU prio in CPW to PRIO
Generate re-schedule interrupt to CPU
Write Unlock
A

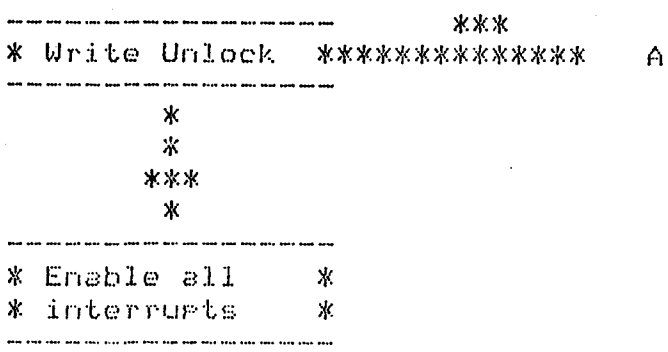

*
*
*
Enable
Interrupts

SIGNAL


Disable All Interrupts
Read Lock
Increment COUNT field of SCB
> 0 ?
yes
no
A
Remove PCB from Semaphore queue
Insert PCB in appropriate runnable queue


* based on PRIO *
* field *


Fix RQSW if necessary
Scan CPW to set identity of CPU runnings lowest priority process and its priority
priority < PRIO ?
no
yes
Change CPU prio in CPW to PRIO
Generate Re-schedule interrupt to CPU


Write Unlock
A
Enable all
interrupts

```
.end literal
.CENTER;11.##<SOFTWARE <INTERRUPTS
.CENTER;------------------------
.BLANK 1
.LM 5
^SOFTWARE INTERRUPTS ARE A TECHNIQUE FOR ALLOWING A USER TO
HANDLE CONCURRENT ACTIVITIES HIMSELF.  ^A SOFTWARE INTERRUPT
MAY BE DUE TO ^CONTROL ^C, END OF RECORD, END OF FILE, ETC.
.BLANK 1
^SOFTWARE INTERRUPTS CAN BE HANDLED IN ANY ONE OF THREE WAYS,
IN ORDER OF DECREASING GENERALITY, SPACE, AND SPEED:
.BLANK 1
.LM9.INDENT -4
1.##^AS SEPARATE PROCESSES WITH SEPARATE MAPS.  ^THE CONTENTS
OF THE MAP BEING IDENTICAL, EXCEPT FOR SEPARATE STACK SEGMENTS.
^CONCURRENCY COULD BE ALLOWED, BUT PRIORITY LEVELS CANNOT BE
USED FOR SYNCHRONIZATION.
.BLANK 1
.INDENT -4
2.##^AS SEPARATE PROCESSES WITH A SHARED MAP.  ^THE OPERATING
SYSTEM
MUST CHANGE THE STACK SEGMENT ENTRY IN THE <PST ON EACH
INTERRUPT IN ORDER TO PREVENT THE INTERRUPT LEVEL FROM
MODIFYING THE INTERRUPTED
STACK.
^IN A MULTI-PROCESSING SYSTEM THE OPERATING SYSTEM MUST DEACTIV
ATE
THE MAIN PROCESS BEFORE CHANGING THE MAP.
.BLANK 1
.INDENT -4
3.##^AS A SINGLE PROCESS WHICH TIME SHARES A SINGLE <PCB.
^THE OPERATING SYSTEM WOULD <DEACTIVATE THE MAIN PROCESS,
SAVE THE <PCB, THEN MARK <PCB RUNNABLE, AND <ACTIVATE IT AGAIN.
^WHEN THE INTERRUPT IS OVER, THE OPERATING SYSTEM MUST COPY
THE <PCB BACK AND <ACTIVATE THE ORIGINAL PROCESS.
.BLANK 1
.CENTER;12.##<SWAPPING
.CENTER;---------------
.BLANK 1
.LM 5
^THE SWAPPER ALWAYS <DEACTIVATES A PROCESS BEFORE SWAPPING IT
OUT.  ^THE PROCESS STATE TRANSITIONS ARE:
.BLANK 1
.literal
                Old State                    New State

        1a running (not possible)---------
        1b running with deactivte flag (not possible)
        2 runnable                           deactivate-run
        3 waiting                            deactivate-wait
        4a deactivate-run                    deactivate-run
        4b deactivate-wait                   deactivate-wait


If the process was in state  3,  DEACTIVATE  removes  it
from the wait queues so that it is no longer waiting for
```

the semaphore. The swapper may wish to be told when the process can own the semaphore, so it can ACTIVATE a vestisial process and put it back in the semaphore queue (at the end). This vestisial process uses the real processes 8-word PCB after the swapper has copied and changed the PC. This vestisial process runs the scheduler which decides when to swap in the rest of the real process. When the real process is swapped in, the vestisial process appends the real PCB information and jumps to where the real process had been stopped.

If the process was in state 2, DEACTIVATE removes it from the run queues. In this case no process information need be left in core for the hardware. When the real process is swapped back in, the swapper ACTIVATES the process to place it back in the run queues.

13. SENDING MESSAGES

There are several types of mechanisms for passing messages between processes depending on:

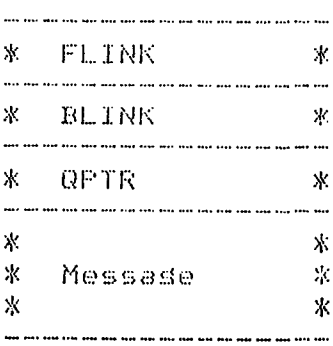
1. Message arguments are by value or reference.

2. The sender wants a completion message returned (including error status) or not.

3. The Sender can be trusted or not.

To illustrate the use of the primitives, lets assume arguments by-value, sender wants a completion message and the sender can be trusted.

Each process, Pi, has associated with it a semaphore, Si, for receiving messages and completion messages. Each process also has a single queue of message control blocks (MCB). The first 3 words look like any other doubly threaded queue. The rest of the MCB contains the message.



The Sender performs the following steps:

1. Fill up MCP with message.
2. Add MCB to end of Receivers queue, Qr.
3. SIGNAL Receivers message queue, Sr.
4. WAIT on Senders message queue, Ss.
5. Remove acknowledgement message from front of Qs.
6. Check error status.

The Receiver performs the following steps:

1. WAIT for receiver message queue, Ss.
2. Remove message from front of Qr.
3. Carry out work indicated.
4. Fill up the MCB with completion status.
5. Add MCB to end of Senders queue, Qs.
6. SIGNAL Senders message queue, Ss.
7. Go to 1.

In order to solve the more difficult problem of an untrustworthy sender, three problems must be solved:

1. The sender cannot write in the first 3 words of the MCB after it has been entered into the queue.
2. The sender must have a limit on the number of messages he can send.
3. The sender must not be able to send messages unless the message queue is public or the sender has been authorized.

Problem 3 can be solved using the Access Modes of the mapping hardware along with the indirect capability. Problem 1 can be solved by keeping MCB'S in a highly protected Access Mode and giving out read-only indirect pointers which point to the link words of MCB's and read-write pointers which point to the message part of MCB's. The software can solve Problem 2 by limiting the number of indirect pointers given to a process.

14. UNRESOLVED ISSUES

---

1. How is rescheduling implemented when there is a CPU - process binding (i.e. certain processes can only run on certain CPU's)?

2. Are formal message primitives needed and, if so, how do they work?

3. Do semaphores owned by a process need to be linked together so they can be reclaimed when a process terminates?

4. How are PCB's and Semaphores addressed:

   1. Physical addresses,
   2. Virtual addresses, or
   3. Special name space (i.e., system index)?

   How are they protected?

5. Should there be a priority associated with a semaphore and how does this interact with process priority during the time a process owns a semaphore?

6. What are the RAS issues:

   For RAS considerations, it must be possible for the system to detect deadly embraces, semaphores which have been owned too long and processes which have waited too long. Therefore, each time a semaphore becomes owned, WAIT and REQUEST set a bit in the SCB which a watch dog process can clear from time to time. Similarly, each time a process enters a wait queue, a bit is set in the PCB which a watch dog process can clear to keep track of excessively long waits. How should waiting process be told of excessive wait? How should owning process be told of excessive ownership? How should a process indicate that it wants the watch dog process to be involved?

7. Need a name for Signal which does not conflict with PL1 and BLISS usage (for inter-procedure event communication within a process).

TH:WS:mJK (INTPRT.HS1)

MAKEW

.42
- immed to reg - 33
- reg - reg - 12
- push reg - 8
- reg - vector - 8
- indexed to reg - 8
- vector - vector - 7

0.26
- reg. - 42
- indexed - 33
- vector - 10

FINCOMPILE

.44
- immed to reg - 28
- reg - reg - 15
- {immed vector} vector - 10
- reg - vector - 15
- push reg - 9
- indexed to reg - 7

0.26
- reg - 52
- indexed - 28
- regpntr - 8
- vector - 6

LINKER

.53
- immed to reg - 34
- reg to index - 12 !
- reg - reg - 8
- regpnt - regpnt - 10
- immed vect - vector - 5
- autodec. autodec - 5 ?
- immed - indexed - 4

.08
- reg pnt - 68
- indexed - 13
- reg - 9

SUBJ: 32 BIT HARDWARE IMPLEMENTATION SCHEDULE ASSUMPTIONS

COMPANY CONFIDENTIAL

1.*	Arch. Breadboard Available for Software Development	Early Q2, FY76
2.*	Large System Breakboard Available for Limited Use by Software Development	Late Q3, FY76
3.*	Large System Prototype at Speed Delivered to Software Development	Early Q2, FY77
4.	FCS Large System	Mid Q3, FY77
5.	Small System Prototype Available	Late Q2, FY77
6.	Small System FCS	Q1, FY78
7.	Medium System Prototype Available	Q2, FY78
8.	Medium System FCS	Q3, FY78

---

* I would anticipate architectural differences (hopefully minor) from 1, 2, and 3.

(This plan assumes all 3 systems will contain a 16 bit compatibility mode.)

WRD:mjk [5/28/75, 32SCHD.DEM]

SUBJ: HIGH END IMPLEMENTATIONS OF 32 BIT MACHINES

There are three basic choices for a first implementation of a high end 32 bit machine:

1. 11/70 based
2. Unicorn based
3. Neither

11/70 BASED

The primary reasons for using the 11/70 as the base would be to have the absolute minimla development time and cost. The RH70 and MJ11 memory would be used unchanged, some of the cache control (those parts which interface to memory and the RH70) and UNIBUS MAP could be used with relatively little change, and the CPU, most of the cache, FPP and Memory Management would be completely redesigned.

The primary problems with using the 11/70 as a base are:

1. No multi-processor configurations could be done in any reasonable fashion.

2. The performance would probably not be as good as the other possibilities.

3. It is likely none of the hardware developed would be usable in future implementations of the 32 bit architecture.

There is a potential marketing plus with using the 11/70 base; it might be possible to upgrade 11/70's in the field to 32 bits. This would help reinforce the image of the 32 bit architecture being a member of the PDP-11 family.

UNICORN BASED

The primary reason for using the UNICORN bus (SBI), would be to overcome the 3 limitations listed above for the 11/70. However, it appears there are still several problems:

1. It is likely that none of the hardware (except chips) developed for the UNICORN could be used. The backplane has dedicated slots for the CPU; the memory is 36 bits wide (plus ECC) so would probably be depopulated; the memory control has 36 bit registers; the UNIBUS interface does not have the ability to allow direct memory access from the UNIBUS; and the MASSBUS interface is not 11 like and has 36 bit registers.

2. Multi-processing is specifically a NON-goal of the UNICORN, and it appears likely that some areas of the SBI would have to be changed to implement multi-processors, especially in the area of interrupts.

Thus, it appears using the UNICORN as a base is not really one of the choices for implementing a 32 bit machine.

START ALL OVER AGAIN

This choice would presumably use a Unicorn-like (Dragon-line) bus structure, and would solve all the multi-processing problems. Since it isn't defined, it is obviously the fastest, cheapest, and best, and solves all the known problems. Also since it isn't defined, it also takes the longest to get to market.

MY RECOMMENDATION

I believe what we do depends on the overall strategy we choose:

Strategy A

Do the Unicorn and the 32 bit machine. The Unicorn project slows down a bit, and the 32 bit gets to market ASAP.

Recommendation A

Use the 11/70.

Strategy B

Do the Unicorn and the 32 bit, but the Unicorn goes to market first.

Recommendation B

Start all over again. I believe if the 11/70 were used here, the final product would be too little, too late.

Strategy C

Kill the Unicorn

Recommendation C

It is a toss-up. If the requirement is still that we get to market ASAP, use the 11/70. Otherwise dosomething NEW. Note that it might be reasonable to use the 11/70 as a base, and use some of the dollars that are saved to write software so we have a SYSTEM product. The hardware wouldn't be quite as good, but we would have software.

COMPATIBILITY

Compatibility between 16 and 32 bit families	User Programs	Run old programs with no new capabilities	Hardware support to run 16 bit user programs unchanged. OS32 must provide proper environment. Some utilities (EDIT,PIP) will run this way, maybe forever. Some Some language compilers may run this way while generating code for 32 bit machines. Cost is 15% to 20% additional hardware at low end. Question: do we ever delete this mode from implementations? Low end only?
		Run old subroutines with new programs.	Can't be done. Old subroutine expects old addresses.
		Run old program with new subroutines	Could be done, but not likely. Probable method is to use assembly language compatibility (below).
		Convert old programs to get new capabilities.	Conversion cannot be done automatically.
		write new programs that can be assembled for both machines.	Can be done with conventions and smart assemblers. Useful for limited areas, e.g., FORTRAN SIN, COS, etc. routines. In general, a program that will run on the old machine cannot make use of features of new one, unless we write emulator of new machine on old one (not likely).
	Operating Systems	Run old operating system unchanged	Can't be done; no KT compatibility.
		Convert old operating system to run with no new capabilities.	Not worth using resources just to get back to where we are now.

		Convert/ rewrite/ start-all-over-again operating system to provide new and old capabilities	This will provide the proper environment to run old user programs.
Compatibility between different members of 32 bit family	User pro-	Run any program or any model	will be done, within reasonable limits, e.g., programs that require 16 million bytes of address space will not run on LSI-32 with floppies. (Would take 60.) Some instructions will be emulated (64 bit arithmetic) on low end.
	Operating System	Run any OS on any model	OS will require a well defined, fixed environment provided by a combination of hardware and small (less than 10K bytes) software kernel. Again, reasonable limits will be imposed.

SR:mjk [5/27/75, HIENIM.SR1]

SUBJ: THE TECHNOLOGY AND SYSTEM ENVIRONMENT 1975-1985

The purpose herein is to provide the technology/price background environment for the 11VAX time scale, a proposed "new architecture" based on the 11. This summarizes the memo: Projected Prices of Main Memories as a Means of Modelling System Prices (Bell, March 24, 1975) and C.performance 11VAX Models (Bell April 8, 1975).

What are the implications on 11VAX given the technology environment 1975-1985?

1. People costs will dominate.
   People costs (at price of $60/hour in 1980) will really dominate more than they do now, almost any feature that saves time will have a tremendous payoff. For example, we can't afford to sell low level activities like binary patching, although the customer may be able to afford them if he doesn't account time very well.

2. We all have to move to high level languages both to increase productivity and to increase reliability (e.g. checking of data-types, array bounds).

3. The networks aspect will be important.
   The user can move jobs among machines as a function of cost and performance. Note, he can't afford to have people carry data, programs, etc. Distributing the problem spatially, could reduce the interactions among the parts, and drastically reduce the solution time.

4. Standards will be even more important.
   Users can't afford to rewrite for each system, except in very high volume, highly specialized case. Learning a system has to be easier and cheaper.

5. The range is important.
   He pays his money and takes his choice...at as late a time as possible. User's don't want to feel bound, but feel we'll have various alternative solutions to their changing problem.

6. Since we are attempting a larger range, and the range is proportional to memory size, then we must be able to express the range in the address.

7. Our traditional $50K-100K is in jeopardy as a cost-effective system. Although we can supply larger memories, we give only limited access to it.

8. Programming and Data security are becoming significant.

Technology of Specific Computer Components

The technological growth of a computer can be broken into its constituent parts. Therefore, various sized machines will evolve differently depending on the constituent parts. If we look at various parts, we find that some evolve very rapidly and some very slowly. 1965 Armour, (Turn 1974) projected a decline of 80% per year for CPU storage cost per mega word accessed, permit time over a 15 year time scale; whereas his projection for the typewriter cost per mega character printed was roughly constant.

For disks, the price of a given physical structure, e.g. a disk pack with 10 platters, actually increases slightly with time; but the magnetic storage density drastically increases. This is to be expected, since a complete disk mechanism is built up of components which have been "learned"--i.e. sheet metal, motors, power supply, etc. IBM's disk packs with 10 platters (beginning with the 1311) have increased in capacity at the rate of 42% per year, and the price per bit has decreased 38% per year.

In Table 1, different styles of disks are presented (very roughly) in terms of 1975 prices. There are 4 styles of disk packs beginning with the flexible disk and going to the 10 platter disk pack.

Here we can see an economy of scale, i.e., the number of bits for larger units increases more rapidly than cost giving a decreased cost per bit. The overhead of motors and packaging increases less rapidly with size. Also, more effort is placed to increase the densities at the large sizes. This technology is used on the smaller disk pack after a two year delay.

Conservatively, semiconductor technology has improved at the rate of 60 to 80% per year, as measured by cost and/or cost-performance. In fact, semiconductor memory densities have doubled each year, beginning in 1962 with a single bit.

The various technology improvement rates are given in Table 2. We can look at these characteristics in terms of the net effect on minicomputer systems. Minis have declined in price at about 31% per year. Beginning in 1972, the 8 bit micro-computer was introduced, and the name is now synonomous with an 8-bit comfputer and a processor-on-a-chip (1 single silicon semiconductor chip). These machines are parallel to the 12 bit and 16 bit in price decline, although they have significantly less performance.

Memory Cost and Price

Memory has declined at least 26%/year during the previous 10 years, and is (conservatively) likely to decline at least this fast during the next 10 years. Thus each 10 years prices decline by a factor of 10! MOS is pushing core, and there are secondary memory technologies such as magnetic domain and charge coupled device memories which will help drive the costs at faster rates. In 1972, the cost was $.005/bit, and therefore:

Cost/bit = $.005/1.26^(t-1972).

Assuming only 3X markup:
Price/bit = $.015/1.26^(t-1972).

Note, this checks with the Telex papers rumor of $1K/1Megabyte rental for IBM; and it checks with HP's current MOS prices of $800 for 16K bytes.

Memory Price Determsines System Price

In order to effectively balance and use a computer, each configuration of a given memory size and processor speed can effectively support a certain amount of peripherals--particularly secondary memory. For this model, I assume the system price is:

System-price = 5* Primary Memory Price

therefore,

System-price = $(.075/1.26^(t-1972))*number of bits

Memory Size Determines System Function (total use)

I believe in the following statement (rule):

memory size, to a first approximation, determines the total functionality (use) of a system. Through memory size, one can most likely (and easily) classify a machine's structure and what it will be used for.

Classification of all systems by Dedicated versus Programmable; number of users, and function (use)

The Function (use) we usually subscribe to a system is:

1. Interactive--connected to people (e.g. POS, text manipulation, transaction processing, information retrieval, program development).

2. Real time--connected to a mechanical process which demands certain response.

   A. Control--discrete and continuous
   B. Switching--i.e. communications.
   C. Data collection and processing--i.e. laboratory and some industrial.

3. Hybrid--a mixture of two above.

4. Interactive, can also be looked at as real time without deadline scheduling. Batch is a further reduction in deadline scheduling. This function (use) category has very little effect on the memory size!

Each system has a degree of generality depending on whether it is fixed or dedicated to an application.

1. Dedicated--system is well bounded by size at purchase time

we sell to OEM's are all dedicated, hence, have up to a factor of 2 less memory for the same task that a programmable system might have.

2. programmable (general purpose)--large at purchase time and over life time of a system, since the final use/application varies over the machine's lifetime.

The final dimension which affects size is the degree of multiprogramming. This has a drastic affect on memory size, in part, because an operating system is required to keep track of the machine resources. In the case of multiprogrammed systems which are programmable, an even larger operating system is required to protect the users from one another and to permit sharing.

1. Single user.
2. n-users.

This leads to 4 cases which affect size: dedicated 1 user, programmable n-user. There can be hybrid cases like foreground/background, which in effect is 1 user dedicated + 1 or n users dedicated to another common task, these reduce memory requirements by eliminating generality.

Table 3 illustrates the System Structures, memory size, and Function (use).

Figure 1 plots the memory price and system price versus time for various memory sizes (which determine the various system functions).

Figure 2 plots the cost versus data rate for various busses and I/O devices.

Correlating processor Performance With Memory Size

Amdahl (formerly with IBM) has several "rules-of-thumb" which I think we can use as guidelines for relating processor performance, memory size and I/O bandwidth:

1. For each instruction/SEC the processor executes, 1 byte of memory is required. For our more scientifically and control-oriented processing, I would believe that:

   1 inst/sec implies 0.5 to 1 byte of memory

2. 1 bit of I/O data transmission occurs for each instruction executed. Again, this may be low for our more real time orientation. Until there is a great deal of I/O per instruction (i.e. more than 1 byte per instruction), the system configurations are relatively insensitive to I/O.

   A. Each 11 instruction is 3-5 bytes.
   B. 1 instruction generates 1 byte (much worse case) of I/O traffic.
   C. Total memory bandwidth requirements are 4-6 bytes/inst/sec.

In our larger configurations, the cache structure can be used to reduce memory-processor traffic.

Various alternatives are given in the forward.

POSSIBLE 11VAX MODELS BASED ON OUR BUSSES

STRUCTURE	Pc	BUS (Mips)*	Pc. Perf (Mips)	Mp.size (K bytes)	Mp size Implied Pc.Perf**	System Price (1980 K$)
LSI-11	1-WD	.3	.05-.1	8	.008-.016	.8
LSI-11	1-WD	.3	.05-.1	65	.06-.13	6.5
UNIBUS	?	.4	.2	32	.03-.06	3.2
UNIBUS	?	.4	.4	256	.25-.5	25.6
85 BUS	1 Pc	2	1.0	1000	1-2.	100.
85 BUS	4 Pc	2	4.0	4000	4-8.	400.

*Assumes no cahce. A cache lowers bus access requirements by a factor of 10.

**The memory size is used to impute the Pc performance (in Mips) assuming .5 to 1 byte requires 1 inst/sec.

Note, that this gives a memory size range of 512, and there is still bus and memory size capacity to get to 8Mby, although the number of devices attached to the bus would be a limiting factor.

The low end performance is overstated, since a general job mix will do better on a larger Pc, which has proportionally faster instructions.

The model doesn't go according to Grosch's law, (i.e. performance=K x Cost^2) since the model assumes cost scaling with Mp.size. This is also probably not true, and the very high end will cost less per byte. Traditionally minis really haven't performed according to Grosch's law--otherwise, we'd not exist.

Small Decentralized Versus Large Central Systems

We can contrast the minicomputer with the large machine (Turn, 1974) and observe that technology improvement goes to increase performance. However, in the case of several recent large machines, the machines are yet to be operational within 5 years of their scheduled time, therefore, it is hard to characterize them as actual points on technology curves. Indeed, they may never exist as entities as specified. The small machine, a PDP-8, and the large CDC 6600 can be compared as of 1967 (Table 4). A possible large machine can be compared with a small, board-only, processor-on-a-chip computer costing $1000 for 1975 technology (also Table 4).

Actually, it is perhaps more important to put machine cost into perspective as to other costs of use. (See Table 5.) Note, that the dominate costs are the human, user costs.

Figure 3 shows what is likely to be the future structure of computing. Because of the decline of machine prices, we would expect substantially more stand-alone computers (e.g. the DEC CLASSIC system), which are capable of running any language; in essence a generalization over the HP and WANG BASIC-only language machines. There is, in fact, a strong affinity to have these machines totally stand alone and not require external services for their support. We would, however, expect all of these machines to require some support for sharing certain common facilities (e.g. archival storage, classroom programs, printing, plotting) through a switching structure (including closed circuit video) to other systems. At the other extreme, we would expect large general purpose systems to prevail, which are predicated on an economy of scale. These would be capable of exploring varied application spaces where generality is needed. Thus, we would see at the two ends of the cost spectrum: a continued use of larger, general purpose computers; and totally dedicated low cost systems would exist that have been tuned to specific applications. Table 6 contrasts the two designs: small decentralized systems versus the large general centralized system.

It can be conjectured what eventual systems like this would sell for and how they operate. The stand-alone systems, e.g. the DEC CLASSIC, and the specialized BASIC-only language terminals, now sell for about $8000 or $200 per month. This price is only a factor of 3 to 8 more expensive than a dumb terminal capable of only accessing a more complex large central system. The total economies are a function of the expected use and the amount of support required in the two systems. As Selwyn, at MIT, has observed, there is a very large economy of scale associated with centralization. The total cost to operate a computer is proportional to the monthly rental raised to the .8 power. But in the very small totally decentralized case, we would not expect this to hold. However, there are clearly hidden costs associated with individual computers. The exact mix is not clear, but both will exist.

The new exciting areas of application for computers is clearly in the small systems and the coupling with other machines.

Some Implications to other DEC Machines

0. It really doesn't matter, from a technical viewpoint, which machine we select, all will use memory at about the same rate to express an algorithm.

1. The higher end systems will all migrate to substantially do the same thing...n-user dedicated and general (programmable). The 10 will be drawn into the mini fray, as minis get larger memories and do the same function (i.e. gPts).

2. PDQ is severely memory size limited, so is Eclipse. In 1980, a PDQ system will be limited to $25K unless the memory address space is expanded. The 11/70 is well-balanced, good for multiprogramming provided the address space for a single job is not a limit.

3. The LSI-11 address space will limit it by 1980.

   People costs dominate, so the smallest number of programmable machines is clearly the most profitable (i.e. we don't replicate function).

4. The 11/85 bus looks nice from a data-rate viewpoint. It will get into a "can't have enough devices on it" problem perhaps, but there are several ways out:

   A. Serial bus for all terminals and unit record devices.

   B. Interconnected busses--the problem is solved unlike the UNIBUS window which is inherently unreliable.

   C. A UNIBUS system can connect to it and handle conventional I/O.

GB:mjk [6/4/75 Version 2, PI4.V01]

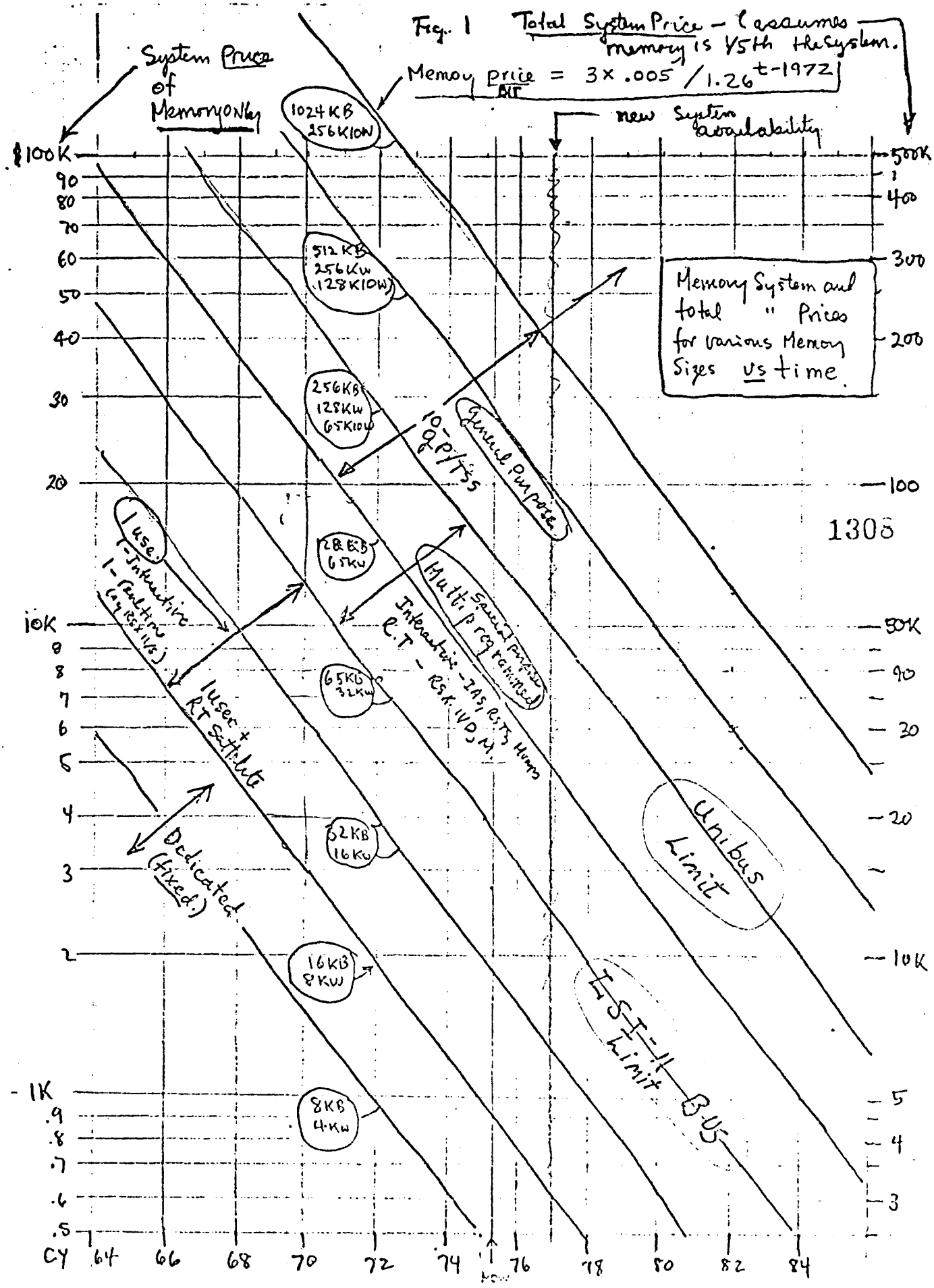


Fig. 1 Total System Price – (assumes memory is 1/5th the system.
Memory price = 3 × .005 / 1.26^(t-1972)
$/BIT
System Price of Memory Only
new System availability
1024 KB 256K10W
512 KB 256 KW 128K10W
256KB 128KW 64K10W
128KB 64KW
64KB 32KW
32KB 16KW
16KB 8KW
8KB 4KW
10-qp/t's
General Purpose
Multi Prog Optimized
Special purpose
Interactive - IAS, RSX, Unix C.T - RSX-VD, M.
1 use. (T-Interactive 1-real time (eg RSX IVS))
1 use + RT Settling
Dedicated (fixed)
Unibus Limit
PDP-11 limit BUS
Memory System and total " Prices for various Memory Sizes vs time
1308
$100K 90 80 70 60 50 40 30 20
10K 9 8 7 6 5 4 3 2
1K .9 .8 .7 .6 .5
500K 400 300 200 100
50K 40 30 20 10K 5 4 3
CY 64 66 68 70 72 74 76 78 80 82 84

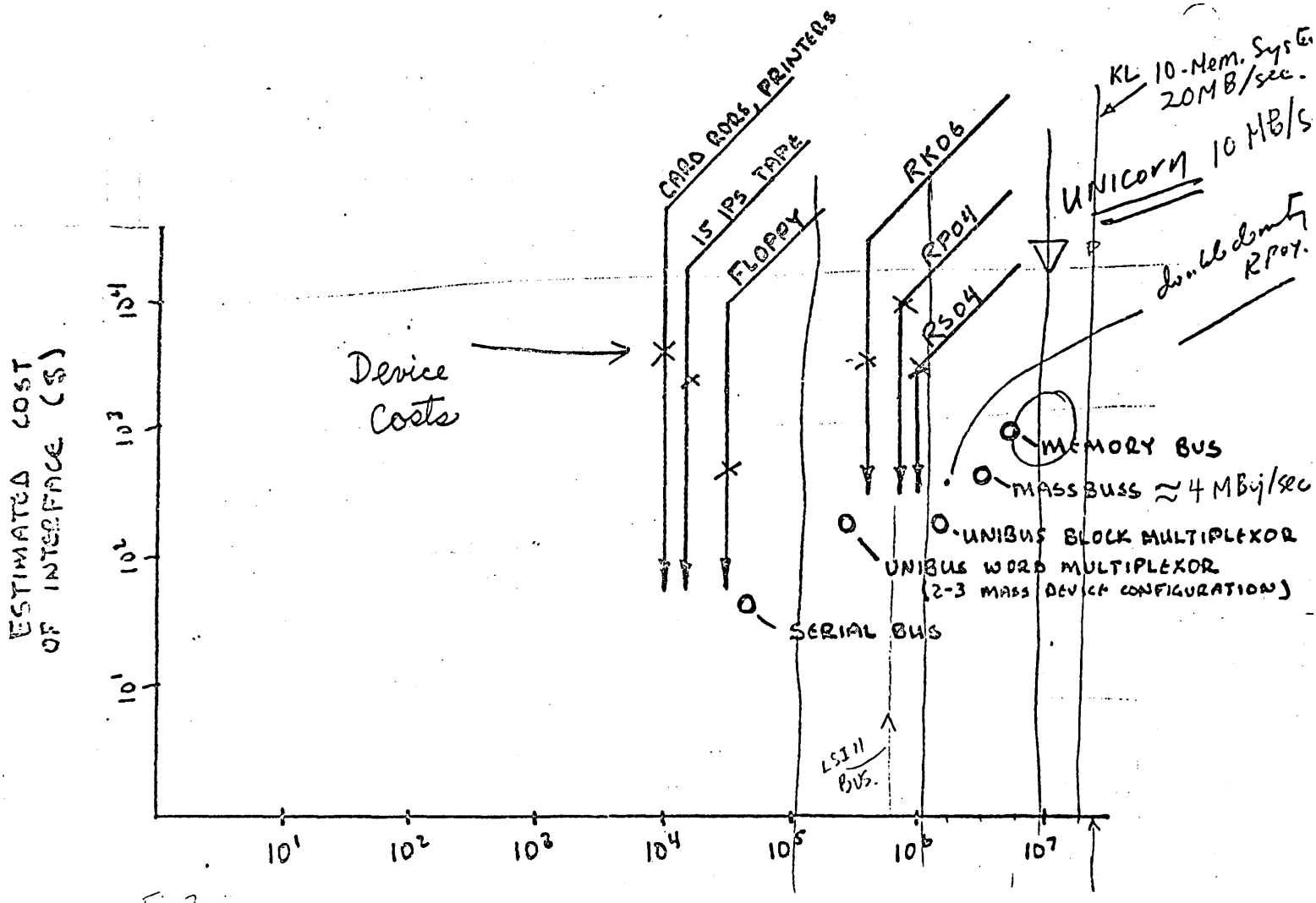


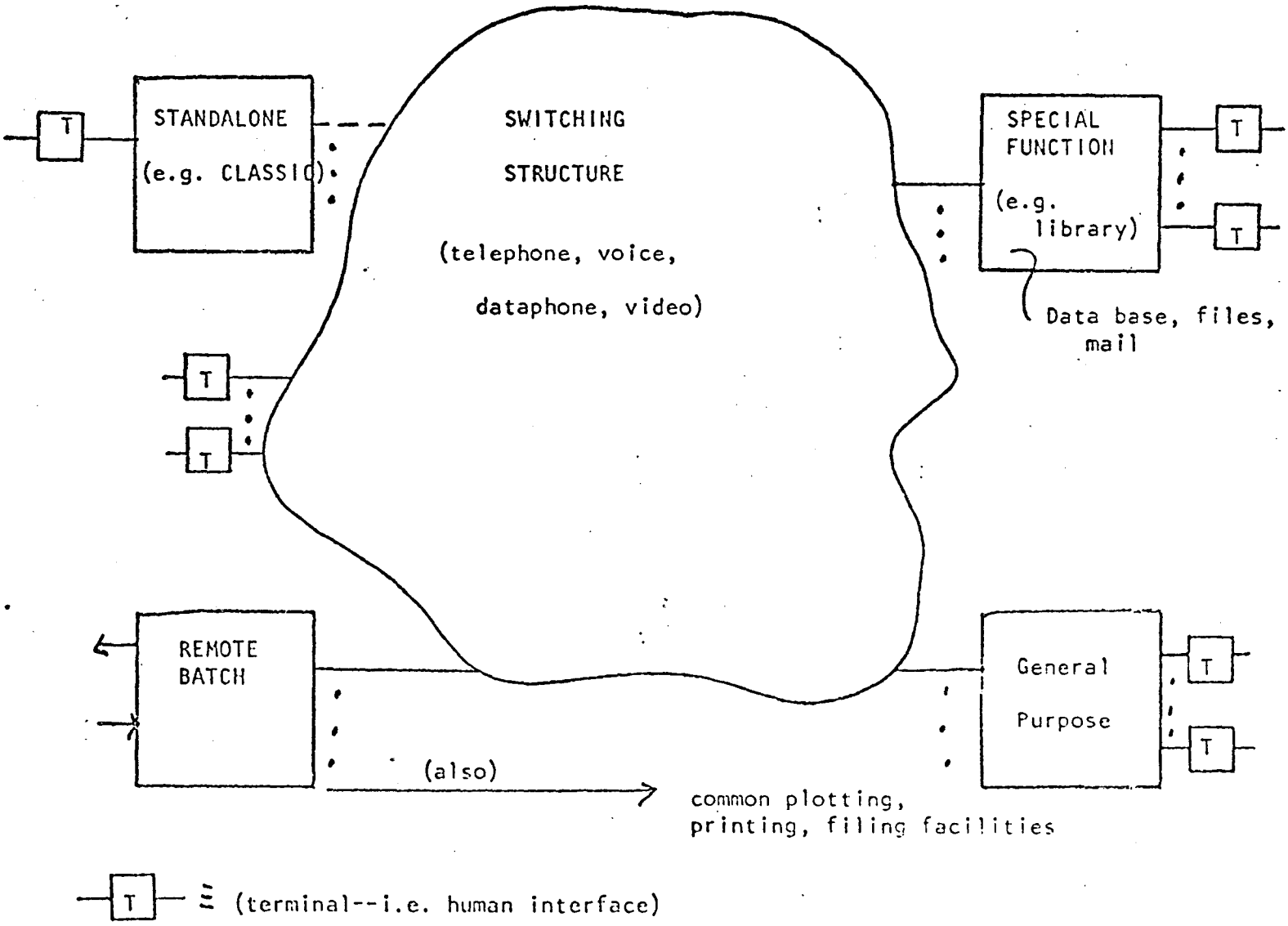
Fig 2.

Fig 2

D NELSON

1303

FIGURE 3 GENERAL STRUCTURE OF COMPUTATION FACILITIES



GB
5/8/75

TABLE 1 DISK TECHNOLOGY (1975)

DISK CONFIGURATION	PRICE (Roughly)	CAPACITY (Bits)	COST/ BIT	ACCESS TIME
Flexible (floppy)	3	2.5	1.2	1 s
1 Platter	6	30.0	.2	50 ms
3 - 5 Platters	12	160.0	.075	50 ms
10 Platters	24	800.0	.03	20 ms

GB
5/8/75

TABLE 2 YEARLY IMPROVEMENTS FOR VARIOUS COMPUTER TECHNOLOGY COMPONENTS

Technology Indicator	% Improvement
Semiconductor density	60 - 80
Semiconductor memory density (bits/chip--leading edge)	100 (1962 - 1974)
Magnetic recording density (disks)	41 (1962 - 1975)
Core (price)	30
Terminals	25
Magnetic tape (density) (data-rate)	23 (1952 - 1973) 29
Power Supply (cost/watt)	-3
Packaging (cost/in↑3)	-3
Minicomputers	31 (1960-1975)

GB
5/8/75

Table 3--SYSTEM STRUCTURE, MEMORY SIZE, AND RESULTANT USE

Structure	Memory Range	Function (use)	
Dedicated (fixed-] use)	16KB (4KB - 8KB)	Unteractive--e.g. POS Real time--e.g. scope, traffic control, automobile	Special purpose, Fixed
Programmable (1 user)	16KB - 65KB	Interactive--RT11 Real time--RSX11S,M	Small scale, generality
Dedicated (multiprogrammed n-users)	65KB - 256KB	Interactive--MUMPS, Trans. Process n RSTS Real time--RSX-11M, D	Special purpose
Programmable (multiprogrammed n-users)	128KB - 1024KB	Interactive--IAS, TOPS 10, RSTS Real time--RSX-11D	Generality

There are many implications of the 4 categories of structure and the 2-sub categories as to the operating system, its overall system structure, etc.

GB
3/24/75

TABLE 4 LARGE AND SMALL COMPUTERS 1967 AND 1975

Circa 1967	COST	WL	Mp Size	Bits/$	MIPS	MIPS X WL	REAL (MIPS)	P/C (MIPS/M$)
8	10^4(1)	12 (1)	$5X10^4$(1)	5	$.3X10^6$(1)	(1)	10^3(1)	30
6600	$3X10^6$(300)	60 (5)	$8X10^6$(160)	2 2/3	$3X10^6$(10)	(50)	$3X10^6$(3000)	1
Circa 1975								
BRD	10^3(1)	16 (1)	$4X10^3$w(1)	64	.3 (1)			100
Large	10^7(10^4)	64 (4)	$1X10^6$w(10^3)	6.4	100 (10^4)			10

Some Observations:

1. Performance: small is about the same. Large up 10~30.
2. Cost: small is cheaper by X10. Large up X3.
3. Mp. size no economy of scale.

GB
5/3/75

TABLE 5 OPERATING COSTS FOR COMPUTING

(1K) COST/Year		COST/HR. @ 2400 HR.	
Human	0, 5, 10, 20, 40	0, 2, 4, 8, 16	
Computer	1.2 ~ 2.5	.5 ~ 1.	
Terminal	.25 ~ .75	.1 ~ .4	
Service	.05	.02	
Power	.005 ~ .01	.002 ~ .004	
Line	0 ~ 2.4	0 ~ 2.	
Paper	0 ~ .1	1/3¢ ~ 3¢	
Space	.05 ~ .1	.02 ~ .04	

GB
5/8/75

TABLE 6 CHARACTERISTIC DIFFERENCES OF SMALL (decentralized) VERSUS LARGE (central) FOR COMPUTATION

Attribute	Small (Decentral)	Large (Central)
Performance	Greater average.	Greater peak Large memory (programs)
Cost	Economies through production. Lower, given a terminal is is needed for a system. f(COMM line, terminal, and memory costs).	Economy of scale for disk.
	Production limited.	Design limited.
	Overhead of maintenance on individual (hidden)	Explicit, central maintenance costs.
Use	Small (or 0) data bases.	Large, shared data base.
	Fixed, (well-defined) computation.	General (undefined) computation.
Non-use	Programming where tasks are unbounded.	Defined, bounded tasks.
Security	Private	Public

GB
5/8/75

SUBJ: VAX SOFTWARE DEVELOPMENT PLAN

This plan documents both the short and long term system software plans in support of the 11VAX. The budget for FY76 fits within the overall Central Engineering plan, assuming that the UNICORN money becomes available.

WHAT SOFTWARE IS PROVIDED FOR FIRST CUSTOMER SHIPMENT?

Since the first VAX machine will be a high-end, high-performance system, we will provide an RSX-11D-like system. The customer will see a product very similar to RSX-11D, it will have all the same capabilities, will execute all user programs with binary compatibility, and will offer users the ability to utilize extended addressing.

This system will be produced by converting either RSX-11M or RSX-11D. The decision as to which one will be made by the end of August, following cost estimates and based on resource availability. For planning purposes, this plan assumes that RSX-11D will be the basis in order to emphasize the high-end orientation.

FORTRAN IV-PLUS will be the only FORTRAN offered. The compiler will run in compatibility mode since it requires <32K of memory; but the code generator will be rewritten to exploit the new instruction set.

An implementation language will be offered for first customer shipments. In addition an assembler will be produced.

Existing utilities, languages and applications will be offered in compatibility mode, particularly COBOL. The new BASIC-PLUS will be available (the product will be funded directly by the Product Lines).

WHAT COMPATIBILITY WILL EXIST WITH THE CURRENT SOFTWARE?

Software compatibility is a strong overall VAX goal, and binary compatibility at the user program level will be provided. Application and system programs will execute with no modifications, but they will not use any of the new instructions. Examples include COBOL, MACRO, FILEX, COGO, etc.

Operating Systems which do not reference memory management (such as RT-11 and RSX-11M/non mapped) will operate with only local changes. Systems which exploit the current KT11 (RSTS/E, RSX-11D) will require substantial changes to use the new memory management. In addition any system which supports extended addressing will require a total conversion because of the new instruction set provided. It is not our intention to convert the current RSTS/E V6.

The compatibility path for current RSTS users will be through a three step process.

1. Minor enhancements to RSTS/E V6 will continue for 18 months.

2. Immediate development of TPM-16 targeted for 18-24 month completion with a 2-3 year life on the 16 bit products (more detail elsewhere in this plan).

3. Immediately begin to develop an operating system designed for the VAX machine, with a 24-30 month target, but not committed for FCS. This system will provide the ultimate operating system for VAX and be the upgrade path for TPM-16.

Our plan is to convert RSX-11D (possibly RSX-11M) and to provide two user environments, one which allows existing binary programs to execute and the other which allows users to create programs which use extended addressing. Therefore, existing RSX-11D applications will run with no modifications, unless they use priviledged tasks.

The same FORTRAN-IV PLUS "language" will be provided on both 16 and 32 bit systems, making user's programs interchangeable except for large arrays.

WHAT OPERATING SYSTEM WILL BE DESIGNED SPECIFICALLY FOR THE HARDWARE?

A kernel operating system will be produced, starting immediately, in parallel and in cooperation with the hardware design. This kernel will provide the basis for both real time and commercial operating system products. It will offer the fundamental primitives required for any system and will be designed to use and exploit the VAX architecture.

The kernel by itself will not be a "system", but it will be the foundation for real time and transaction processing systems. It is very analogous to the LSI-11 chip set, namely, high technology, a key system component, but not a standalone product by itself.

WHEN WILL THIS KERNEL BE AVAILABLE?

The kernel itself will not ship. It will be available for internal use in a "prototype" form by first customer shipment. The specifications will be firm 3-6 months prior to this and will be used as the basis for internal system developments.

Our goal is to base all of our systems on this kernel, particularly for real time and transaction processing. TPM-32 will be based on this kernel and will be delivered by mid FY78. The mid range real time system will also be offered by mid FY78 to coincide with the availability of the mid range hardware.

WHAT OTHER UNIQUE SOFTWARE WILL BE PRODUCED?

After the first FORTRAN IV-PLUS release, we can expect the compiler itself to be rewritten to exploit the hardware. Currently it is a batch oriented compiler which cannot be shared in a multi-user environment. It will be rewritten to be sharable.

The current COBOL system will begin modifications in FY76 to support VAX. In particular the code generator will be rewritten

to produce machine code. This will make it possible to produce a unique VAX code generator, but not in time for first shipments.

As noted previously, an implementation language will be produced. It will produce optimized code for VAX as well as contain specific language elements in support of the architecture (CALL statement, ring structure).

HOW DOES RSTS/E V7 FIT IN?

We will build an evolutionary product from RSTS/E to be known as TPM-16. It is called "TPM" to emphasize the transaction processing orientation. It is not a one-for-one replacement for RSTS/E, since it will be first a transaction processing (application-oriented; runtime-oriented) product and secondly a time sharing product. The orientation of the current RSTS/E is towards time sharing. This system will be designed specifically for 16 bit computers, will be built with FILES-11 and will contain the RSX-11M/D internal monitor interface. Its human engineering will be as polished as RSTS/E. The new BASIC-PLUS will be offered (also offered on the real time products).

Due tothe new file system and the improvements in the new BASIC-PLUS, existing RSTS/E user will have to convert and upgrade to this product.

WHAT ARE THE UPGRADE/MIGRATION PATHS FOR EXISTING USERS?

Real Time

At first customer shipments, real time customers will have the same family capabilities as they have today, except the high-end system will be based on the 32 bit system. RSX-11S and RSX-11M will continue to be offered on 16 bit processors.

About mid FY78, the new real time system will be available, based on the Kernel. This system will be available on small, medium and large VAX machines, as the hardware becomes available. A small 16 bit version of RSX-11S will still be available on LSI-11 style machines. The real time family of products will be VAX-based providing a broad upgrade path for existing family users.

Time Sharing

RSTS/E customers can upgrade either to the DECsystem-20, or to TPM-16 or to TPM-32. BASIC-PLUS will be the primary tool to encourage upgrades.

WHAT IS THE FY76 BUDGET?

Product	FY76	FY77 (estimate)
TPM-16	260K	200K
RSX-11D Conversion	150K	150K
New Kernel	150K	600K
FORTRAN IV-PLUS	150K	150K
Implementation Lang.	110K	100K

New BASIC-PLUS	200K	200K
COBOL	--	200K

HOW HAS THE "RED BOOK" STRATEGY BEEN MODIFIED TO ACCOMODATE THIS BUDGET?

When the Red Book was issued in March, there were no explicit VAX software funds identified. The current budget reflects the following changes:

$150K from Real Time to cover RSX-11D conversion to VAX
$150K from Languages to cover FORTRAN IV-PLUS
$260K of the $600K for UNICORN is appropriated for
    new kernel - $150K
    implementation language - $110K

TPM-16 was funded as part of the original strategy. $100K of the FILES-11 money is allocated for the TPM-16 file system.

Business Products has agreed to directly fund the new BASIC-PLUS.

WD:mjk [6/5/75, SOFPLN.WAD]

TECHNICAL SUMMARY

digital

Contents

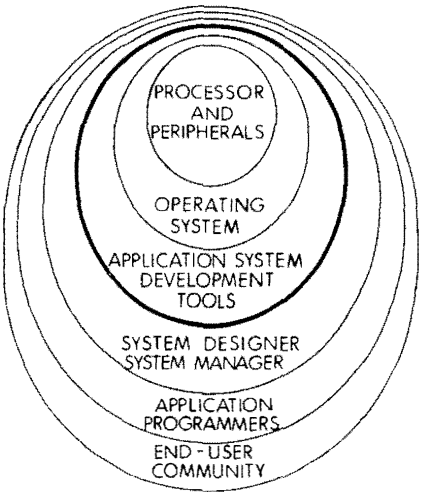
1
Introduction

The VAX Technical Summary introduces the characteristic features and capabilities of the VAX system to computer analysts and system programmers. Application programmers, and system managers and operators may also use this summary as a tool to become familiar with the components, services, and operations of the VAX system.

The intent of this summary is to familiarize the reader with the features and capabilities of the VAX (Virtual Address Extension) system. It serves as a detailed technical introduction to the growing family of VAX processors, the powerful and unique Virtual Memory operating System, VAX/VMS, and the increasing number of supported peripherals. In particular, this edition of the VAX Technical Summary introduces several significant system enhancements:

- the newest member of the VAX family of processors, the VAX-11/750
- version 2.0 of the VAX/VMS operating system
- the addition of major new languages including VAX-11 FORTRAN, VAX-11 COBOL, VAX-11 BASIC, VAX-11 PL/I, VAX-11 PASCAL, VAX-11 CORAL 66, and VAX-11 BLISS-32

Although the VAX Technical Summary contains useful information for the application programmer, the system manager, and the computer operator, the level of technical detail makes the book particularly appropriate for the system programmer and/or computer system analyst.



As the reader proceeds through the technical summary, the term "VAX architecture" or simply "architecture" may seem confusing. VAX architecture is the collection of attributes that all family members have in common that assures software compatibility. For example, the architecture includes the instruction set, the addressing modes, data types, etc. Examples of attributes not included in the architecture are processor internal bus structure, implementation-specific privileged registers, execution speed, etc. As new processors are added to the VAX family, a significant part of the engineering effort will be dedicated to preserving this software compatibility. This will assure that programs written for today's VAX computers will execute on future VAX systems.

The VAX Technical Summary is designed to be read in either a selective or sequential manner. The reader might start with section 1 and continue through the book, or first glance through the table of contents to locate those topics of most interest. As an added convenience, an abstract can be found prefacing each section of the summary. Many of the system's concepts and features are repeated throughout the text in various contexts. Appendix A contains a collection of most frequently used mnemonics and their definitions.

The following paragraphs introduce the VAX Technical Summary.

The System section presents a comprehensive overview of VAX system features. This section serves as an introduction for the reader presently familiar with computer industry terminology, identifying VAX characteristics and introducing the system features described in detail throughout the remainder of the summary.

The Users section contains a description of the command language and its features. It also introduces many of the aspects of the system that support applications programming, system management, and operator control.

The VAX Family of Processors and Operating System sections provide an in-depth discussion of the system's characteristics and capabilities. The concepts developed in both sections are closely related; for example, the respective discussions of memory management, I/O processing, and the compatibility mode environment should be read together to gain a full appreciation for the system's design. These sections should prove beneficial to systems programmers or analysts already familiar with an assembly language or software executive.

Perhaps one of the most important features of the VAX system is that its programmers do not have to know assembly language to use the system effectively. Both the hardware and software contain many features that promote efficient and productive high-level language programming. High-level language programmers should find the Users, Languages, Data Management Services, and Network Services sections, of particular interest. The beginning of the Operating System section is also helpful because it introduces some of the VAX software terminology and concepts.

Also of interest to high level language programmers is the Program Development section which describes the basics of creating, editing, debugging, and executing a program written in any of the VAX high level languages. This section also contains an introduction to some of the advanced features of the command language.

If the reader is uncertain about a particular term or phrase, its definition can probably be found in the glossary at the end of the summary. The glossary does not generally contain standard computer-related terms, but it does contain most of the terms found throughout the VAX documentation that have special meanings in the context of the VAX system. The glossary is followed by a list of mnemonics that may be encountered in the text. The mnemonics list is particularly useful during the system familiarization stage.

For additional literature describing VAX features, capabilities and applications please contact the nearest DIGITAL sales office.

2
The System

The VAX system provides the performance, reliability, and programming features often found only in much larger systems. The VAX family of processors have a 32-bit architecture based on the PDP-11 family of 16-bit minicomputers. While using addressing modes and stack structures similar to those of the PDP-11, VAX provides 32-bit addressing for a 4 gigabyte program address space, and 32-bit arithmetic and data paths for processing speed and accuracy.

The processor's variable length instruction set and variety of data types, including decimal and character string, promote high bit efficiency. The processor hardware and instruction set specifically implement many high-level language constructs and operating system functions.

VAX is a multiuser system for both program development and application system execution. It is a priority-scheduled, event-driven system: the assigned priority and activities of a process in the system determine the level of service needed. Real-time processes receive service according to their priority and ability to execute, while the system manages CPU time and memory residency allocation for normal executing processes.

VAX is a highly reliable system. Built-in protection mechanisms in both the hardware and software ensure data integrity and system availability. On-line diagnostics and error detecting and logging verify system integrity. Many hardware and software features provide rapid diagnosis and automatic recovery should power, hardware, or software fail.

The system is both flexible and extendible. The virtual memory operating system enables the programmer to write large programs that can execute in both small and large memory configurations without requiring the programmer to define overlays or later modify the program to take advantage of additional memory. The command language enables users to modify or extend their command repertoire easily, and allows applications to present their own command interface to users.

INTRODUCTION

VAX is a high performance multiprogramming computer system ideally suited for a wide variety of applications including real time, batch, time sharing, commercial, data processing, and program development. The system combines a 32-bit architecture, efficient memory management, and a virtual memory operating system to provide essentially unlimited program address space.

The processor's instruction set includes floating point, packed decimal arithmetic, and character string instructions. Many of the instructions are direct counterparts for high-level language statements. The software system supports programming languages that take advantage of these instructions to produce extremely efficient code.

The VAX/VMS virtual memory operating system provides a multiuser, multilanguage programming environment on the VAX hardware. The floating point instructions, efficient scheduler, and optional VAX-11 FORTRAN language are ideal for real-time and scientific computational environments. The optional VAX-11 COBOL language, data management services, and large capacity peripherals make the system equally suited to commercial applications. VAX/VMS supports many other optional high level languages suited for other applications. The system management facilities, command language, and program development tools provide the resources for efficient program applications development and execution. Spooling and extensive job control capabilities support batch processing.

The processor executes variable length instructions in native mode, and non-privileged PDP-11 instructions in compatibility mode. Native mode includes the PDP-11 data types, and uses addressing modes and instructions similar to those of the PDP-11. The software supports compatible languages and file and record formats.

COMPONENTS

The major components of a VAX system are:

- Processor—includes the basic central processing unit implementing the VAX architecture. The specific implementations of the VAX processors will be treated in greater detail in Section 4, The VAX Processors.
- Operating System—includes a virtual memory manager, swapper, system services, device drivers, file system, record management services, command language, and operator's and system manager's tools.
- Languages—includes the native mode languages VAX-11 MACRO and optionally, VAX-11 FORTRAN, VAX-11 COBOL, VAX-11 BASIC, VAX-11 PL/I, VAX-11 PASCAL, VAX-11 BLISS-32, and VAX-11 CORAL 66. Also supported in compatibility mode are PDP-11 BASIC-PLUS-2/VAX, PDP-11 FORTRAN IV/VAX to RSX, and MACRO-11. Development tools for both native and compatibility mode programs include editors, linkers, librarians, and debuggers.
- Peripherals—includes a range of small- and large-capacity disk drives, magnetic tape systems, hard copy and video terminals, line printers, card readers, and real-time I/O devices.
- Network Services—includes the DECnet-VAX network software and the DMC11 interprocessor communications link.

Processor

Architecturally, a VAX processor provides 32-bit addressing, sixteen 32-bit general registers, and 32 interrupt priority levels. The instruction set operates on integer, floating point, character and packed decimal strings, and bit fields. The instruction set supports nine addressing modes.

The processor includes an efficient memory cache resulting in greatly reduced memory access time.

Error Correcting Code (ECC) MOS memory is connected to the memory interconnect via a memory controller. Each memory controller includes a request buffer that substantially increases overall system throughput and eliminates the need for interleaving in most applications.

The processor uses two standard clocks—a programmable real-time clock used by the operating system and by diagnostics, and a time-of-year clock used for system operations. The time-of-year clock includes battery backup for automatic system restart operations.

The console is the operator's interface to the central processor. Via the console terminal, the operator can execute diagnostics, install new software, examine and deposit data in memory locations or processor registers, halt the processor, step through instruction streams, and boot the operating system.

Virtual Memory Operating System

VAX/VMS is a multiuser, multifunction virtual memory operating system that supports multiple languages, an easy to use interactive command interface, and program development tools. The VAX/VMS operating system is designed for many applications, including scientific/real-time, computational, data processing, transaction processing, and batch.

The operating system performs process-oriented paging, which allows execution of programs that may be larger than the physical memory allocated to them. Paging is handled automatically by the system, freeing the user from any need to structure the program. In the VAX/VMS operating system, a process pages only against itself—thus individual processes cannot significantly degrade the performance of other processes.

The memory management facilities provided by the operating system can be controlled by the user. Any program can prevent pages from being paged out of its working set. With sufficient privilege, it can prevent the entire working set from being swapped out, to optimize program performance for real-time or interactive environments. Sharing and protection are provided for individual 512-byte pages. The processor's memory management includes four hierarchical processor access modes that are used by the operating system to provide read/write page protection between user software and system software. The access modes from most to least privileged are kernel, executive, supervisor, and user.

VAX/VMS schedules CPU time and memory residency on a pre-emptive priority basis. Thus, real-time processes do

not have to compete with lower priority processes for scheduling services. Scheduling rotates among processes of the same priority. The scheduler adjusts the priorities of processes assigned to one of the low 16 priorities to improve responsiveness and to overlap I/O and computation. Real-time processes can be placed in one of the top 16 scheduling priorities, in which case the scheduler does not alter their priority. Their priorities can be altered by the system manager or an appropriately privileged user.

Interprocess communication is provided through shared files and shared address space, event flags, and mailboxes which are record-oriented virtual devices.

VAX/VMS provides system management facilities. A system manager and a system operator are given the tools necessary to control the system configuration and the operations of system users for maximum efficiency.

The VAX/VMS command language is easy to learn and use, and is suitable for both the interactive and batch environments. Extensive batch facilities under VAX/VMS include job control, multistream spooled input and output, operator control, conditional command branching and accounting.

Command procedures are supported by the command languages as an easy method of executing strings of frequently used sequences of commands, or creating new commands from the existing command set. Command procedures accept parameters and can include extensive control flow.

VAX/VMS provides a program development capability that includes editors, language processors, and a symbolic debugger. The VAX-11 FORTRAN, VAX-11 COBOL, VAX-11 BASIC, VAX-11 PL/I, VAX-11 PASCAL, VAX-11 BLISS-32, VAX-11 CORAL 66, and VAX-11 MACRO language processors produce native code. The PDP-11 BASIC-PLUS-2/VAX,PDP-11 FORTRAN IV/VAX to RSX, and MACRO-11 language processors produce compatibility mode code.

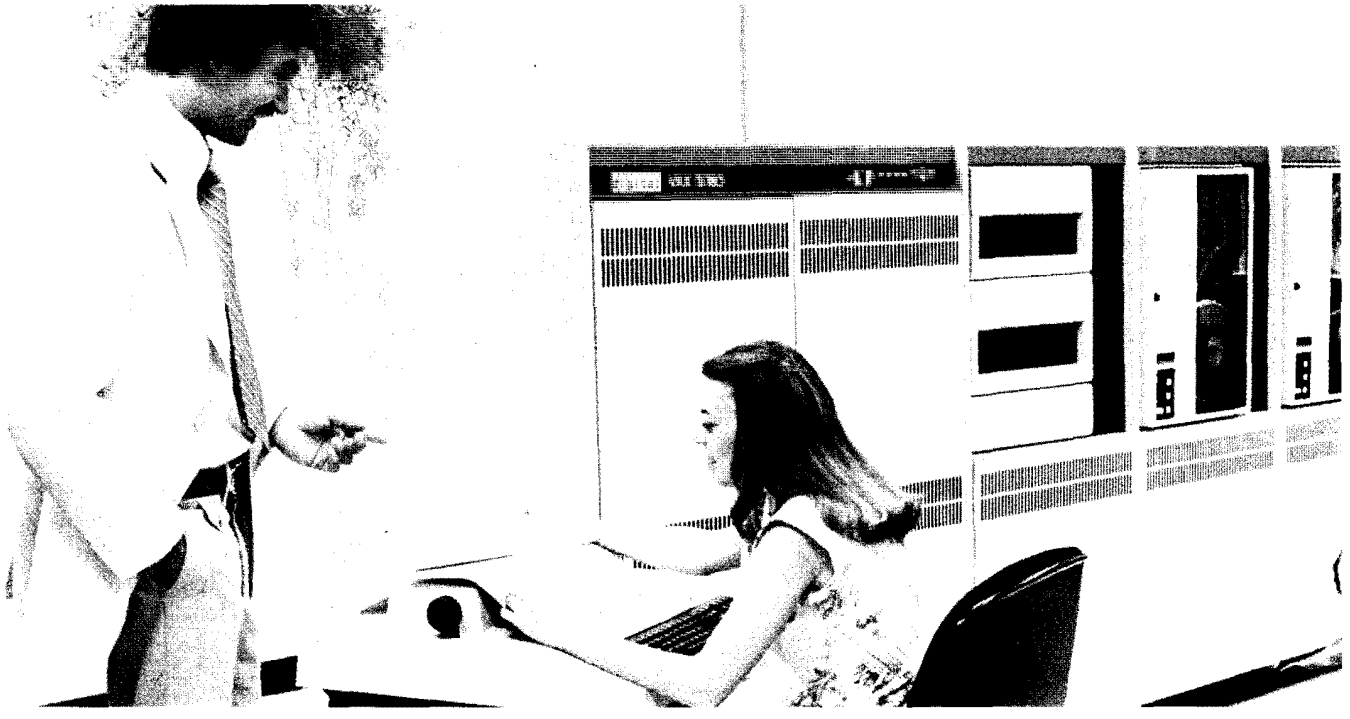
The VAX/VMS operating system provides a file and record management facility that allows the user to create, access, and maintain data files and records within the files with full protection. The record management services handle sequential, relative, and multi-key indexed file organizations, sequential and random record access, and fixed and variable-length records. Indexed files with sequential and random record access are available to compatibility mode programs, such as those written in PDP-11 BASIC-PLUS-2/VAX.

The VAX/VMS operating system supports the Files-11 On-Disk Structure Level 2 (ODS-2), which provides the facilities for file creation, extension, and deletion with owner-specified protections and multilevel directories. On-Disk Structure Level 2 is upwardly compatible with Level 1, the file system currently available under the PDP-11 IAS and RSX-11 operating systems. Both native and compatibility mode programs can access both Level 1 and Level 2 volume structures.

DECnet-VAX networking capabilities are available as an option for point-to-point interprocess communication, file access, and file transfer, and include down-line command file and RSX-11S system image loading. The Network Command Terminal facility allows users on one system to log into another VAX system on the network. The Mail facility allows electronic mail to be addressed to users on any VAX node in the network.

Peripherals

Medium capacity disks, unit record devices, terminals, the interprocessor communications links, and user-specific devices are UNIBUS peripherals. The UNIBUS adaptor(s)



provides the hardware pathways for data and control information to move between the UNIBUS and the memory interconnect.

High-performance MASSBUS mass storage peripherals are connected to the memory interconnect via a buffered MASSBUS adaptor. The MASSBUS adaptors provide the hardware pathways for data and control information to move rapidly between a MASSBUS peripheral controller and the memory interconnect.

PERFORMANCE

Many features of VAX ensure that the system provides real-time, computation, and transaction processing applications with the processing speed and throughput they need.

The processor provides 64-bit, 32-bit, 16-bit, and 8-bit arithmetic, instruction prefetch, and an address translation buffer.

An optional high-performance floating point accelerator (FPA) can be added to the VAX-11/780 system. The FPA is an independent processor executing in parallel with the base CPU. The FPA takes advantage of the CPU's instruction buffer to prefetch instructions and memory cache to access main memory. Once the CPU has the required data, the FPA overrides the normal execution flow of the standard floating point microcode and forces use of its own code. Then, while the FPA is executing, the CPU can be performing other operations in parallel. The FPA executes standard floating point instructions with substantial performance improvement. This execution is architecturally transparent to the programmer. In addition, the FPA enhances the performance of a number of additional instructions including:

- extended multiply and integerize (EMOD)
- polynomial evaluation (POLY), (F_floating and D_floating formats for both instructions)
- all floating to integer and integer to floating conversions
- 8- and 16-bit integer multiply (MULB and MULW)
- extended multiply (EMUL)
- 32-bit integer multiply (MULL)

The EMOD instruction is used for fast, accurate range reduction of mathematical function arguments. The POLY instruction is used extensively in the evaluation of mathematical functions such as sine and cosine (the VAX/VMS mathematics library makes use of the POLY instruction to significantly reduce the execution time of its subroutines). The MULL instruction is often used in matrix manipulation subscript calculations.

The VAX processor architecture is specifically designed to support high-level language programming. Because the instruction set is extremely bit efficient, compilation of high level language user programs is also very efficient. Among the features of the processor that serve to reduce program size and increase execution speed are the:

- variable length instruction format
- floating point, packed decimal, and character string data types
- indexed, short displacement, and program counter relative addressing modes
- small constant short literals

Furthermore, many instructions correspond directly to high-level language constructs:

- the Add Compare and Branch instruction for DO and FOR loop control
- the Case instruction for computed GO TO statements
- the 3-operand arithmetic instructions for statements such as "A = B+C"
- the Index instruction for computing index values, including subscript range checking
- the Edit instruction for output formatting

Much of the processor architecture also ensures that the operating system incurs minimal overhead for real-time multiprogramming. For example, the operating system uses:

- the context-switching instructions and queue instructions to schedule processes
- the asynchronous system trap (AST) delivery mechanism to speed returns from system service calls

Careful design, coding, and performance measurement ensure that the flow within the system is as rapid as possible.

RELIABILITY

Built-in reliability features for both hardware and software provide data integrity, increased up-time, and fast system recovery from power, hardware, or software failures. Several of the VAX reliability features are discussed in the following paragraphs.

Data Integrity

VAX provides memory access protection both between and within processes. Each process has its own independent virtual address space which can be mapped to private pages or shared pages. A process cannot access any other process' private pages. The VAX/VMS operating system uses the four processor access modes to read and/or write protect individual pages within a process. Protection of shared pages of memory, files, and interprocess communication facilities such as mailboxes and event flags is based on file ownership and application group identification.

The VAX/VMS file system detects bad blocks dynamically and prevents re-use once the files to which they are allocated are deleted.

Integral fault detection hardware includes:

- memory error correcting code that detects all double-bit errors and corrects all single-bit errors
- disk error correcting code that detects all errors up to 11 bits and corrects errors in a single burst of 11 bits
- extensive parity checking
- peripheral write-verify checking that checks all input and output disk and tape operations and ensures data reliability
- track offset retry hardware that enables the operating system to recover from many disk transfer errors

System Availability

The VAX/VMS operating system allows the VAX system to continue running even though some of its hardware components have failed. The system automatically determines the presence of peripherals on the processor at bootstrap time. If the usual system device is unavailable, the system can be bootstrapped from any disk. If memory units are defective, memory is configured so that defective modules are not referenced. Software spooling allows output to be generated even if the normal output devices are not available. Also, devices can be added on line.

The system operator can perform software maintenance activities without bringing the system down for stand-alone use. The operator can perform disk backup for both full volume backup/restore and single file backup/restore concurrent with normal activities.

The VAX/VMS operating system supports on-line peripherals diagnostics. VAX/VMS performs on-line error logging of CPU errors, memory errors, peripheral errors, and software failures. The operator or field service engineer can examine and analyze the error log file while the system is in operation.

System Recovery

Automatic system restart facilities bring up the system without operator intervention after a system failure caused by a power interruption, a machine check hardware malfunction, or a fatal software error. The VAX/VMS operating system automatically performs machine checks and internal software consistency checks during system operation. If the checks fail, VAX/VMS performs a system dump and reboots the system if the operator has set the system for auto-restart.

Memory battery backup can be used to preserve the contents of memory during a power outage. A special memory configuration register indicates to the recovery software whether data in memory was lost. Following a power failure, the recovery software restarts all possible I/O in progress before the failure occurred. Programs can use the VAX/VMS power-fail asynchronous system trap facility to initiate user-specific power fail recovery processing. If memory battery backup is used, the time-of-year clock allows the recovery software to calculate elapsed time of the outage.

VAX remote diagnosis allows DIGITAL to run diagnostics, examine memory locations, and diagnose hardware/software problems from a remote diagnosis center. The engineer who goes to the site is prepared in advance to correct any problems that may have occurred.

FLEXIBILITY

VAX is a system that is easy to use because it is both flexible and easy to extend. Several of the ways in which VAX provides the user with flexible operating and programming environments are introduced below.

Flexibility in the Operating Environment

Virtual memory gives the user the ability to write and execute arbitrarily large programs without concern for addressing limitations. The paging and swapping algorithms allow more programs to execute than the available physical memory would allow if all programs had to be totally resident.

Both paging and swapping are transparent to the user, and therefore allow the system to be extended without reprogramming. The system's physical memory configuration can change without requiring program redesign or relinking. Programmers never have to structure their programs, although they can, at their option, to achieve maximum efficiency and performance for a given program. They can control working set size, lock pages in the working set or memory, and lock an entire working set in memory. In addition, the system manager can control the amount of time a process is guaranteed memory residency once it is swapped in.

The VAX/VMS scheduler recognizes 32 scheduling priorities. A program can modify its priority during execution. Real-time processes execute at one of the high 16 priority levels, and normal processes (including system processes) execute at one of the low 16 priority levels. The scheduler may temporarily increase the priority of a normal process to increase its response to I/O events or system events (but it can never lower the priority of real-time process).

Batch and printer output processing are completely flexible. The operator controls the number of batch jobs that can run concurrently. The operator defines the number of spooler queues. There can be multiple print queues: a generic queue for jobs that can be output on any printer, and several queues for jobs that are designated for a specific printer.

Batch jobs can be submitted to batch streams from the interactive environment using a terminal command, from another batch job, or by any program using a system call. Submitted batch jobs are queued, and a time can be specified after which a batch job should be executed.

Flexibility in Programming Interfaces

The I/O programming facilities can be as device-independent or device-specific as desired. The record management services support high-level programming languages by providing transparent record access and also enable the programmer to request specific record management services or system services to control file allocation, record blocking, or record accessing directly. Programmers can also use the system services to access file-structured devices or non-file structured devices if they wish to use their own record processing or volume structuring techniques.

Access to network facilities is device-independent, but a user who so desires can exert control over operations to obtain error reports or notification of broken connections (interrupt messages, inbound connections). System access protection applies to all network access.

Programmer Productivity

In addition to the system's reliability and performance features described above, VAX offers many tools to aid programmer productivity.

- Interactive editors with CAI startup—VAX/VMS supports several interactive and batch text editors, including SOS, SLP, and now the DIGITAL standard editor EDT. The system features a computer-aided instruction course to introduce the user to the power and flexibility of EDT.
- Interactive symbolic debugger—The interactive symbolic debugger provides a fast and efficient method by which the user can trace program errors. The debugger offers the user such features as; support of various native languages, support of many data types, and its interactive symbolic operation, i.e., the user can refer to program locations using those symbols created within the program.
- FMS interactive screen format generation—The Forms Management System contains an interactive editor which allows the application programmer to create and/or modify screen formats.
- DATATRIEVE—DATATRIEVE software provides fast and convenient access to data within a file or files. This query/report writing system provides the user with either video or hardcopy output.
- HELP facility—The HELP facility provides the user with on-line instructions pertaining to selected system operations.

Extending the System

The VAX/VMS command language can be extended with user-defined commands through the use of command procedures. A command procedure is a set of commands, data, or other command procedures processed in sequence. The user can invoke command procedures by a single command that can include parameters for the procedure, such as file names or values for symbols. Command procedures can execute programs, transfer control within the command procedure conditionally or unconditionally, request input from the user, and manipulate numeric and string symbols.

VAX/VMS uses a standard procedure call interface supported by the processor's call instructions. The calling program and called procedure can be written in different languages. This contributes to the writing of error-free, modular, and maintainable software that can be shared by many programs. The standard procedure call interface is particularly useful to systems programmers who want to add their own shareable libraries and library procedures to the VAX/VMS Common Run Time Procedure Library.

PDP-11 Compatibility

Users who already know the PDP-11 will find the native VAX-11 instruction set and programming characteristics easy to learn when developing new applications for the VAX system. The PDP-11 and the VAX systems have almost identical FORTRAN, BASIC, and COBOL languages. Users who have programmed in any of these languages on the PDP-11 will need to spend very little time learning the VAX system.

VAX offers many PDP-11 compatibility features:

- the VAX processor can execute a subset of PDP-11 16-bit instructions in compatibility mode
- the VAX/VMS operating system provides functionally equivalent system services for many RSX-11M execu-

tive directives

- the VAX/VMS high-level language compilers accept source languages that are upwardly compatible with the same PDP-11 compilers
- the VAX/VMS file system can read and write disk volumes and magnetic tapes written under RSX-11 and IAS operating systems
- the VAX/VMS record management services provide record processing methods that are upwardly compatible with RMS-11 record management services
- the VAX/VMS operating system provides an RSX-11 MCR command language interpreter
- the DECnet-VAX package supports RSX-11S system image down-line loading

These features make VAX an ideal host system to PDP-11 systems in a distributed processing environment.

3
The Users

The VAX system is designed to execute many different kinds of jobs concurrently. Jobs consist of one or more processes, each of which can be executing a program image that interacts with on-line users, controls peripheral equipment, and communicates with other jobs in the same system or in remote computer systems. Jobs include:

- customer-written interactive, batch, and real-time applications
- interactive and batch program development jobs
- system management and control jobs

Typically, VAX users interact with application or system jobs via an on-line terminal, or benefit from production batch or real-time jobs. To aid in the development of interactive, batch, and real-time applications, and manage and control system resources, VAX enables:

- *The application programmer* to write, compile, and test programs interactively or in batch mode, taking advantage of source code, object code, and program image libraries.
- *The system programmer* to design application systems that require a high degree of job and process interaction, data sharing, response time, and system and device independence.
- *The system manager* to authorize users, limit resource usage, and grant or restrict privileges individually.
- *The system operator* to monitor operations, service user requests, and control batch production.

Users can directly control the operation of VAX through the operating system's command language. In general, the command language is used by programmers to develop application software, by operators to monitor the system, and by system managers to assign user privileges.

Application programmers may also employ the command language to execute their application programs explicitly. The command language may be easily extended to provide custom-tailored commands defined by the user. Customer-written application programs can provide their own command interfaces for people using the system. Transaction processing applications may require several terminals to be slave terminals, meaning that they are tied to particular application programs that handle requests entered by the user.

The system manager can assign access user names and passwords to users who log on the system at a command terminal, and determine their privileges for obtaining services and limits for using resources. Users who access the system through an application terminal interface have the resources and privileges granted to the application programs run on their behalf. An application program itself determines who can request its services.

THE APPLICATION PROGRAMMER

The application programmer has four basic tools for requesting services of the system:

- command interpreter
- programming languages
- programmed file and record management services
- programmed system services

The application programmer gains access to the system through the command interpreter. The command interpreter enables the programmer to create, compile, and execute programs written in any of several programming languages. The record management services are available through any programming language to provide device-independent data processing. The system services, although primarily of interest to systems programmers, are also available to the application programmer for requesting special services of the operating system.

Command Language

The command interpreter is interactive, comprehensive, easy to use, and extremely flexible. It enables the user to log onto the system, manipulate files, develop and test programs, and obtain system information. Furthermore, it enables users to extend or redefine their command repertoire as well as write command procedures easily. The command language includes:

- a set of English commands that provide the basic command repertoire
- a set of control characters that provide special functions such as erase command line, interrupt the program currently executing, etc.
- a set of special operators and commands that can be used to automate command streams and extend the command repertoire

Table 3-1 lists the basic set of English commands accepted by the command interpreter. The command interpreter is easy to use because its commands can be abbreviated, because it prompts for necessary arguments, and because it assumes standard or user-selected default values for command parameters and qualifiers.

A command line normally consists of a command verb followed by one or more parameters that identify the object of the operation (a file, for example) or qualify how the operation is to be performed. If the interactive user enters an incomplete command, the command interpreter prompts for any necessary parameters. For example, the COPY command, which creates a copy of an existing file, accepts a total of two file specifications: one for the file to be created and one for the file to be copied. The file specifications identify the exact location and name of the files. The COPY command can be entered in any of several ways:

```
$ copy
$_FROM: file-name-1
$_TO: file-name-2

$ copy file-name-1
$_TO: file-name-2

$ copy file-name-1 file-name-2
```

The command interpreter displays the dollar sign to prompt for a command, and the dollar sign underscore to prompt for a missing parameter.

Command Procedures

To eliminate the need for typing frequently repeated sequences of commands, users can create **command procedures**. A command procedure is a file containing complete command lines (including the $ prompt character). The user can request the command interpreter to read and process the command lines in a command procedure file just as if they were being typed successively at the terminal. To execute a command procedure, the user simply precedes the name of the command procedure file with an "at" sign (@):

```
$ @procedure-file
```

Figure 3-1 is a simple example of how the user might create, interactively, a new command called EXECUTE. The user has previously written a VAX-11 PASCAL program named AVE, and now wishes to compile, link, and execute this program. To the user, EXECUTE appears to be a command like any other command in the DCL command repertoire. The first step is to create the command procedure file EXECUTE.COM. Following this, the user enters the PASCAL, LINK, and RUN DCL commands as input to the command file.

The blue dollar signs in Figure 3-1 represent DCL system prompts, the remainder of the text is user supplied.

```
$ CREATE EXECUTE.COM
$ PASCAL AVE
$ LINK AVE
$ RUN AVE
↑Z (CONTROL Z)
$
```

Figure 3-1
A Simple Command Procedure

Now, to execute the command procedure file, EXECUTE.COM, the user can type:

```
$ @EXECUTE
```

or the user can create a symbol to execute the command file. The user may equate a unique series of letters to the command procedure file. For instance:

```
$ EXE := @EXECUTE
```

Now, to execute the command procedure file, EXECUTE.COM, the user need only type the symbol EXE:

```
$ EXE
```

The user could just as easily have created a symbol called GO to execute the command procedure file EXECUTE.COM. In this instance:

```
$ GO:=@EXECUTE
```

Now, to execute the command file, EXECUTE.COM, the user can enter GO in response to the DCL prompt ($):

```
$ GO
```

Table 3-1
DCL Command Language Summary

GENERAL SESSION INFORMATION AND CONTROL

LOGIN	The user initiates an interactive session with the system by typing CTRL/C, CTRL/Y or by pressing the carriage return on a terminal not currently in use. The system then prompts for username and password, and validates them.
HELP	Displays information to assist the user in selecting the proper command qualifiers.
SHOW	Displays any of the following information: current day, date, and time; current default device and directory name; status of devices in the system; logical device name assignments; current characteristics and status of specified mag tape device; name, number, and status of local network node and lists available remote nodes; default characteristics of system printer; status of current process; current file protection to be applied to all new files created during terminal session or batch job; current status of entries in printer/batch job queues; current disk quota; current VAX-11 RMS default multiblock and multibuffer counts; status of currently executing image in process; current value of a local or global symbol; displays a list of processes and status information in system; current characteristics of a specified terminal; logical name translations; display of working set quota and limit assigned to current process.
SET	Defines default translation mode for cards read into system card reader; controls whether command interpreter receives control when CTRL/Y is pressed; changes the user's default device name or directory name; defines default characteristics for specific magtape device; determines whether command interpreter performs error checking following execution of commands in command procedures; changes execution characteristics of currently executing process; changes a file's protection; changes current status or attributes of a file queued for printing or for batch job execution; defines default values for multiblock and multibuffer counts used by VAX-11 RMS; changes characteristics of a specified terminal; controls whether or not command lines in command procedures are displayed at terminal or printed in batch job log; redefines default working set size for the current process.
ASSIGN	Assigns a logical name to a given character string (equivalence name) and stores the the pair of names in a process, group, or system logical name table. Generally used to create a logical name for a device.
DEFINE	Creates a logical name equivalence. (Same as ASSIGN except for syntax.)
DEASSIGN	Breaks the correspondence between a logical name and its equivalence name (see ASSIGN and ALLOCATE), or deletes a symbol (see DEFINE).
MCR	Signifies that the given command or following command lines are to be interpreted by the RSX-11 command interpreter.
LOGOUT	Terminates an interactive session and releases all resources allocated to the user.
REQUEST	Displays a message at a system operator's terminal and optionally requests a reply.

BATCH AND COMMAND PROCEDURE SPECIFIC CONTROL*

SUBMIT	Places a given batch command file or command procedure in a batch queue for processing.
$PASSWORD	Specifies the password associated with the user name specified on a JOB card for a batch job.
$JOB	Indicates the beginning of a batch command file and provides job control information (such as time limit).
$INQUIRE	Requests interactive assignment of a value to a symbol and assigns the symbol a name.
$GOTO	Transfers flow of control to a given labeled line.
$ON	Transfers flow of control to a given labeled line if an error of a given severity or greater is encountered at any time during command procedure processing.
$IF	Transfers flow of control to a given labeled line if the result of a logical comparison of symbolic values is true.
$EOD	Signifies the end of data in the input stream following a $DATA command.
$EXIT	Terminates the command procedure.
$EOJ	Marks the end of a batch job submitted through the system card reader. EOJ performs the same functions as the LOGOUT command.
DECK	Marks the beginning of an input stream for a command or program.

*Command names preceded by a $ are meaningful only in a batch command file or command procedure. All other commands listed in this table can either be issued interactively or used in a batch command file or command procedure.

VOLUME AND DEVICE RESOURCE CONTROL

MOUNT	Requests the operator to make a volume available to the user and optionally associates a logical name with the volume or volume set.
INITIALIZE	Writes a directory file and other volume structuring information on a disk or magnetic tape volume to prepare it for use.
DISMOUNT	Requests the operator to break the logical association of this device with the user's job.
ALLOCATE	Obtains exclusive ownership of device and enables the user to assign a logical name to the device.
DEALLOCATE	Releases allocated devices.

FILE MANIPULATION

DIRECTORY	Reports information (size, protection, ownership, creation time, etc.) on a given file or set of files.
CREATE	Creates a new file from data subsequently entered in the input stream (user at terminal or batch stream). Creates a directory file on a volume.
EDIT	Opens a text file and accepts commands to insert, delete or modify data in the file.
DELETE	Deletes one or more files from a mass storage disk volume.
DELETE/ ENTRY	Deletes one or more entries from a printer or batch job queue.
DELETE/ SYMBOL	Deletes one or more symbol definition from a local symbol table or from the global symbol table.

Table 3-1 (con't)
DCL Command Language Summary

PURGE	Deletes all but the latest version of a given file or files, optionally keeping the latest two or more versions.
RENAME	Changes the name of one or more existing files.
COPY	Copies the contents of a file or files, creating another file or files.
APPEND	Concatenates the contents of sequential files to a given output file, or creates a new output file from the concatenated contents of given sequential files.
DIFFERENCES	Compares two files and reports the differences between the two.
SORT	Creates a file by rearranging the records in a given file based on the contents of key fields within the records.
OPEN	Opens a file for reading or writing at the command level.
CLOSE	Closes a file that was opened for input or output with the open command and deassigns the logical name specified when the file was opened.
READ	Reads a single record from a specified input file and equates the record to a specified symbol name.
WRITE	Writes a record to a specified output file.
PRINT	Sends the contents of a given file or files to a spooled output device such as a line printer.
TYPE	Displays the contents of a given file or files on the device identified by the logical name SYS$OUTPUT: (default generally the user's terminal).
DUMP	Produces a printed listing of the contents of a file, ignoring any print formatting characters that may appear in the records.
UNLOCK	Permits access to a file that was improperly closed.
ANALYZE/ OBJECT	Provides a description of the contents of an object file or an executable image file.
PROGRAM DEVELOPMENT AND EXECUTION CONTROL	
MACRO	Assembles given assembly language source modules, producing an object module.
FORTRAN	Invokes the VAX-11 FORTRAN compiler to compile one or more source programs.
COBOL	Compiles given COBOL language source modules using VAX-11 COBOL compiler, producing an object module.
BASIC	Compiles given BASIC language source modules, producing an object module.
BLISS	Invokes the VAX-11 BLISS-32 compiler.
PL/I	Invokes the VAX-11 PL/I compiler to compile one or more source programs.
PASCAL	Invokes the VAX-11 PASCAL compiler to compile one or more source programs.
CORAL	Invokes the VAX-11 CORAL 66 compiler to compile one or more CORAL source programs.
LIBRARY	Creates, deletes, or maintains libraries of object modules, shareable images, or macro source modules.
LINK	Links modules to produce images.
RUN	Executes a program image in the current process context, or creates a detached process and executes a program image in that process context.
DEBUG	Starts interactive debugging session after interrupting program image execution by typing a Control C or Control Y.
EXAMINE	Displays the contents of a location in virtual memory.
DEPOSIT	Replaces the contents of a location in virtual memory with the given data.
CONTINUE	Resumes execution of a program interrupted by typing a Control C or Control Y.
STOP	Terminates the program currently interrupted by a Control C or Control Y.
SUBMIT	Enters a command procedure in the batch job queue.
SYNCHRONIZE	Places the process executing a command procedure in a wait state until a specified batch job completes execution.
WAIT	Places the current process in a wait state until a specified period of time has elapsed.
CANCEL	Cancels scheduled wakeup requests for a specified process. This includes wakeups scheduled with the run command and with the schedule wakeup ($SCHDWK) system service.

In addition to executing command procedures at a terminal, an interactive user can also submit batch jobs. Batch jobs execute under control of the system operator and leave the user's terminal free to continue interactive or command procedure processing. A batch job can be submitted as a deck of cards or as a batch command file. A batch command file is identical to a command procedure file, except that a batch command file submitted as a deck of cards begins with a $JOB card that provides job control information.

However, DCL is more than just a string of commands capable of standing alone; it possesses true high level programming language statements such as GOTO, IF, etc., and accepts a series of up to eight user-defined parameters 'P1' through 'P8'. DCL can be used to completely define and control a user environment tailored to a specific application.

The power and flexibility of command procedures and the DCL command language will be treated in greater detail in the Program Development and Support Facilities section.

RUN Command

The RUN command includes several qualifiers (DELAY, INTERVAL, and SCHEDULE) which are of particular importance to the real-time programmer.

Specifying any of the above qualifiers places a process in

hibernation, a wait state in which the process can be reactivated only when a particular time value occurs. The time value can be specified in delta time (/DELAY qualifier), in absolute time (/SCHEDULE qualifier) or at recurrent intervals (/INTERVAL qualifier). When the image completes execution, the process returns to a state of hibernation.

Programming Languages

The system includes the VAX-11 MACRO assembler for programming the machine using its native instruction set. A wide variety of language processors are optionally available to high-level language programmers: VAX-11 FORTRAN, VAX-11 COBOL, VAX-11 BASIC, VAX-11 PL/I, VAX-11 PASCAL, VAX-11 CORAL 66, and VAX-11 BLISS-32. In addition, VAX/VMS supports several optional language compilers that execute in compatibility mode. These include PDP-11 BASIC-PLUS-2/VAX, PDP-11 FORTRAN IV/VAX to RSX and MACRO-11. These language processors, introduced below, are described fully in the Languages section.

The VAX-11 FORTRAN language processor is based on the American National Standard FORTRAN specification X3.9-1977 (commonly referred to as FORTRAN-77). The VAX-11 FORTRAN compiler supports this standard at the full language level. Additionally, however, VAX/VMS provides support for the industry-standard FORTRAN features based on FORTRAN-66 (an option that can be selected at compile time), the previous ANSI standard.

The VAX-11 FORTRAN compiler produces shareable, highly optimized VAX-11 native object code. The compiler takes advantage of the system's large virtual address space while utilizing the floating point and character string instructions. FORTRAN I/O processing is supported by the record management services (VAX-11 RMS). VAX-11 FORTRAN object modules can be linked with assembler-produced object modules and the system's run-time library, which is common to all native mode programs, to provide standard library functions. The VAX-11 FORTRAN language processor offers the programmer such features as:

- Full ANSI-77 FORTRAN language support
- Access to ISAM files as well as relative and sequential files
- Access to the VAX/VMS system services and the run time library procedures
- FORTRAN program can call external routines written in other VAX-supported high level languages
- The compiler itself is shareable

The VAX-11 COBOL language processor produces highly efficient shareable native mode code which utilizes the system's packed decimal and character instruction set and extended call facility. The VAX-11 COBOL language is based on the American National Standard Programming Language COBOL, X3.23-1974, the industry wide accepted standard for COBOL. Many features of the planned COBOL standard (anticipated in 1981) are also included. The VAX-11 COBOL language processor offers the programmer such features as:

- the ability to manipulate data strings via the INSPECT verb
- performing sorting and merging operations at the COBOL source language level
- complete file organization capability including sequential, relative, and indexed I/O
- structured programming
- support for the full range of data types including packed decimal and floating point
- COBOL programs can call external routines written in COBOL or other VAX-supported high level languages
- capability of writing shareable code for use by other native mode high level languages
- accepts source programs coded in either ANSI standard format or the shorter easy to read DIGITAL terminal format

VAX-11 BASIC is a native mode, shareable language processing system producing shareable VAX native object code. The language compiler utilizes VAX floating point and character string instructions while supporting a fast RUN command and immediate mode execution which makes it well suited for interactive use. VAX-11 BASIC is a superset of PDP-11 BASIC-PLUS-2, offering the VAX user major enhancements such as:

- access to VAX-11 RMS file and record processing
- long variable names (up to 31 alphanumeric characters)
- dynamic string handling
- CALL statement providing interface to common language environment
- shareable and re-entrant code

VAX-11 PL/I is an extended implementation of the proposed ANSI X3.74, American National Standard PL/I General Purpose Subset to full PL/I (ANSI X3.53-1976). VAX-11 PL/I extensions include some full language features and VAX/VMS system-specific extensions.

PL/I is a versatile language that is suited to commercial, scientific, and systems programming applications. Some of the features of VAX-11 PL/I include:

- block structured language
- full support for all VAX-11 hardware data types
- powerful I/O capabilities including ISAM support
- user control of storage allocation
- condition handling
- standard VAX-11 CALL interface, including access to VAX/VMS System Services and the run-time library
- fast, native-mode optimizing compiler
- shareable, position-independent code

VAX-11 PASCAL, a re-entrant native mode compiler, is an extended implementation of the PASCAL language as defined in the PASCAL User Manual and Report (Jensen and Wirth, 1974). Particularly suited to instructional use, PASCAL is gaining increasing popularity as a general purpose language. Major features of the VAX-11 PASCAL language include:

- block structuring via the BEGIN...END compound statement to allow easy logic flow
- data structuring including the ability to declare and use pointers, records, files and arrays

- predefined procedures and functions to deal with I/O handling and data manipulation

VAX-11 PASCAL takes advantage of the VAX hardware floating point and character instructions as well as the virtual memory capabilities of the VAX/VMS operating system. Many of the features common to other native languages are available through VAX-11 PASCAL including:

- separate compilation of modules
- standard CALL interface to routines written in other languages
- access to VAX/VMS system services

The VAX-11 CORAL 66 compiler executes in compatibility mode and generates native mode object code under VAX/VMS. The CORAL language, derived from JOVIAL and ALGOL-60 in 1966, is the standard language prescribed by the British government for military real-time applications and systems implementation. VAX-11 CORAL 66 is essentially a high level block-structured language whose compiler offers the user many features including:

- several numeric types (byte, long and double)
- generation of re-entrant code at the procedure level
- code optimization
- English text error messages
- INCLUDE keyword to incorporate CORAL 66 source code from user-defined files

VAX-11 BLISS-32 is a high-level systems implementation language for VAX-11, which runs in native mode under VAX/VMS. The BLISS language is specifically designed for building language compilers, real-time processors, utilities, and operating system software. BLISS contains many of the features of high-level languages (e.g., DO loops, IF-THEN-ELSE statements, automatic stack, and mechanisms for defining and calling routines), but it also provides the flexibility and access to hardware that one would expect from an assembly language. VAX-11 BLISS-32 can be used as an alternative to assembly language coding in all except the most machine-dependent systems programming applications. The VAX-11 BLISS-32 language processor offers the programmer such features as:

- program execution on architecturally different machines with little or no modification
- construction of complex expressions in which several different kinds of operations can be performed in a single program statement
- exploitation of high level language constructs

PDP-11 BASIC-PLUS-2/VAX is an optional language processing system that includes a compiler and an object time system. PDP-11 BASIC-PLUS-2 is also available as an optional language processor for the RSTS/E, RSX-11M, RSX-11M-PLUS, and IAS operating systems. The PDP-11 BASIC-PLUS-2/VAX compiler produces code that executes in PDP-11 compatibility mode.

PDP-11 FORTRAN IV/VAX to RSX is an extended FORTRAN IV processor based on ANSI FORTRAN X3.9-1966. It supports mixed mode arithmetic, extended I/O facilities for data formatting, error condition transfer statements, bit manipulation, library usage, and several debugging facilities. The FORTRAN IV compiler (and its run time system) execute in the compatibility mode environment.

MACRO-11, the PDP-11 assembly language, is included in the compatibility mode environment. Programs written in MACRO-11 can be assembled to produce relocatable object modules and optional assembly listings.

Record Management Services

The record management services (RMS) are a collection of procedures that extend the programming languages by providing general purpose file and record handling capabilities. Programmers using RMS include in their programs statements that read, write, find, delete, and update records within files. Records can be fixed or variable length.

RMS enables the programmer to choose the file organization and record access method appropriate for the data processing application. The file organizations and record access methods are independent of the language in which they are programmed, although some languages support file organizations and access methods not provided in others. Every programming language uses RMS to process files organized to provide sequential, random or multi-keyed indexed record accessing.

For further information on RMS and the system's data management techniques, refer to the section on Data Management Facilities.

THE SYSTEM PROGRAMMER

The system programmer can use this system to design and build application systems for multiprogramming environments requiring fast response and a high degree of job interaction and data sharing.

Job and Process Structure

The user program environment consists of a job structure that can contain many processes. A process is the schedulable entity capable of performing computations in parallel with other processes. It consists of an address space and an execution state that define the context in which a program image executes. An executing program is associated with at least one process, but it can be associated with several processes.

A multiple process job structure allows one job to execute more than one program image at the same time. One process can wait for an event (such as I/O completion) to occur while another process continues its computations. The processes can communicate in several ways. They can coordinate their execution synchronously using event flags or asynchronously using software-simulated interrupts. They can send messages back and forth using virtual record-oriented devices called "mailboxes," and they can share code and data on disk and in memory.

Jobs can be grouped into application subsystems that share code and data protected from other applications. The processes within jobs in the same group can coordinate their activities using group interprocess communication facilities such as mailboxes and event flags, as well as those facilities local to the job. They can access files and data in memory that are protected from other groups in the system.

Multiprogramming Environment

The system supports multiprogrammed applications that require high performance by providing:

- event driven priority scheduling
- rapid process context switching
- minimum system service call overhead
- processor access mode memory protection
- memory management control

The system schedules processes for execution based on the occurrence of events such as I/O completion as well as time quantum expiration. When scheduled, the context switching and interrupt processing hardware and software ensure that processes are activated quickly. Real-time processes can be assigned high priorities to ensure that they receive processor time on demand. A process can schedule its execution at a given time of day or after an interval has elapsed, and an appropriately privileged process can modify its priority during execution.

The system's memory management hardware and software ensure that paging, swapping, and dynamic memory allocation are both efficient and transparent to the programmer. Where real-time applications require performance control, both paging and swapping can be reduced or eliminated by increasing the amount of memory allocated to a process and by locking a process in memory. Because memory management is transparent, programs can be written and later tuned for performance after they are tested. The system provides a utility program to aid system programmers in evaluating the effectiveness of the memory management system for their processes.

Program Development

This system provides the system programmer with tools that support highly modular program and applications development. By taking advantage of these tools, the programmer can build applications quickly, and easily modify and extend them later.

The system includes editors, compilers, librarians, linkers, and debuggers for both the native and compatibility programming environment. All program development utilities can be used either interactively or in batch mode, including the editors and debuggers. The native symbolic debugger recognizes a command language similar to the operating system command language and uses expressions similar to the language in which the program being debugged was written.

Executable program images can be built using extensive libraries. In the native programming environment, the programmer can create libraries of assembler macro definitions, of object modules, and of image modules. The system also includes the common run time procedure library, which provides library functions common to all native programming languages.

All program interfaces to the operating system and its utilities have uniform calling standards. System programmers can add new library procedures to the common run time procedure library and install them on-line without modifying existing programs and utilities, since all arguments are passed using standard data structures.

Furthermore, user programs can be written to be completely device independent through the system service and command language logical naming facilities. All files and devices can be identified using arbitrarily defined logical names that can be assigned values at run time.

THE SYSTEM MANAGER

A job is normally associated with a user known to the system to have certain privileges, quotas, and resources. The system manager authorizes users, plans data access and protection, grants privileges, controls resource utilization, and analyzes the system's accounting and performance information.

User Authorization

The system manager controls use of the system primarily by creating user authorization information. This information is recorded in a specially maintained and protected file called the **user authorization file**. The system manager can create, examine, and update this file at any time.

The file contains one entry for each user authorized to access the system. Each entry:

- identifies the user
- supplies defaults
- specifies privileges
- limits resource usage

User identification consists of a unique user name, a password, a default account name, and a user identification code (UIC). When logging onto the system, people must always enter their user name and password to gain access to the system. The password is not displayed on the user terminal. Privileged users can change the passwords they are assigned as often as they desire.

This system's data protection scheme is based on the user identification code (UIC) that the system manager assigns. A UIC controls each user's access to the data structures protected by UICs, which include both files and the interprocess communication facilities such as mailboxes, shared areas of memory, and event flags.

A UIC consists of a group number and a member number. Every user is assigned a UIC, and every data structure is assigned both a UIC and a protection code. A protection code identifies what types of access are available to which users. There are four types of access (read, write, execute, and delete), and there are four types of user (owner, group, world, and system). The owner is any user that has the same UIC as that assigned the data, the group is any user that has the same group number as that assigned the data, the world is any user, and the system is any user with a group number of 1 through 10.

Using this protection scheme, a system can have files and interprocess communication facilities that are available for access only by users having the same UIC, or for access only by users in the same group, or for universal access. Furthermore, since each data structure has its own protection code, it is possible to protect each data structure assigned the same UIC on a different basis. The system UICs are generally reserved for system users and system pro-

grams and data structures. This arrangement enables a user to protect a file from access by anyone other than the owner or group, but still enables the system to access the file for operations such as backup.

In addition to identifying the user and the set of data structures the user can access, the user authorization file supplies the user with a default file protection, a default directory name, and a default device name. When the user creates a file, the system assigns the default file protection unless requested otherwise. An owner can modify a file's protection at any time.

Directory names are arbitrary character strings identifying a directory file. A directory is simply a file containing a list of file names and other identification information that is used to find files on a volume. The default directory name identifies the directory that lists the files the user normally accesses. The default device name is the name of the device on which the volume containing the files the user normally accesses is mounted.

When the user issues a command to the command interpreter that operates on a file, or runs a program that opens a file, the file system uses the default directory name and default device name to locate the file unless specifically requested to use some other directory name or some other device name. The user can change the default directory and device names for a given session.

For further information on directories and directory structures, refer to the section on Data Management Facilities.

Privileges

Each user's authorization file entry contains a list of the privileges that the user can invoke. They include interprocess communication and control privileges, performance control privileges, file and device access privileges, and system operational control privileges. The system manager can grant distinct privileges individually to each user. Table 3-2 lists some of the privileges.

Privileges are checked when the user executes program images. If a user runs an image that attempts to execute a function requiring a privilege the user is not granted, the image incurs a privilege violation. For example, diagnostic programs require the privilege to issue device level diagnostic functions and the privilege to send messages to the error logger. Users not granted these privileges will receive privilege violations if they attempt to run diagnostics.

In certain cases, however, it is desirable to let a user run an image that requires privileges the user is not granted. For example, the login program image requires the privilege to switch to a more protected processor access mode to set the user's initial context in a protected area of memory. To let a user run an image that requires special privileges, the system enables the system manager to install **known images**. When the user runs a known image, the user obtains the necessary privileges to execute the functions required by the image, but only for the duration of that image's execution.

Resource Quotas and Limits

The user authorization file also provides the limits on how many system resources a user can tie up while logged on the system, and quotas for how much of a resource a user can use up during an accounting period. The system manager can assign user quotas for the maximum amount of CPU time accumulated during a given accounting period, and can limit the amount of dynamic system memory a job can utilize for buffers. The system manager can set disk usage quotas via the disk quota utility on a per user, per volume or volume set basis. VAX/VMS will automatically record usage and enforce the assigned quotas during file operation. However, each user possessing a private volume controls the disk quotas on that volume. The limits

Table 3-2
Privileges Summary

INTERPROCESS CONTROL
- create event flag clusters
- create permanent common event flag clusters
- create temporary mailboxes
- create permanent mailboxes
- create global sections
- suspend, resume, wake, and delete processes within the same group
- suspend, resume, wake, and delete any process
- create detached processes
- create and delete shared memory sections
- map to physical pages

ACCESS TO FILES AND DEVICES
- insert logical names in group logical name table
- insert logical names in system logical name table
- allocate spooled devices
- obtain exclusive ownership of a shared device
- override volume protection
- issue mount requests

PERFORMANCE CONTROL
- execute time critical images
- lock process in memory

SYSTEM OPERATION CONTROL
- issue operator commands
- set any privilege bits
- set process priority

PROGRAM EXECUTION
- execute Change Mode to Executive system service
- execute Change Mode to Kernel system service
- bypass file protection
- issue diagnostic functions
- send messages to error logger
- suppress accounting messages
- issue logical and physical I/O functions

imposed by VAX/VMS include the maximum number of:

- outstanding open files
- CPU time
- outstanding subprocesses created
- pages in a process working set
- pages in system paging files
- outstanding entries in the timer queue
- outstanding system buffered I/O requests
- bytes in system buffered I/O request
- outstanding direct I/O requests

Resource Accounting Statistics

The system maintains an accounting information file for collecting cumulative resource usage statistics. The system updates the accounting information file with detail records each time a process terminates. The detail statistics include:

- elapsed CPU time
- login (connect) time
- number of volumes mounted
- number of pages printed
- largest process virtual size
- largest process working set size
- number of page faults
- number of system buffered I/O requests
- number of direct I/O requests

A detail record identifies the account name, user name, and user identification code (UIC) to which the statistic applies. The accounting information file can be used to calculate billing information and reporting by account name, user name, or UIC. Because the system collects all detail records, system managers can define their own algorithms for resource usage billing.

Performance Analysis Statistics

The system collects statistics on its activities to help system programmers and managers tune the system for maximum performance. The information collected includes:

- System and Job Statistics—indicate the current number of processes, interactive users, and batch jobs in the system, the date and time at which the system was booted, and the current date and time.
- Processor Access Mode Usage—indicates how much time is spent executing at each of the access modes as a measure of the type of code being executed and the computational workload.
- Page Fault Activity—indicates how many and what kind of page faults occurred as a measure of the effectiveness of memory management.
- I/O Activity—indicates how many and what kind of I/O operations are taking place.
- Network Activity—indicates network workload (current number of nodes in the network, number of bytes transmitted and received, number of messages transmitted and received, number of buffers currently in use, number of successful and failing attempts to obtain space for network buffers).
- Response Time Histograms—indicate the time it takes the system to initiate user requests.

Display Utility Program

The Display Utility Program (DISPLAY) provides a dynamic display of system performance measurement statistics on a VT100 or VT52 video display terminal. By typing appropriate commands, system users may list information regarding I/O system activity, paging, CPU usage, current process activity, and other relevant statistics. Figure 3-2 shows a typical screen display.

THE SYSTEM OPERATOR

An operator is any user given the privileges by the system manager to perform operator functions. A system does not require an operator, but it can have one or several operators, and they can use any terminal to issue commands or run programs. Operator functions include:

- system startup and shutdown
- job control (change process priorities, kill jobs, etc.)
- device allocation
- volume mount and dismount request servicing
- on-line disk and magnetic tape volume and file backup
- spool and batch queue control
- software maintenance update installation
- diagnostic execution

An operator uses the command language to control operations, check system status, and run utility programs.

A special system program, the Operator Communications Manager (OPCOM), is the primary operator aid. It collects and delivers the messages all users and user programs send to the operators. Any operator can respond to user requests, and the Operations Communications Manager will remind operators of any outstanding requests.

Spooling and Queue Control

The operators define the number and kind of input and output spool queues in the system. The operators can create output spool queues for any number of devices, including line printers, terminals, or even magnetic tape. The operators can also create input queues for spooling batch input from a card reader.

The operators can assign each queue a priority, merge or redirect queues to other devices, and modify the queue set-up at any time. It is possible to have more than one print queue for the same printer. For example, an operator can create a generic printer queue that will collect jobs that can be printed on any of a set of printers, and at the same time have a print queue for each individual printer. A user can issue a print request for a generic printer or a particular printer, and the operator can override the user's request.

A print job can contain one or more files to be printed together. Print jobs can be submitted by an interactive user, batch job, or any program. Print jobs are also automatically submitted at the end of a batch job. A print job can specify the forms type required, the number of copies of the job, the job priority, and a "hold until given time" request. Each file within a print job has its own copies count,

FREE LIST: 2016 I/O SYSTEM RATES MODIFY LIST: 35
16:17:09

NAME	VALUE	RATE /SEC	AVG RATE	NAME	VALUE	RATE /SEC	AVG RATE
DIRECT I/Os	32	7.30	1.50	PAGE FAULTS	65	14.84	1.83
BUFFERED I/Os	29	6.62	3.24	PAGES READ	4	0.91	0.11
MAILBOX WRITES	0	0.00	0.00	READ I/Os	2	0.45	0.07
WINDOW TURNS	3	0.68	0.14	PAGES WRITTEN	0	0.00	0.00
LOGNAME TRANS	39	8.90	0.98	WRITE I/Os	0	0.00	0.00
FILE OPENS	3	0.68	0.07	TOTAL INSWAPS	0	0.00	0.00

Figure 3-2

Display Utility Program (I/O System Rates)

and each can have these options: double space, inhibit form feed, print a flag page, label each page, or delete after printing. The operator can choose whether or not to print burst (job separator) pages, and can put jobs on indefinite hold, modify the priority of a job, or abort a job.

Batch Processing

The system supports multiple stream, multiple queue batch processing. The operators control how many batch job streams can run concurrently. Batch jobs can be submitted by an interactive user, another batch job, or any program. When the number of batch jobs submitted exceeds the number of streams, the remainder of the batch jobs are held in a batch input queue. As with the spool queues, the operators can control the batch job queue. They can change job priority, hold a job until after a given time, hold a job indefinitely, and kill a job.

Volume mount commands issued in a batch job can request a generic device, such as any disk, or a specific device, such as disk drive unit 2. The batch job waits until an operator satisfies the mount request, while other batch jobs proceed. Operators can find out which job has a given device.

On-line Software Maintenance

An operator can incorporate maintenance updates to the software without bringing down the system for stand-alone use. For example, VAX-11/780 maintenance patches are distributed on floppy disk and the operator simply loads the console floppy disk drive with the maintenance floppy to update the software on disk. Depending on the nature of the update, an operator may have to restart the system to activate the patched modules.

System Recovery

An operator can select manual or automatic system recovery following a power interruption or hardware or software failure.

On automatic system recovery after power interruption, the system determines whether the contents of memory are still valid, and if so, restarts all possible I/O in progress at the time of the power interruption and continues operations from the point of interruption. If the contents of memory are not valid, either because memory battery backup is not included in the configuration, or because the power failure lasted longer than the battery, the system automatically boots itself from disk and executes the start-up command procedures.

Error Logging and Reporting

The error logger is a job that runs continuously. It collects errors detected by both hardware and software, including:

- device errors
- interrupt timeouts
- interrupts received from nonexistent devices
- memory, translation buffer, and cache parity errors

In addition, system software sends complete recovery information to the error logger following a power interruption or hardware or software failure.

The error logger writes all messages it receives into an error log file, noting vital system statistics at the time of the message. The error logger also notes benign events when they occur, such as when volumes are mounted and dismounted, and periodic time stamps indicating that no entries have occurred for a specified period of time. The error logger can accept messages from the operators at any time, and from any programs privileged to send messages to the error logger.

The system includes an error report generating program that converts the information in the log file into a text file that can be printed for later study.

On-line Diagnostics

An operator can run diagnostics to check the operation of both hardware and software. An operator can run system exercisers and device verification diagnostics while normal operations proceed. System exercisers test general purpose software and compare the results with known answers, reporting any discrepancies to the error logger.

Operators can run device verification diagnostics either stand-alone or concurrent with other processes. Diagnostics check the peripheral functions, including disk head alignment. In addition, fault isolation diagnostics, which isolate problems to replaceable units, are available for stand-alone use.

Remote Diagnosis

If the system is equipped with the remote diagnosis option, an operator sets up the system for remote preventive maintenance or troubleshooting. When a hardware error is detected or suspected, the operator mounts a diagnostic disk pack, loads a diagnostic floppy disk in the console, sets a switch on the processor console, and calls the local DIGITAL service office. An operator does not need to be present at the installation once the call is made. The DIGITAL Diagnostic Center can then connect to the installation, run automated diagnostics, operate the diagnostic console manually, and check the error log file. If a problem is found, the Field Service engineer can bring the proper equipment and replacement modules to make the repairs.

THE USER ENVIRONMENT TEST PACKAGE

The User Environment Test Package (UETP), consists of a series of tests designed to demonstrate that the hardware and software components of a system are in working order. The UETP consists of six phases:

- The Initialization Phase—This phase verifies that I/O devices are operational via simple read/write operations. In addition, users are prompted to supply several parameters which define the scope of the test, e.g.,number of users to be simulated by UETP, amount of information to be displayed at the console, and number of consecutive runs to be made by the UETP.
- The I/O Device Test Phase—In this phase, I/O devices undergo comprehensive testing. Terminals and line printers generate pages or screens of output containing header information and a test pattern of ASCII characters. Disks and magnetic tapes are also exercised. Files are created on the mounted volumes. Data are then written to the files. The test then checks the written data for errors and erases the files.
- The Native Mode Phase—This phase includes three tests, each of which exercises software services provided explicitly for VAX/VMS. The first exercises VAX/VMS system services; the second exercises native mode utilities such as the Symbolic Debugger and Image File Patch Utility; and the third exercises VAX-11 RMS.
- The System Load Test Phase—This phase creates a number of detached processes which simulate the action of a group of users concurrently issuing commands from terminals; it tests the system's ability to handle various levels of utilization.
- The Compatibility Mode Test Phase—This phase tests most RSX-11M utilities running in compatibility mode on VAX/VMS.
- Termination Phase—In this phase, temporary files are deleted and other cleanup activities are performed. If miltiple runs were requested during the initialization phase, then the UETP is restarted and control is passed directly to the device test phase.

The UETP is invoked via command procedures. The entire package may be specified by executing the master procedure, UETP.COM, or tests may be executed individually by specifying particular command procedures as illustrated in Figure 3-3.

```
$ RUN  UETINIT00          Initialization Phase
$ RUN  UETINIT01
$ RUN  UNETPDEV01         I/O Device Test Phase
$ @UETCOMP00              Compatibility Mode Test Phase
$ RUN  UETNATV01
$ @UETNATV02              Native Mode Test Phase
$ @UETNRMS00
$ RUN  UETLOAD01          System Load Test Phase
$ RUN  UETTERM01          Termination Phase
```

Figure 3-3

UETP Command Procedures

APPLICATIONS EXAMPLES

To illustrate how the multiprogramming capabilities of the system can be effective in widely diverse applications, Figures 3-4 and 3-5 show two hypothetical application systems: a commercially oriented data processing system and a real-time flight training simulation system.

Commercial System Example

The commercial system diagram (Figure 3-4) begins with both a programming group and a data processing operations group. Within the programming group, jobs can be performing requests for programmers at terminals who are using the system's text editors, compilers, and linkers to write and test programs for both the VAX/VMS and an RSX-11M system in the manufacturing department. The programmers can execute command procedures and submit batch jobs to automate repetitive development steps.

Within the operations group, the system's operators can be managing batch and spool queues, backing up disks, and monitoring performance. They may be down-line loading tasks into the RSX-11M system. They may be running an accounting program that interprets and summarizes the accounting statistics collected by the operating system during the period and sends that data to the business data processing subsystem.

A billing process within the business data processing group can be suspended until it is activated by a process (such as the accounting process) that wants to send it bill-

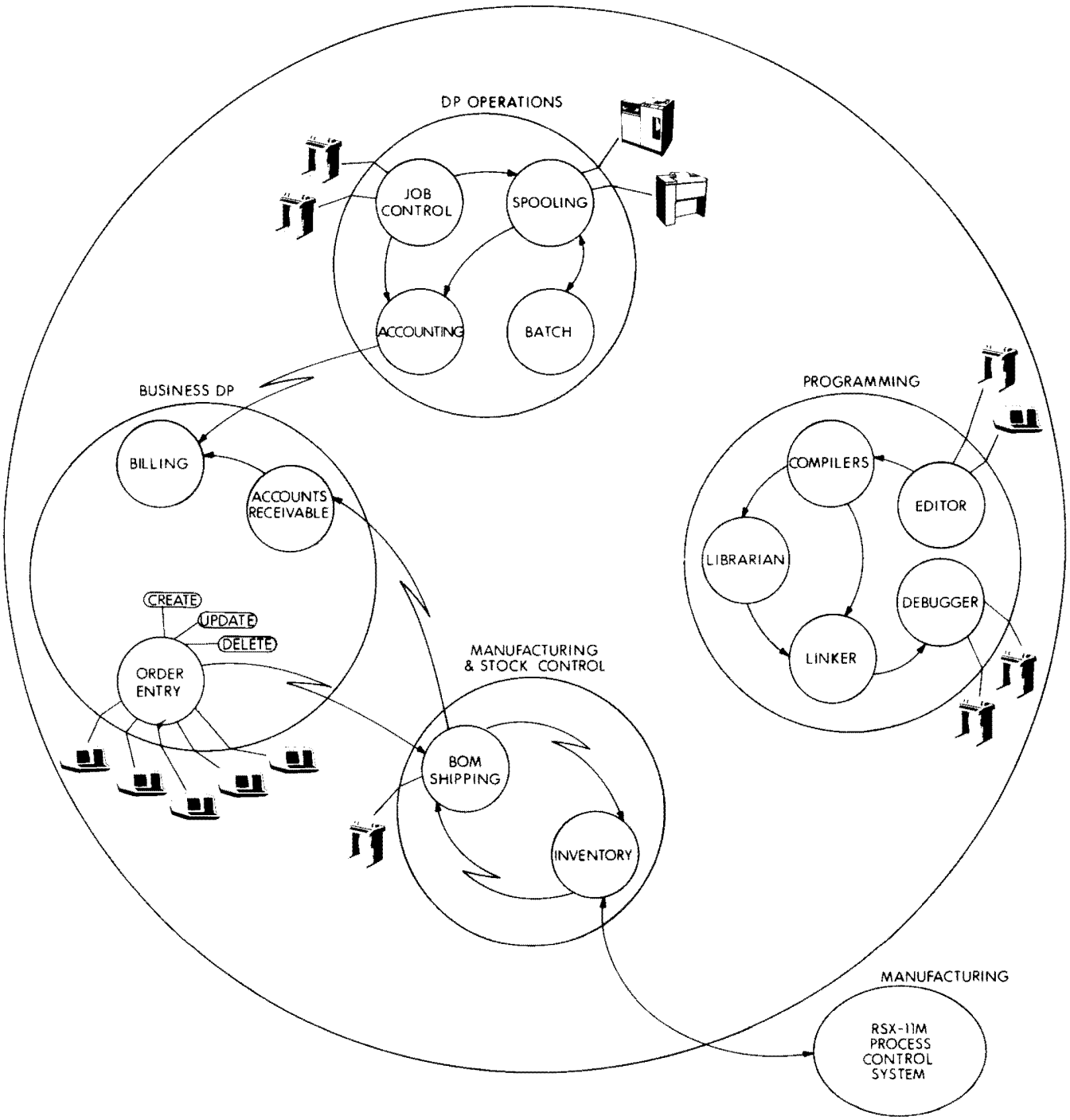


Figure 3-4

Commercial Data Processing System

ing information. The accounting process can send to the billing process the name of an account summary file that it created, using a permanent mailbox defined for that purpose.

The business data processing group may also run a job that handles order entry terminals. The job can consist of a controlling process that handles input from the terminals, and several subprocesses that perform the actual record processing functions. As users at the terminals enter their requests, such as create order record, update order record, etc., the controlling process collects the input from a terminal and sends the request to the appropriate subprocess's mailbox. The subprocess may be hibernating, having requested the operating system to activate it when

VAX-11/780 SAMPLE APPLICATION SYSTEM
FLIGHT TRAINING SIMULATION

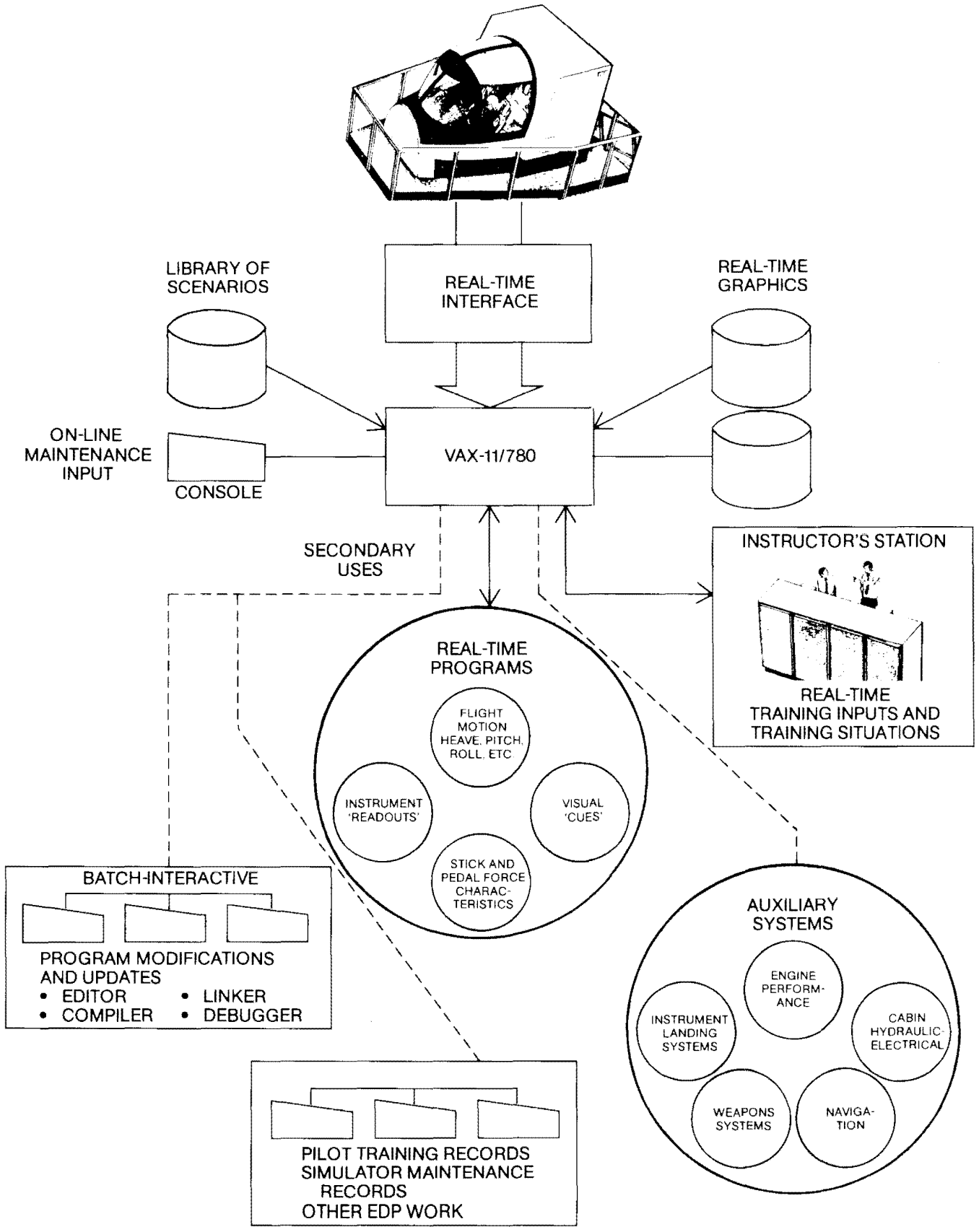


Figure 3-5
Real-time Flight Simulation System

anything is written to its mailbox. Or the controlling process can simply set an event flag to notify a subprocess that it has received a request for which the subprocess is waiting.

A process in the manufacturing application group may regularly collect orders from the order entry job to create materials parts lists, or notify stockroom clerks of high-priority orders. The stockroom clerks can keep inventory records up to date by using the shipping and inventory jobs, because they can collect manufacturing statistics from a background task in the RSX-11M process control system on the manufacturing floor. Once orders are shipped, the shipping process can notify an accounts receivable process in the business data processing application group, which in turn can activate the billing process.

Real-Time Flight Simulation Example

To illustrate how VAX's facilities can be as extended to the real-time environment, Figure 3-5 shows a sample flight training simulation system. Flight simulation is a particularly good application for the VAX system, since in addition to fast real-time response, such systems must also be capable of solving large and complex equation systems. In addition, such systems also require general program development facilities such as FORTRAN compilation, assembly, editing, debug, library facilities, and fast-access file management.

The illustrated system shows how the multiprogramming capabilities of VAX allow it to handle the basic real-time tasks of data acquisition and transmission, while also performing a wide range of other activities:

Looking from top to bottom, the diagram begins with a representation of an aircraft fuselage containing the cockpit throttle levers and control panel which the student operates to simulate flight. The signals and control movements of the student go through an analog to digital conversion, and are then passed through a real-time interface device into VAX memory. Before the system can respond to the data generated by the student, complex flight-motion equations must be called and combined with current aircraft data. This, in turn, produces a new set of circumstances with which the student must deal.

Additional inputs to the system are also provided by an instructor at a timesharing terminal (shown in the right central part of the diagram). By typing commands at the terminal, the instructor can control the situations which the student must face—for example, by injecting an engine failure, weather change, or some other complication. The instructor can also introduce additional variables into the system by inputting predefined "scenarios" which have been stored on libraries. The student's flight environment can include "visual cues" (e.g., terrain, runway approaches, other aircraft, etc.) produced by sophisticated, real-time graphics modules.

In addition to performing basic flight simulation functions, the system also performs a number of auxiliary functions which are less time-critical in nature. These would include such activities as monitoring of engine performance, testing of navigation and instrument landing systems, testing of weapons systems, and monitoring cabin, hydraulic, and electrical systems. The system also generates a file of student performance statistics which can be analyzed at a later time.

The system also provides facilities for program development. As shown in the lower left-hand side of the diagram, programmers at terminals can write and test new applications programs (in either batch or interactive mode) using text editors, compilers, linkers, and debuggers. Because of the way the VAX architecture handles real-time vs. normal processes (described below), program development can take place simultaneously with the running of the flight simulator; no significant reduction in real-time processing speed will result.

Design Considerations

Systems such as that shown in Figure 3-5 must be designed to operate at extremely high speeds, both in terms of real-time I/O and computation. Many simulators require turnaround times (data sampling, computation, and output) in the 25-50 millisecond range.

There are several elements of the VAX hardware and the VAX/VMS operating system which aid in achieving such speeds. Most important is VAX's context switching ability—a result of special processor instructions which relieve the operating system software of having to individually load or save the hardware registers which define the hardware context. Another element is the design and operation of VAX/VMS device drivers. VAX/VMS drivers are forked processes which are created dynamically in response to a user I/O request or unsolicited device interrupt. They operate with minimal context, execute to completion when invoked, and remain memory resident throughout execution. (VAX/VMS device drivers can be written for user-developed devices which interface to the VAX UNIBUS.)

In addition, system designers can utilize the VAX/VMS scheduling and system service facilities to yield still higher processing speeds. For example, the following design schema could be employed.

Those processes which perform the basic flight simulation functions (i.e., flight motion equations, visual graphics modules, etc.) can be designed as a group of hibernating processes capable of being immediately reawakened as required by the student's control activities. This could be accomplished via the use of system service calls such as ($HIBERNATE/$WAKE) or ($SUSPEND/$RESUME) or by using interprocess control structures such as mailboxes and common event blocks.

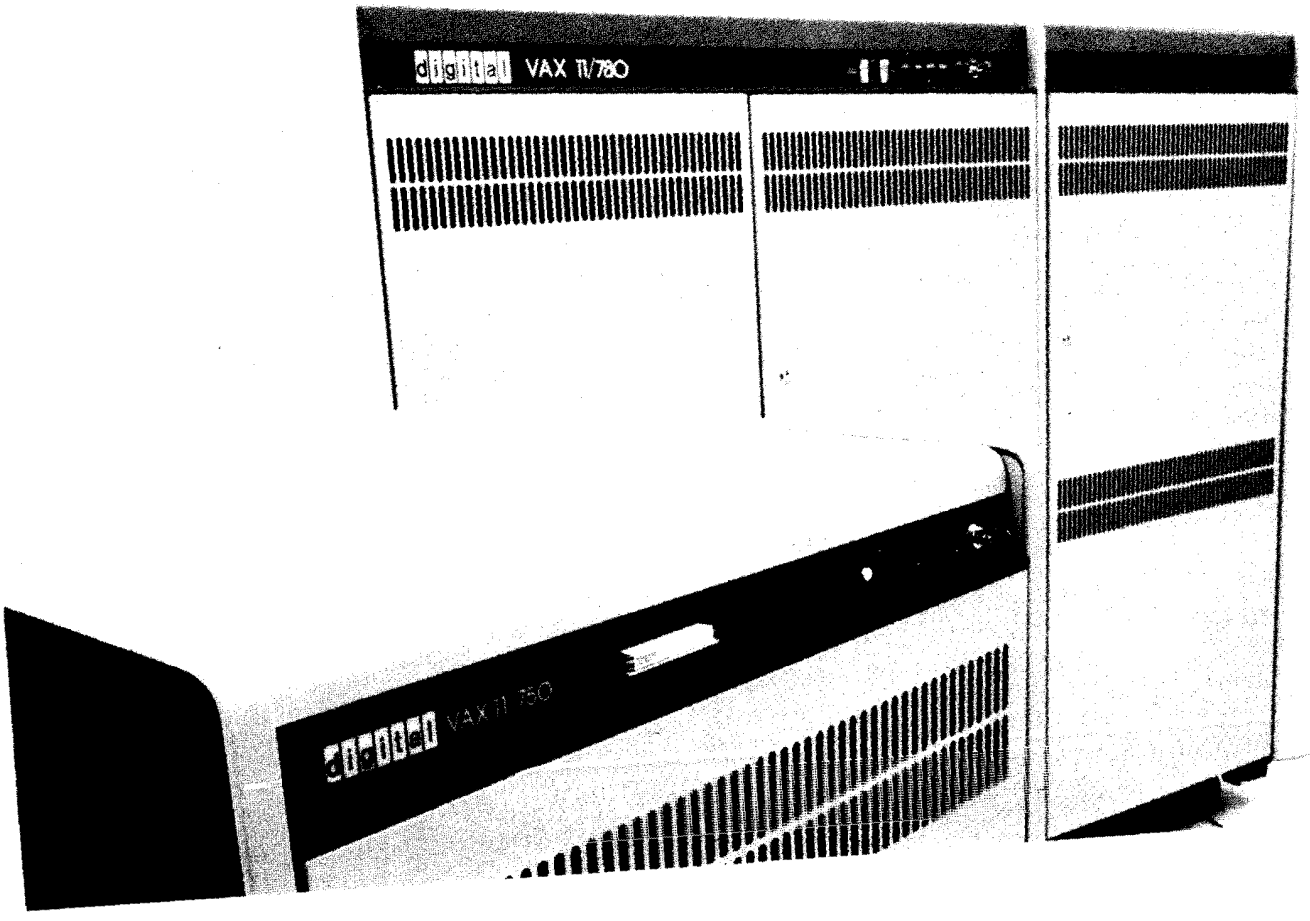
To further ensure that basic simulation functions will operate at the fastest possible speed, real-time processes can be granted the highest scheduling priorities. Such processes can also be given special privileges which allow them to eliminate paging and swapping and thus assure memory residency. (Note that when a VAX process runs at real-time priority its priority level will actually be higher than system processes such as the Swapper, Linker, and Symbionts).

Processes that are not time-critical can be assigned normal scheduling priorities. Those processes which perform what is essentially a monitoring function (e.g., engine performance, electrical system, etc.) can also be hibernated, then monitored periodically via the issuance of a programmed system timer service. Other activities such as

program development and statistical processing can take full advantage of the VAX/VMS system with minimum impact upon the basic real-time core of the simulation system.

For more information on the concepts of jobs, processes and program images, refer to the Operating System section.

4
The VAX Processors



A VAX processor executes variable-length instructions in native mode and non-privileged PDP-11 instructions in compatibility mode. The VAX processor includes integral memory management, sixteen 32-bit registers, 32 interrupt priority levels, an intelligent console, a programmable real-time clock, and a time-of-day and date clock.

The VAX native instruction set provides 32-bit addressing enabling the processor to address up to 4 billion (10^9) bytes of virtual address space. The processor's memory management hardware includes mapping registers used by the operating system, page protection by access mode, and an address translation buffer that eliminates extra memory accesses during virtual to physical address translation.

VAX also provides sixteen 32-bit general registers that can be used for temporary storage, as accumulators, index registers, and base registers. Four registers have special significance: the Program Counter and three registers that are used to provide an extensive procedure CALL facility. The processor offers a variety of addressing modes that use the general registers to identify instruction operand locations, including an indexed addressing mode that provides a true post-indexing capability.

The native instruction set is highly bit efficient. It includes integral decimal, character string, and floating point instructions, as well as integer, logical, and bit field instructions. Instructions and data are variable length and can start at any arbitrary byte boundary or, in the case of bit fields, at any arbitrary bit in memory. Floating point instruction execution can be enhanced by an optional floating point accelerator.

The I/O subsystem consists of the processor's internal bus and the UNIBUS and MASSBUS interfaces.

INTRODUCTION

This section is divided into two parts. The first discusses the architecture of a VAX processor, and the second discusses VAX processor implementation details.

VAX ARCHITECTURE

The following sections discuss the architecture, i.e., the programming characteristics of the processing system as seen from the general user's and the operating system's viewpoint.

PROCESSING CONCEPTS FOR USER PROGRAMMING

A program is a stream of instructions and data that a user can request the operating system to translate, link, and execute. An executable program is called an **image** to distinguish it from source and object programs. When a user runs an image, the context in which the image is executed is called a **process**. A process is the complete unit of execution in this computer system and typically runs several images, one after another. Process context includes the state of the image it is currently executing and includes the image's allowable limitations, which primarily depend on the privileges of the user executing the image.

The next few pages introduce some of the concepts that concern assembly language programmers in general, including addressing, data types, instruction sets, and other programming aspects of the processor. Further details on these programming characteristics follow this introduction.

Process Virtual Address Space

Most data are located in memory using the address of an 8-bit byte. The programmer uses a 32-bit virtual address to identify a byte location. This address is called a *virtual address* because it is not the real address of a physical memory location. It is translated into a real address by the processor under operating system control. A virtual address is not a unique address of a location in memory, as are physical memory addresses. Two programs using the same virtual address might refer to two different physical memory locations, or refer to the same physical memory location using different virtual addresses.

The set of all possible 32-bit virtual addresses is called virtual address space. Conceptually, virtual address space can be viewed as an array of byte "locations" labelled from 0 to $2^{32}-1$, an array that is approximately four billion bytes in length. This address space is divided into sets of virtual addresses designated for certain uses. The set of virtual addresses designated for use by a process, including an image it executes, is one half of the total virtual address space. Addresses in the remaining half of virtual address space are used to refer to locations maintained and protected by the operating system.

Instruction Sets

At any one time, the processor's instruction interpretation hardware can be set to either of two modes: native mode or compatibility mode. In native mode the processor executes a large set of variable-length instructions, recognizes a variety of data types, and uses sixteen 32-bit general purpose registers. In compatibility mode the processor executes a set of PDP-11 instructions, recognizes integer data, and uses eight 16-bit general purpose registers. While native mode is the primary instruction execution state of the machine and compatibility mode the secondary state, their instruction sets are closely related, and their programming characteristics are very similar. A user process can execute both native mode images and compatibility mode images.

A native instruction consists of an operation code (opcode) and zero or more operands, which are described by data type and addressing mode. The native instruction set is based on over 200 different kinds of operations, of each operand of which can be addressed in any one of several ways. Thus, the native instruction set offers a tremendous number of instructions from which to choose.

In spite of the large number of instructions, the native instruction set is a natural programming language that is very easy to learn. Many of the instructions correspond directly to high-level language statements, and the assembler mnemonics are readily associated with the instruction function.

To choose the appropriate instruction, it is only necessary to become familiar with the operations, data types, and addressing modes. For example, the ADD operation can be applied to any of several sizes of integer, floating point, or packed decimal operands, and each operand can be addressed directly in a register, directly in memory, or indirectly through pointers stored in registers or memory locations.

Registers and Addressing Modes

A register is a location within the processor that can be used for temporary data storage and addressing. The assembly language programmer has sixteen 32-bit general registers available for use with the native instruction set. Some of these registers have special significance. For example, one register is designated as the Program Counter, and it contains the address of the next instruction to be executed. Three general registers are designated for use with procedure linkages: the Stack Pointer, the Argument Pointer, and the Frame Pointer.

An instruction operand can be located in main memory, in a general register, or in the instruction stream itself. The way in which an operand location is specified is called the operand *addressing mode*. The processor offers a variety of addressing modes and addressing mode optimizations. There is one addressing mode that locates an operand in a register. There are six addressing modes that locate an operand in memory using a register to:

- point to the operand
- point to a table of operands
- point to a table of operand addresses

Additionally, there are six addressing modes that are indexed modifications of the addressing modes that locate an operand in memory. Finally, there are two addressing modes that identify the location of the operand in the instruction stream: one for constant data, and one for branch instruction addresses.

Data Types

The data type of an instruction operand identifies how many bits of storage are to be treated as a unit, and what

the interpretation of that unit is. The processor's native instruction set recognizes four primary data types: integer, floating point, packed decimal, and character string. For each of these data types, the selection of operation immediately tells the processor the size of the data and its interpretation. The processor can also manipulate a fifth data type, the bit field, in which the user defines the size of the field and its relative position. In addition, the processor supports two types of linked-list queue structures.

There are several variations on the four primary data types. Table 4-1 provides a summary of the data types available, each of which are illustrated in Figure 4-1. Integer data are stored as a binary value in either byte, word, longword, or, in some cases, in quadword or octaword format. A byte is eight bits, a word is two bytes, a longword is four bytes, a quadword is eight bytes, and an octaword is sixteen bytes. The processor can interpret an integer as either a signed (2's complement) value or an unsigned value. The sign is determined by the high-order bit.

Floating point values are stored using a signed exponent and a binary normalized fraction. The normalization bit is not represented. Four types of floating point data formats are provided. The two PDP-11 compatible formats (F_floating and D_floating) are standard on all VAX family processors. Two extended range formats (G_floating and H_floating) are available as options on VAX family processors. F_floating and D_floating are 4 and 8 bytes long respectively, with an 8-bit excess 128 exponent. The effective 24-bit fraction of F_floating yields approximately 7 decimal digits of precision. The 56-bit fraction of D_floating yields approximately 16 decimal digits of precision. G_floating is also 8 bytes in length, but has an 11-bit excess 1024 exponent and effectively 53 bits of fraction. Its precision is approximately 15 decimal digits. H_floating is 16 bytes in length with a 15-bit excess 16384 exponent and 113-bit fraction. Its precision is approximately 33 decimal digits.

Packed decimal data are stored in a string of bytes. Each byte is divided into two 4-bit nibbles. One decimal digit is stored in each nibble. The first, or high-order, digit is stored in the high-order nibble of the first byte, the second digit is stored in the low-order nibble of the first byte, the third digit is stored in the high-order nibble of the second byte, and so on. The sign of the number is stored in the low-order nibble of the last byte of the string.

Character data are simply a string of bytes containing any binary data, for example, ASCII codes. The first character

Table 4-1
Data Types

DATA TYPE	SIZE	RANGE (decimal)	
Integer		Signed	Unsigned
Byte	8 bits	-128 to $+127$	0 to 255
Word	16 bits	-32768 to $+32767$	0 to 65535
Longword	32 bits	-2^{31} to $+2^{31}-1$	0 to $2^{32}-1$
Quadword	64 bits	-2^{63} to $+2^{63}-1$	0 to $2^{64}-1$
Octaword	128 bits	-2^{127} to $+2^{127}-1$	0 to $2^{128}-1$
F_and D_floating point		$\pm 2.9 \times 10^{-37}$ to 1.7×10^{38}	
F_floating point	32 bits	approximately seven decimal digits precision	
D_floating	64 bits	approximately sixteen decimal digits precision	
G_floating point		$\pm 0.56 \times 10^{-308}$ to 0.9×10^{308}	
G_floating	64 bits	approximately fifteen decimal digits precision	
H_floating point		$\pm 0.84 \times 10^{-4932}$ to $\pm 0.59 \times 10^{4932}$	
H_floating	128 bits	approximately thirty-three decimal digits precision	
Packed Decimal String	0 to 16 bytes (31 digits)	numeric, two digits per byte sign in low half of last byte	
Character String	0 to 65535 bytes	one character per byte	
Variable-length Bit Field	0 to 32 bits	dependent on interpretation	
Numeric String	0 to 31 bytes (DIGITS)	$-10^{31}-1$ to $+10^{31}-1$	
Queue	$\geq$ 2 longwords/ queue entry	Queue entries at arbitrary displacement in memory	

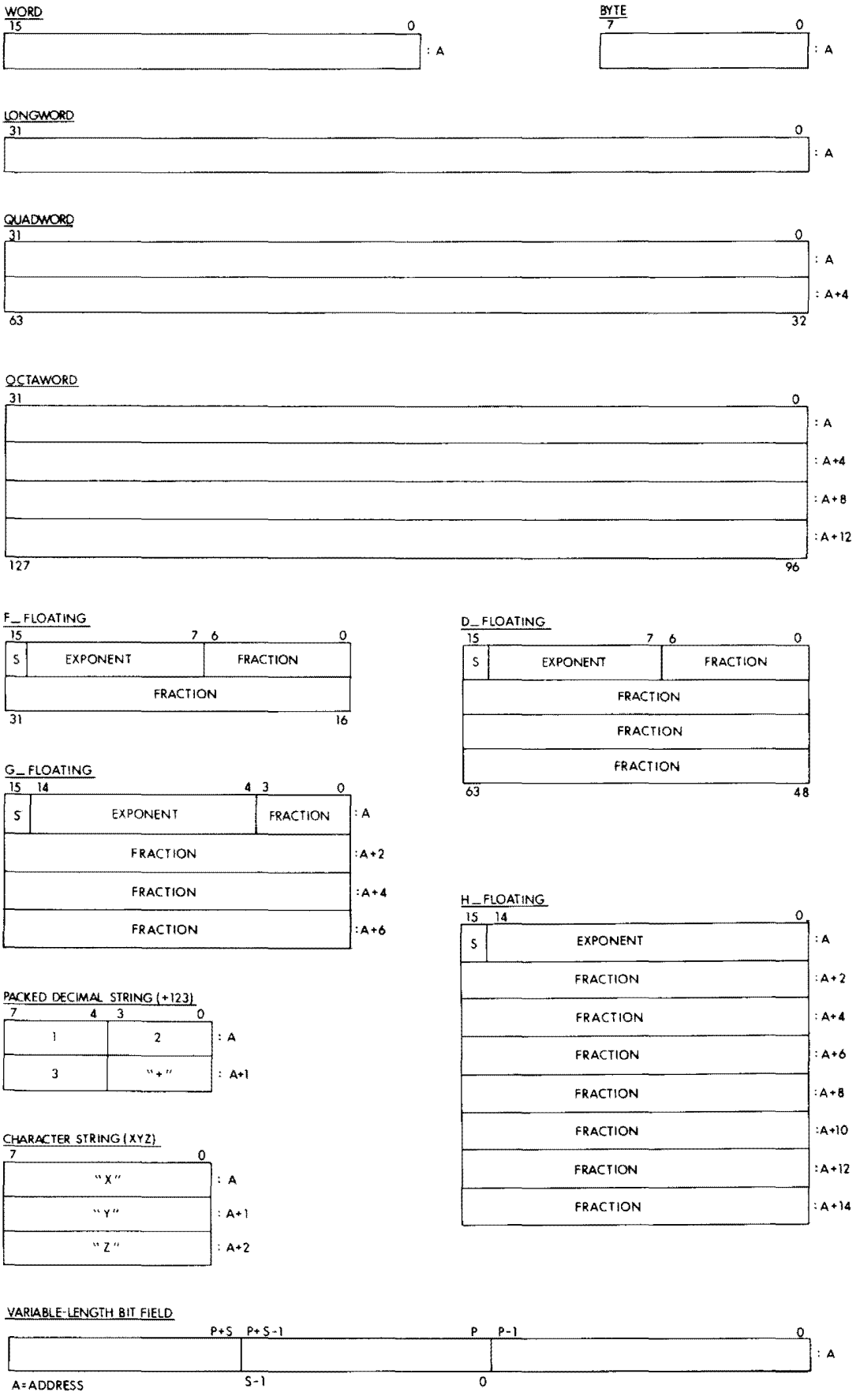


Figure 4-1
Data Type Representations

in the string is stored in the first byte, the second character is stored in the second byte, and so on. A character string that contains ASCII codes for decimal digits is called a numeric string.

The address of any data item is the address of the first byte in which the item resides. All integer, floating point, packed decimal, and character data can be stored starting on an arbitrary byte boundary. A bit field, however, does not necessarily start on a byte boundary. A field is simply a set of contiguous bits (0-32) whose starting bit location is identified relative to a given byte address or register. The native instruction set can interpret a bit field as a signed or unsigned integer.

The VAX queue data types consist of circular, doubly linked lists. A queue entry is specified by its address. Each queue entry is linked to the next entry via a pair of longwords. The first longword is the forward link; it specifies the location of the succeeding entry. The second longword is the backward link; it specifies the location of the preceding entry. VAX processors support two queue types according to the nature of the links: absolute and self-relative. An absolute link contains the absolute address of the entry that it points to. A self-relative link contains a displacement from the address of the queue entry.

Stacks, Subroutines, and Procedures

A stack is an array of consecutively addressed data items that are referenced on a last-in, first-out basis using a general register. Data items are added to and removed from the low address end of the stack. A stack grows toward lower addresses as items are added, and shrinks toward higher addresses as items are removed.

A stack can be created anywhere in the user's program address space. Any register can be used to point to the current item on the stack. The operating system, however, automatically reserves portions of each process address space for stack data structures. User software references its stack data structure, called the **user stack**, through a general register designated as the Stack Pointer. When the user runs a program image, the operating system automatically provides the address of the area designated for the user stack.

A stack is an extremely powerful data structure because it can be used to pass arguments to routines efficiently. In particular, the stack structure supports re-entrant routines because the processor can handle routine linkages automatically using the Stack Pointer. Routines can also be recursive because arguments can be saved on the stack for each successive call of the same routine.

The processor provides two kinds of routine call instructions: those for **subroutines**, and those for **procedures**. In general, a subroutine is a routine entered using a Jump to Subroutine or Branch to Subroutine instruction, while a procedure is a routine entered using a Call instruction.

The processor provides more elaborate routine linkages for procedures than for subroutines. The processor automatically saves and restores the contents of registers to be preserved across procedure calls. The processor provides two methods for passing argument lists to called procedures: by passing the arguments on the stack, or by passing the address of the arguments elsewhere in memory. The processor also constructs a "journal" of procedure call nesting by using a general register as a pointer to the place on the stack where a procedure has its linkage data. This record of each procedure's stack data, known as its **stack frame**, enables proper returns from procedures even when a procedure leaves data on the stack. In addition, user and operating system software can use the stack frame to trace back through nested calls to handle errors or debug programs.

Condition Codes

A user program can test the outcome of an arithmetic or logical operation. The processor provides a set of condition codes and branch instructions for this purpose. The condition codes indicate whether the previous arithmetic or logical operation produced a negative or zero result, or whether there was a carry or borrow, or an overflow. There are a variety of branch on condition instructions: those for overflow and carry or borrow, and those for signed and unsigned relational tests.

Exceptions

Certain situations may require that the results of an operation be tested either by the user or by the processor directly. The processor recognizes many events for which it must test directly, and automatically changes the normal flow of the user program when they occur. These events, called **exceptions**, are the direct result of executing a specific instruction. Exceptions also include errors automatically detected by the processor, such as improperly formed instructions.

All exceptions trap to operating system software. There are essentially no fatal exceptions. All exceptions either wait for the instruction that caused them to complete before trapping or they restore the processor to the state it was in just prior to executing the instruction that caused the exception. In either case, the instruction can be retried after the cause of the exception is cleared. Depending on the exception, it may be desirable to correct the situation and continue. If not, the operating system issues an appropriate message and aborts the instruction stream in progress. To continue, the user can request the operating system software to start execution of a condition handler automatically when an exception occurs.

USER PROGRAMMING ENVIRONMENT

A process context includes the definition of the virtual address space in which it executes an image. An image executing within a process context controls its execution through the use of one of the instruction sets, the general purpose registers, and the Processor Status Word. These hardware resources are discussed in detail in the following sections.

Process Virtual Address Space Structure

As mentioned earlier, certain sets of virtual addresses in virtual address space are designated for particular uses. The processor and operating system provide a multiprogramming environment by dividing virtual address space into two halves: one half for addressing context-dependent code and data, the other half for addressing context-independent code and data.

The first half is called **per-process space** (or more simply, "process space"), which is capable of addressing approxi-

mately two billion bytes. An image executing in the context of a process and the operating system on behalf of the process use addresses in process space to refer to code and data particular to that process context. A process cannot represent virtual addresses in any process space but its own. Thus, code and data belonging to a process are automatically protected from other processes in the system.

The second half of virtual address space is called **system space**. The operating system assigns specific meanings to addresses in system space. The significance of any address in system space is the same for every process, independent of process context. Although most locations referred to by system space addresses are protected from access by user images, if two images executing in different process contexts do use an address in system space, the address always refers to the same physical location in memory.

Process space is further subdivided into two equal regions. Addresses in the first region, called the **program region**, are used to identify the location of image code and data. Addresses in the second region, called the **control region**, are used to refer to stacks and other temporary program image and permanent process control information maintained by the operating system on behalf of the process. Program region addresses are allocated from address 0 up, and control region addresses are allocated from address $2^{31}-1$ down.

System space is also subdivided into two equal regions. The operating system assigns the **system region** addresses for linkages to its service procedures, for memory management data, and for I/O processing routines. The second region is presently unused.

General Registers

Instruction operands are often either stored in the processor's general registers or accessed through them. The sixteen 32-bit programmable general registers are labelled R0 through R15 (in decimal). Registers can be used for temporary storage, accumulators, base registers, and index registers. A base register contains the address of the base of a software data structure such as a table, and an index register contains a logical offset into a data structure.

Whenever a register is used to contain data, the data are stored in the register in the same format as would appear in memory. If a quadword or double floating datum is stored in a register, it is actually stored in two adjacent registers. For example, storing a double floating number in register R7 loads both R7 and R8.

Some registers have special significance depending on the instruction being executed. Registers R12 through R15 have special significance for many instructions, and therefore have special labels. These special registers are:

- The Program Counter (PC or R15), which contains the address of the next byte to be processed in the instruction stream.
- The Stack Pointer (SP or R14), which contains the address of the top of a stack maintained for subroutine and procedure calls.
- The Frame Pointer (FP or R13), which contains the address of the base of a software data structure stored on the stack called the stack frame, maintained for procedure calls.
- The Argument Pointer (AP or R12), which contains the address of the base of a software data structure called the argument list, maintained for procedure calls.

In addition, the first six registers, R0 through R5, have special significance for character and packed decimal string instructions, and the Cyclic Redundancy Check and Polynomial Evaluation instructions. These instructions use these registers to store temporary results and, upon completion, leave results in the registers that a program can use as the operands of subsequent instructions.

A register's special significance does not preclude its use for other purposes, except for the Program Counter. The Program Counter can not be used as an accumulator, as a temporary register, or as an index register. In general, however, most users do not use the Stack Pointer, Argument Pointer, or Frame Pointer for purposes other than those designated.

Addressing Modes

The processor's addressing modes allow almost any operand to be stored in a register or in memory, or as an immediate constant. Table 4-2 summarizes the addressing modes.

There are seven basic addressing modes that use the general registers to identify the operand location. They include:

- Register mode, in which the register contains the operand.
- Register Deferred mode, in which the register contains the address of the operand.
- Autodecrement mode, in which the contents of the register are first decremented by the size of the operand, and then used as the address of the operand. The size of the operand (in bytes) is given by the data type of the instruction operand, and depends on the instruction. For example, the Clear Word instruction uses a size of two, since there are two bytes per word.
- Autoincrement mode, in which the contents of the register are used as the address of the operand, and then incremented by the size of the operand. If the Program Counter is the specified register, the mode is called Immediate mode.
- Autoincrement Deferred mode, in which the contents of the register are used as the address of a location in memory containing the address of the operand, and then are incremented by four (the size of an address). If the Program Counter is the specified register, the mode is called Absolute mode.
- Displacement mode, in which the value stored in the register is used as a base address. A byte, word, or longword signed constant is added to the base address, and the resulting sum is the effective address of the operand.

Table 4-2
Addressing Modes: Assembler Syntax

Literal (Immediate)	{S^ / I^} *#constant*	
Register	**R***n*	
Register Deferred	(**R***n*)	Indexed [**Rx**]
Autodecrement	-(**R***n*)	
Autoincrement	(**R***n*)+	
Autoincrement Deferred	@(**R***n*)+	
(Absolute)	@*#address*	
Displacement	{B^ / W^ / L^} *displacement*(**R***n*) *address*	
Displacement Deferred	@{B^ / W^ / L^} *displacement*(**R***n*) *address*	

n = 0 through 15
x = 0 through 14

- Displacement Deferred mode, in which the value stored in the register is used as the base address of a table of addresses. A byte, word, or longword signed constant is added to the base address, and the resulting sum is the address of the location that contains the actual address of the operand.

Of these seven basic modes, all except register mode can be modified by an index register. When an index register is used with a basic mode to identify an operand, the addressing mode is the name of the basic mode with the suffix "indexed." For example, the indexed addressing mode for register deferred is called "register deferred indexed" mode. In addition to the seven basic addressing modes that use registers, the processor recognizes six indexed addressing modes.

In an indexed addressing mode, one register is used to compute the base address of a data structure, and the other register is used to compute an index offset into the data structure. To obtain the operand's effective address in an indexed addressing mode, the processor: 1) computes the base operand address provided by one of the basic addressing modes (except register mode), 2) takes the value stored in the index register and multiplies it by the given operand size, and 3) adds the resultant value to the operand address. The index register contents are not affected and can be used for subsequent processing operations.

The processor also provides literal mode addressing, in which an unsigned 6-bit field in the instruction is interpreted as an integer or floating point constant.

The variety of addressing modes enables the assembly language programmer to write, and high-level language compilers to produce, very compact code. For example, literal mode is a very efficient way to specify small constants. The 6-bit field is interpreted as an integer when used with integer operations, and can therefore express the constants 0 through 63 (decimal). The 6-bit field is interpreted as a floating point constant when used in floating point operations, where three bits express an exponent and three a fraction.

The autoincrement and autodecrement modes enable automatic stepping through tables. Displacement mode enables generation of offsets into a table, with a choice of either short or long displacements. The deferred modes enable the user to maintain tables of operand addresses instead of the operands themselves.

The indexed addressing modes allow indexing into tables with a step size automatically determined by the operand. As in autoincrement and autodecrement addressing, the index is calculated in the context of the operand data type. This means that the user can easily access several tables of differing data types using the same index key.

Furthermore, because the indexed addressing modes enable specification of the base operand address using any mode that generates an actual address (that is, any mode except register or literal), the user has the ability to construct double indexing. The base address can be selected from a table of base addresses using displacement deferred mode, and then use an index register to provide the offset into the particular table selected.

Thus the processor's addressing modes allow considerable flexibility in the arrangement and processing of data structures. A data structure's design does not have to be tied to its processing method for efficiency.

Program Counter

A native mode instruction has a variable-length format, and instructions are thought of as byte aligned. A variable-length format not only makes code more compact, it

means that the instruction set can be extended easily. The opcode for the operation is either one or two bytes, and is followed by zero to six operand specifiers, depending on the instruction. An operand specifier can be one or several bytes long, depending on the addressing mode. Figure 4-2 illustrates the representation of an instruction as a string of bytes. Just before the processor begins to execute an instruction, the Program Counter contains the address of the first byte of the next instruction. The way in which the Program Counter is updated is totally transparent to the programmer.

The Program Counter itself can be used to identify operands. The assembler translates many types of operand references into addressing modes using the Program Counter. Autoincrement mode using the Program Counter, or **immediate mode,** is used to specify in-line constants other than those available with literal mode addressing. Autoincrement deferred mode using the Program Counter, or **absolute mode,** is used to reference an absolute address. Displacement and displacement deferred modes using the Program Counter are used to specify an operand using an offset from the current location.

Program Counter addressing enables the user to write position-independent code. Position-independent code can be executed anywhere in virtual address space after it has been linked, since program linkages can be identified as absolute locations in virtual address space and all other addresses can be identified relative to the current instruction.

Stack Pointer, Argument Pointer and Frame Pointer

The Stack Pointer is a register specifically designated for use with stack structures. Autodecrement mode addressing using the Stack Pointer can be used to place items on the stack. Auoincrement mode can be used to remove items from the stack. To reference and modify the top element on a stack without removing it, use register deferred mode, and to reference other elements of the stack use displacement mode addressing.

The processor designates Register 14 as the Stack Pointer for use with both the subroutine Branch or Jump instructions, and the procedure Call instructions. On routine entry, the processor automatically saves the address of the instruction that follows the routine call on the stack. It uses the Program Counter and the Stack Pointer to perform the operation. Before entering the subroutine, the Program Counter contains the address of the instruction following the Branch, Jump, or Call instruction; the Stack Pointer contains the address of the last item on the stack. The processor pushes the contents of the Program Counter on the stack. Returning from a subroutine, the processor automatically restores the Program Counter by popping the return address off the stack.

Also for the procedure Call instructions, the processor designates Register 12 as an Argument Pointer, and Register 13 as a Frame Pointer. The Argument Pointer is used to pass the address of the argument list to a called procedure, and the Frame Pointer is used to keep track of nested Call instructions.

An argument list is a formal data structure containing the arguments required by the procedure being called. Arguments may be actual values, addresses of data structures, or addresses of other procedures. An argument list can be passed in either of two ways: by passing only its address, or by passing the entire list on the user stack. The first method is used to pass long argument lists, or lists that are to be preserved. The second method is generally used when calling procedures that do not require arguments, or when building an argument list dynamically.

When issuing a procedure Call instruction, the processor uses the Argument Pointer to pass arguments to the procedure. If arguments were passed on the stack, the proc-

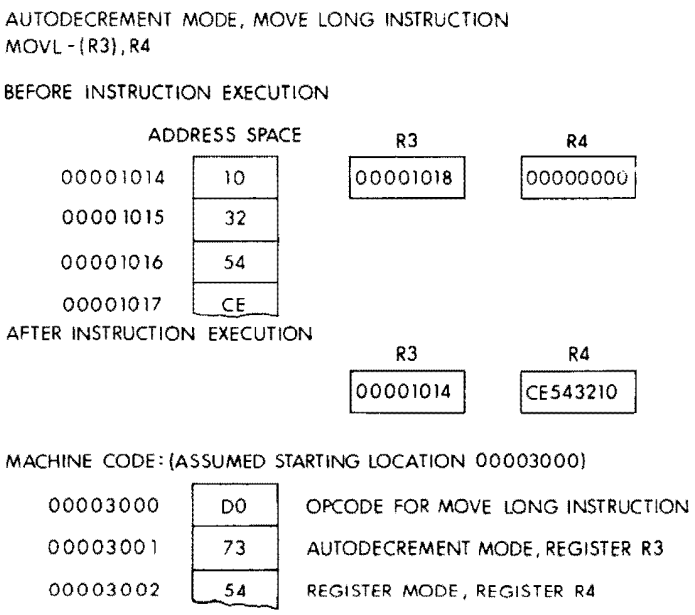


Figure 4-2
Instruction Representation

essor automatically pops the arguments off on return from the procedure.

The importance of the way the Call instructions work is that nested calls can be traced back to any previous level. The Call instructions always keep track of nested calls by using the Frame Pointer register. The Frame Pointer contains the address on the stack of the items pushed on the stack during the procedure call. The set of items pushed on the stack during a procedure call is known as a **call frame** or **stack frame**. Since the previous contents of the Current Frame register are saved in each call frame, the nested frames form a linked data structure which can be unwound to any level when an error or exception condition occurs in any procedure.

Processor Status Word

The Processor Status Word (a portion of the Processor Status Longword) is a special processor register that a program uses to check its status and to control synchronous error conditions. The Processor Status Word, illustrated in Figure 4-3, contains two sets of bit fields:

- the condition codes
- the trap enable flags

The condition codes indicate the outcome of a particular logical or arithmetic operation. For example, the Subtract instruction sets the Negative bit if the result of the subtraction operation produced a negative number, and it sets the Zero bit if the result produced zero. The Branch on Condition instructions can be used to transfer control to a code sequence that handles the condition.

There are two kinds of exceptions that concern the user process: trace faults and arithmetic exceptions. The trace fault is used by debugging programs or performance evaluators. Arithmetic exceptions include:

- integer, floating point, or decimal string overflow, in which the result was too large to be stored in the given format
- integer, floating point, or decimal string divide by zero, in which the divisor supplied was zero
- floating point underflow, in which the result was too small to be expressed in the given format

Of the arithmetic exceptions, integer overflow, floating underflow, and decimal string overflow may be handled in one of two ways. By clearing the exception enable bits in the Processor Status Word, the processor can be directed to ignore integer and decimal string overflow and floating underflow. The user may check for these conditions either by testing the condition codes (except for underflow) using the Branch on Condition instructions or by enabling the exception bits. By enabling the exception bits, the processor treats integer and decimal string overflow and floating underflow as exceptions. In any case, floating overflow and divide by zero exceptions are always enabled.

Handling Exceptions

When an exception occurs, the processor immediately saves the current state of execution and traps to the operating system. The operating system automatically searches for a procedure that wants to handle the exception. Procedures that respond to exceptions are called **condition handlers**. The user can declare a condition handler for an entire image and for each individual procedure called. In addition, because the processor keeps track of nested calls using the Frame Pointer register, it is possible to declare condition handlers for procedures that call other procedures in which exceptions might occur. The operating system automatically traces back through call frames to find a condition handler that wants to handle an exception that occurs.

NATIVE INSTRUCTION SET

The instruction set that the processor executes is selected under operating system control to either native mode or compatibility mode. The native mode instruction set is based on over 200 different opcodes. The opcodes can be grouped into classes based on their function and use. Instructions used to manipulate the general data types include:

- integer and logical instructions
- floating point instructions
- packed decimal instructions
- character string instructions
- bit field instructions

Instructions that are used to manipulate special kinds of data include:

- queue manipulation instructions

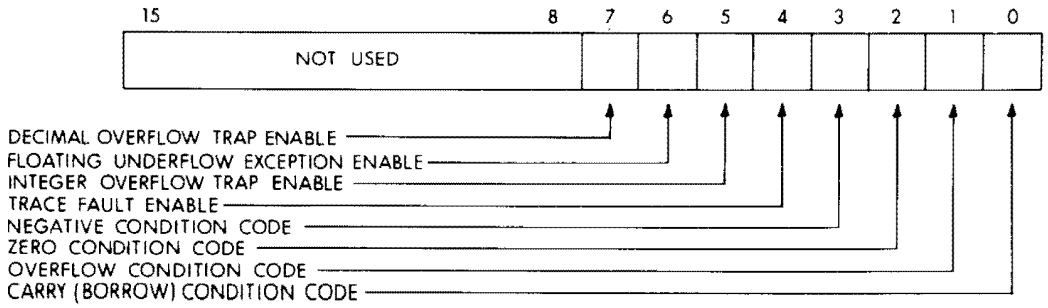


Figure 4-3
Processor Status Word

- address manipulation instructions
- user-programmed general register control instructions

Instructions that provide basic program flow control, and enable the user to call procedures are:

- branch, jump and case instructions
- subroutine call instructions
- procedure call instructions

Table 4-3 lists the basic instruction operations in order by these classifications. Instructions that enable operating system procedures to provide user mode processes with services requiring privilege are listed in the table, but discussed in the system programming environment section. Instructions that are singular in the functions they provide are listed last. The following paragraphs describe the functions of most of the instructions within each class. For further information on the instruction set, refer to the VAX-11 Architecture Handbook.

Table 4-3
Instruction Set Summary

Integer and Floating Point Logical Instructions

MOV_*	Move (B,W,L,F,D,G,H,Q,O)**
MNEG_	Move Negated (B,W,L,F,D,G,H)
MCOM_	Move Complemented (B,W,L)
MOVZ_	Move Zero-Extended (BW,BL,WL)
CLR_	Clear (B,W,L=F,Q=D=G,O=H)
CVT_	Convert (B,W,L,F,D,G,H)(B,W,L,F,D,G,H) except BB,WW,LL,FF,DD,GG,HH,DG, and GD
CVTR_L	Convert Rounded (F,D,G,H) to Longword
CMP_	Compare (B,W,L,F,D,G,H)
TST_	Test (B,W,L,F,D,G,H)
BIS_2	Bit Set (B,W,L) 2-Operand
BIS_3	Bit Set (B,W,L) 3-Operand
BIC_2	Bit Clear (B,W,L) 2-Operand
BIC_3	Bit Clear (B,W,L) 3-Operand
BIT_	Bit Test (B,W,L)
XOR_2	Exclusive OR (B,W,L) 2-Operand
XOR_3	Exclusive OR (B,W,L) 3-Operand
ROTL	Rotate Longword

Integer and Floating Point Arithmetic Instructions

INC_	Increment (B,W,L)
DEC_	Decrement (B,W,L)
ASH_	Arithmetic Shift (L,Q)
ADD_2	Add (B,W,L,F,D,G,H) 2-Operand
ADD_3	Add (B,W,L,F,D,G,H) 3-Operand
ADWC	Add with Carry
ADAWI	Add Aligned Word Interlocked
SUB_2	Subtract (B,W,L,F,D,G,H) 2-Operand
SUB_3	Subtract (B,W,L,F,D,G,H) 3-Operand
SBWC	Subtract with Carry
MUL_2	Multiply (B,W,L,F,D,G,H) 2-Operand
MUL_3	Multiply (B,W,L,F,D,G,H) 3-Operand
EMUL	Extended Multiply
DIV_2	Divide (B,W,L,F,D,G,H) 2-Operand
DIV_3	Divide (B,W,L,F,D,G,H) 3-Operand
EDIV	Extended Divide
EMOD_	Extended Modulus (F,D,G,H)
POLY_	Polynomial Evaluation (F,D,G,H)

Packed Decimal Instructions

MOVP	Move Packed
CMPP3	Compare Packed 3-Operand
CMPP4	Compare Packed 4-Operand
ASHP	Arithmetic Shift Packed and Round
ADDP4	Add Packed 4-Operand
ADDP6	Add Packed 6-Operand
SUBP4	Subtract Packed 4-Operand
SUBP6	Subtract Packed 6-Operand
MULP	Multiply Packed
DIVP	Divide Packed
CVTLP	Convert Long to Packed
CVTPL	Convert Packed to Long
CVTPT	Convert Packed to Trailing
CVTTP	Convert Trailing to Packed
CVTPS	Convert Packed to Separate
CVTSP	Convert Separate to Packed
EDITPC	Edit Packed to Character String

Character String Instructions

MOVC3	Move Character 3-Operand
MOVC5	Move Character 5-Operand
MOVTC	Move Translated Characters
MOVTUC	Move Translated Until Character
CMPC3	Compare Characters 3-Operand
CMPC5	Compare Characters 5-Operand
LOCC	Locate Character
SKPC	Skip Character
SCANC	Scan Characters
SPANC	Span Characters
MATCHC	Match Characters

Variable-Length Bit Field Instructions

EXTV	Extract Field
EXTZV	Extract Zero-Extended Field
INSV	Insert Field
CMPV	Compare Field
CMPZV	Compare Zero-Extended Field
FFS	Find First Set
FFC	Find First Clear

Table 4-3 (Cont.)
Instruction Set Summary

Index Instruction

INDEX	Compute Index

Queue Instructions

INSQUE	Insert Entry in Queue
INSQHI	Insert Entry into Queue at Head, Interlocked
INSQTI	Insert Entry into Queue at Tail, Interlocked
REMQUE	Remove Entry from Queue
REMQHI	Remove Entry from Queue at Head, Interlocked
REMQTI	Remove Entry from Queue at Tail, Interlocked

Address Manipulation Instructions

MOVA_	Move Address (B,W,L=F,Q=D=G,O=H)
PUSHA_	Push Address (B,W,L=F,Q=D=G,O=H) on Stack

General Register Manipulation Instructions

PUSHL	Push Longword on Stack
PUSHR	Push Registers on Stack
POPR	Pop Registers from Stack
MOVPSL	Move from Processor Status Longword
BISPSW	Bit Set Processor Status Word
BICPSW	Bit Clear Processor Status Word

Unconditional Branch and Jump Instructions

BR_	Branch with (B, W) Displacement
JMP	Jump

Branch on Condition Code

BLSS	Less Than
BLSSU	Less than Unsigned
BLEQ	Less than or Equal
BLEQU	Less than or Equal Unsigned
BEQL	Equal
(BEQLU)	(Equal Unsigned)
BNEQ	Not Equal
(BNEQU)	(Not Equal Unsigned)
BGTR	Greater than
BGTRU	Greater than Unsigned
BGEQ	Greater than or Equal
BGEQU	Greater than or Equal Unsigned
(BCC)	(Carry Cleared)
(BCS)	(Carry Set)
BVS	Overflow Set
BVC	Overflow Clear

Branch on Bit

BLB_	Branch on Low Bit (Set, Clear)
BB_	Branch on Bit (Set, Clear)
BBS_	Branch on Bit Set and (Set, Clear) Bit
BBC_	Branch on Bit Clear and (Set, Clear) Bit
BBSSI	Branch on Bit Set and Set Bit Interlocked
BBCCI	Branch on Bit Clear and Clear Bit Interlocked

Loop and Case Branch

ACB_	Add, Compare and Branch (B,W,L,F,D,G,H)
AOBLEQ	Add One and Branch Less Than or Equal
AOBLSS	Add One and Branch Less Than
SOBGEQ	Subtract One and Branch Greater Than or Equal
SOBGTR	Subtract One and Branch Greater Than
CASE_	Case on (B,W,L)

Subroutine Call and Return Instructions

BSB_	Branch to Subroutine with (B,W) Displacement
JSB	Jump to Subroutine
RSB	Return from Subroutine

Procedure Call and Return Instructions

CALLG	Call Procedure with General Argument List
CALLS	Call Procedure with Stack Argument List
RET	Return from Procedure

Protected Procedure Call and Return Instructions

CHM_	Change Mode to (Kernel, Executive, Supervisor, User)
REI	Return from Exception or Interrupt
PROBER	Probe Read
PROBEW	Probe Write

Privileged Processor Register Control Instructions

SVPCTX	Save Process Context
LDPCTX	Load Process Context
MTPR	Move to Process Register
MFPR	Move from Processor Register

Special Function Instructions

CRC	Cyclic Redundancy Check
BPT	Breakpoint Fault
XFC	Extended Function Call
NOP	No Operation
HALT	Halt

* The underscore following the instruction name implies that the instruction will operate upon any data type contained in the parentheses following that instruction.

** B = byte
W = word
L = longword
Q = quadword
O = octaword
F = F_floating
D = D_floating
G = G_floating
H = H_floating

Instructions that operate on G, H, and O formats are only available on VAX family processors equipped with the extended range floating point option.

Integer and Floating Point Instructions

The logical and integer arithmetic instructions illustrate how the opcodes, data types, and addressing modes can be combined in an instruction. Most of the operations provided for integer data are also provided for floating point and packed decimal data. Exceptions are the strictly logical operations for integer data (such as bit clear, bit set, complement), the multiword arithmetic instructions for integer data (such as Add/Subtract with Carry and Extended Multiply and Extended Divide), and the Extended Modulus and Polynomial instructions for floating point data.

The arithmetic instructions include both 2-operand and 3-operand forms that eliminate the need to move data to and from temporary operands. The 2-operand instructions store the result in one of the two operands, as in "Set A equal to A plus B." The 3-operand instructions effectively implement the high-level language statements in which two different variables are used to calculate a third, such as "Set C equal to A plus B." The 3-operand instructions are applicable to both integer and floating point data, and equivalent instructions exist for packed decimal data.

To illustrate the instruction set and addressing modes, consider the FORTRAN language statement:

A(I) = B(I) * C(I)

where A, B, and C are statically allocated REAL*4 arrays and I is INTEGER*4. A code sequence that performs this operation is:

```
MOVL    I,R0                ;Move the longword I
                            ;to a register
MULF3   B[R0],C[R0],A[R0]   ;3-operand floating
                            ;multiply
```

The same code applies if A, B, and C are REAL*8, INTEGER*4, INTEGER*2, or even INTEGER*1 data types: the MULF3 instruction is simply changed to MULD3, MULL3, MULW3, or MULB3, respectively.

If arrays A, B, and C are dynamically allocated arrays, the code sequence could be:

```
MOVL    I,R0
MULF3   Bdisp(FP)[R0],Cdisp(FP)[R0],Adisp(FP)[R0]
```

If A, B, and C are arguments to a procedure, the code could be:

```
MOVL    I,R0
MULF3   @Bargptr(AP)[R0],@Cargptr(AP)[R0],
        ;@Aargptr[R0]
```

In fact, the locations of A, B, and C can be arbitrarily selected. For example, combining the above, if A is statically allocated, B dynamically allocated, and C an argument, then the code sequence could be:

```
MOVL    I,R0
MULF3   Bdisp(FP)[R0],@Cargptr(AP)[R0],A[R0]
```

Some of the arithmetic instructions are used for extending the accuracy of repeated computations. The Extended Multiply (EMUL) instruction takes longword integer arguments and produces a quadword result. The instruction effectively implements a high-level language statement such as "Set D equal to (A times B) plus C." The Extended Divide (EDIV) instruction divides a quadword integer by a longword and produces a longword quotient and a longword remainder.

The Extended Modulus (EMOD) instructions multiply a floating point number with an extended precision floating point number (extended by eight bits for F_floating and D_floating for an effective 9 or 19 digits of accuracy) and returns the integer portion and the fractional portion separately. This instruction is particularly useful for the reduction of the argument of trigonometric and exponential functions to a standard interval.

The Polynomial Evaluation (POLY) instructions evaluate a polynomial from a table of coefficients using Horner's method. This instruction is used extensively in the high-level languages' math library for operations such as sine and cosine.

Packed Decimal Instructions

Many of the operations for integer and floating point data also apply to packed decimal strings. They include:

- Move Packed (MOVP) for copying a packed decimal string from one location to another, and Arithmetic Shift Packed (ASHP) for scaling a packed decimal up or down by a given power of 10 while moving it, and optionally rounding the value.
- Compare Packed (CMPP) for comparing two packed decimal strings. Compare Packed has two variations: a 3-operand (CMPP3) instruction for strings of equal length, and a 4-operand instruction (CMPP4) for strings of differing lengths.
- Convert Instructions, including Convert Long to Packed (CVTLP), Convert Packed to Long (CVTPL), Convert Packed to Numeric with Trailing sign (CVTPT), Convert Numeric with Trailing sign to Packed (CVTTP), Convert Packed to Numeric with Separate overpunched sign (CVTPS), and Convert Numeric with Separate overpunched sign to Packed (CVTSP). These instructions enable the conversion of our packed decimal format to commonly used numeric formats. Numeric with trailing sign allows various sign encodings including zoned and overpunched.
- Add Packed (ADDP) and Subtract Packed (SUBP) for adding or subtracting two packed decimal strings, with the option of replacing the addend or subtrahend with the result (ADDP4 and SUBP4), or storing the result in a third string (ADDP6 or SUBP6).
- Multiply Packed (MULP) and Divide Packed (DIVP) for multiplying or dividing two packed decimal strings and storing the result in a third string.

In addition, the packed decimal instructions include a special packed decimal string to character string conversion instruction that provides output formatting: the Edit instruction.

Edit Instruction

The Edit Packed to Character String (EDITPC) instruction supplies formatted numeric output functions. The instruction converts a given packed decimal string to a character string using selected pattern operators. The pattern operators enable the creation of numeric output fields with any of the following characteristics:

- leading zero fill
- leading zero protection
- leading asterisk fill protection

- a floating sign
- a floating currency symbol
- special sign representations
- insertion characters
- blank when zero

Character String Instructions
The character string instructions operate on strings of bytes. They include:

- move string instructions, with translation options
- string compare instructions
- single character search instructions
- substring search instructions

There are two basic forms of Move instructions for character strings. The Move Character instructions (MOVC3 and MOVC5) simply copy character strings from one location to another. They are optimized for block transfer operations. The 5-operand variation provides for a fill character (user-supplied) that the instruction uses to pad out the destination location to a given size.

The Move Translated Characters (MOVTC) and Move Translated Until Character (MOVTUC) instructions actually create new character strings. The user supplies a string which the instruction uses as a list of offsets into a translation table. The instruction selects characters from the table in the order that the offset list points to the table. The MOVTC instruction allows the user to supply a fill character that the instruction uses to pad out the resultant string to a given size with an arbitrary character. The MOVTUC instruction allows the user to supply any number of escape characters. When the next offset points to an escape character in the table, translation stops.

The Compare Characters (CMPC) instructions provide character-by-character byte string compares. CMPC has a 3-operand form and a 5-operand form. Both instructions compare two strings from beginning to end, informing the user when they reach the first character that is different between the strings, or when they get to the end of either string. The 5-operand variation provides for a fill character which it uses to effectively pad out a string when comparing it with a longer one.

The Locate Character (LOCC) and Skip Character (SKPC) instructions are search instructions for single characters within a string. LOCC searches a given string for a character that matches the search character supplied by the user. This is useful, for example, when searching for the delimiter at the end of a variable-length string. SKPC, on the other hand, finds the first character in the string that is *different* from the search character supplied. This is useful for skipping through fill characters at the end of a field to find the beginning of the next field.

The Match Characters (MATCHC) instruction is similar to the Locate Character instruction, but it locates multiple-character substrings. MATCHC searches a string for the first occurrence of a substring supplied by the user.

The Span Characters (SPANC) and Scan Characters (SCANC) instructions are search instructions that look for members of character classes. For these instructions the user supplies a character string, a mask, and the address of a 256-byte table of character type definitions. For each character in the given string, the instruction looks up the type code in the table for that character, and then AND the given mask with the character's type code. SPANC finds the first character in the string which is of the type indicated by its mask. SCANC finds the first character in the string which is of any type other the one indicated by its mask.

The Index Instruction
The Index instruction (INDEX) calculates an index for an array of fixed length data types (integer and floating) and for arrays of bit fields, character strings, and decimal strings. It accepts as arguments: a subscript, lower and upper subscript bounds, an array element size, a given index, and a destination for the calculated index. It incorporates range checking within the calculation for high-level languages using subscript bounds, and it allows index calculation optimization by removing invariant expressions.

The COBOL statements:

```
01  A-ARRAY
    02      A PIC X(10) OCCURS 15 TIMES.
01  B PIC X(10).
    MOVE A(I) TO B.
```

are equivalent to:

```
INDEX I, #1, #15, #10, #0, R0   ; 1 less than or equal I
                                ; I less than or equal 15
                                ; (0 + I) * 10 is
                                ; stored in R0
MOVC3 #10, A-10[R0],B
```

The FORTRAN statements:

```
INTEGER*4 A(L1:U1, L2:U2), I, J
A(I,J) = 1
```

are equivalent to:

```
INDEX J, #L2, #U2, #M1, #0, R0      ; M1 = U1 - L1
INDEX I, #L1, #U1, #1, R0, R0
MOVL    #1, A-a[R0]                 ; a = ((L2*M1)+L1)*4
```

Variable-Length Bit Field Instructions
The bit field instructions enable the user to define, access, and modify those fields whose size and location were user-specified. Location is determined from a base address or a register and a signed bit offset. If the field is in memory, the offset can reach bits located up to 2^{31} bits (approximately 256 million bytes) away in either direction. If the field is in a register, the offset can be large as 31. Fields of arbitrary lengths (0 to 32 bits) may be used for storing data structure header information compactly, for status codes, or for creating user data types. The field instructions enable manipulation of fields easily.

The Insert Field and Extract Field instructions store data in and retrieve data from fields. Insert Field (INSV) stores data in a field by taking a specified number of bits of a longword (starting from the low-order bit) and writing them into a field, which may start at any bit relative to a given base address. The Extract Field instruction retrieves data from a field by copying the bit field and storing it in the low-order

bits of a longword. The field can either be signed (EXTV) or unsigned (EXTZV).

The Compare Field and Find First instructions enable the user to test the contents of a field. Compare Field extracts a field and then compares it with a given longword. The field can be interpreted as signed (CMPV), or as unsigned (CMPZV). The Find First instructions locate the first bit in a field that is clear (FFC) or set (FFS), scanning from low-order bit to high-order bit. These instructions are particularly useful for scanning a status control longword. For example, the longword may represent a set of queues processed in order by priority 0 (high) to 31 (low). Each set bit represents an active queue. The Find First Set instruction quickly returns the highest priority queue that is active. Together with the SKPC instructions, the Find First instructions are also useful for scanning an allocation table (bit map) of arbitrary length.

Queue Instructions

The processor has six instructions that allow easy construction and maintenance of queue data structures. Queues manipulated using the queue instructions are circular, doubly linked lists of data items.

The first longword of a queue entry contains the forward pointer to the next entry in the queue, and the next longword contains the backward pointer to the preceding entry in the queue.

Two types of queues are provided: absolute and self-relative. Absolute queues use pointers that are virtual addresses, whereas self-relative queues use pointers that are relative displacements.

Two instructions are provided for manipulating absolute queues: INSQUE, and REMQUE. INSQUE inserts an entry specified by an entry operand into the queue following the entry specified by the predecessor operand. REMQUE removes the entry specified by the entry operand. Queue entries can be on arbitrary byte boundaries. Both INSQUE and REMQUE are implemented as non-interruptible instructions.

Four operations can be performed on self-relative queues: insert at head (INSQHI), remove from head (REMQHI), insert at tail (INSQTI), and remove from tail (REMQTI). Furthermore, these operations are interlocked to allow cooperating processes in a multiprocessor system to access a shared queue without additional synchronization. Queue entries must be quadword aligned.

Address Manipulation Instructions

Because the processor offers a variety of addressing modes enabling access to data structures easily via base addresses and indices in registers, addresses are often manipulated. The processor provides two instructions enabling an address to be fetched without actually accessing the data at that location:

- The Move Address (MOVA) instruction, which stores the address of a byte, word, longword (and floating), or quadword (and double floating) datum in a specified register or location in memory.
- The Push Address (PUSHA) instruction, which stores the address of a byte, word, longword (and floating), or quadword (and double floating) datum on the stack.

The Push Address instruction is useful for computing an address to be passed to a called subroutine or procedure. Move Address is useful for loading a base register and performing run time position-independent address computation. It has some interesting uses because it is effectively an ADD instruction:

```
MOVAB disp(R1)[R2],X    ; sets X=R1+R2+displacement
                        ; (two adds in one instruction)

MOVA_disp(Rn)[Rn],Rn    ; multiplies Rn by 3
                        ; (for MOVAW),
                        ; 5 (for MOVAL),
                        ; or 9 (for MOVAQ)
                        ; and adds displacement to it.
```

General Register Manipulation Instructions

The general register manipulation instructions enable any user program to save or load the general purpose registers in one operation, examine the Processor Status Longword, and set or clear status bits in the Processor Status Word. (Processor register control instructions primarily used by operating system software are covered later.)

The Push Longword (PUSHL) instruction pushes a longword on the stack. This instruction is the same as a Move Longword using the Stack Pointer in register deferred mode, but is a byte shorter. It is a consistent and convenient way to move data to the stack.

The Push Registers (PUSHR) instruction pushes a set of registers on the stack in one operation. The user supplies a mask word in which each set bit (0-14) represents a register (R0-R14) to be saved on the stack. (The only general register that cannot be saved using this instruction is R15, the Program Counter.) Pop Registers (POPR) reverses the operation, loading each register from successive longwords on the stack according to the given mask word. The PUSHR and POPR instructions replace the need to write a sequence of Move instructions to save and restore registers upon entry and exit from a subroutine.

The Move from Processor Status Longword (MOVPSL) instruction allows examination of the contents of the processor's status register by loading its contents into a specified location. The Bit Set (BISPSW) and Bit Clear (BICPSW) Processor Status Word instructions enable the user to set or clear the PSW condition codes and trap enable bits. The mask bits represent the bits to be set or cleared.

Branch, Jump and Case Instructions

The two basic types of control transfer instructions are branch and jump instructions. Both branch and jump load new addresses in the Program Counter. With branch instructions, the user supplied displacement (offset) is added to the current contents of the Program Counter to obtain the new address. The jump instructions allow the user specified address to be loaded, using one of the normal addressing modes.

Because most transfers are to locations relatively close to the current instruction, and branch instructions are more efficient than jump instructions, the processor offers a variety of branch instructions to choose from. There are two unconditional branch instructions and many conditional branch instructions.

The unconditional branch instructions allow specification of either a byte-size (BRB) or a word-size displacement (BRW), thereby permitting displacements as far away from the current location as 32,767 bytes in either direction. The Jump instruction (JMP) should be used for transfer of control to locations greater in displacement than 32,767 bytes.

Most conditional branches allow only byte displacements, although some of the more powerful, such as the Add Compare and Branch instruction, allow word displacements. Conditional branch instructions include:

- branch on bit instructions
- set and clear bit instructions with a branch if it is already set or cleared
- loop instructions that increment or decrement a counter, compare it with a limit value, and branch on a relational condition
- computed branch instruction in which a branch may take place to one of several locations depending on a computed value

The Branch on Condition (B) instructions enable transfer of control to another location depending on the status of one or more of the condition codes in the Processor Status Word (PSW). There are three groups of Branch on Condition instructions:

- The signed relational branches, which are used to test the outcome of instructions operating on integer and field data types being treated as signed integers, floating point data types, and decimal strings.
- The unsigned relational branches, which are used to test the outcome of instructions operating on integer and field data types being treated as unsigned integers, character strings, and addresses.
- The overflow and carry test branches, which are used for checking overflow when traps are not enabled, for multiprecision arithmetic, and for the results of special instructions.

The instruction mnemonics clearly indicate the choice between a signed and unsigned integer data type interpretation for relational testing. The relational tests determine if the result of the previous operation is less than, less than or equal, equal, not equal, greater than or equal, or greater than zero. For example, the Branch on Less than or Equal Unsigned (BLEQU) instruction branches if either the Carry or Zero bit is set. The Branch on Greater Than (BGTR) instruction branches if neither the Negative nor the Zero bit is set.

General purpose Branch on Bit instructions similar to Branch on Condition also exist. The Branch on Low Bit Set (BLBS) and Branch on Low Bit Clear (BLBC) instructions test bit 0 of an operand, which is useful for testing Boolean values. The Branch on Bit Set (BBS) and Branch on Bit Clear (BBC) instructions test any selected bit.

There are special kinds of Branch on Bit instructions that are actually bit set/clear instructions. The Branch on Bit Set and Set (BBSS) is an example. The instruction branches if the indicated bit is set, otherwise it falls through. In either case, the instruction sets the given bit. The BBSS instruction can thus be thought of as a Bit Set instruction with a branch side-effect if the bit was already set. There are four permutations:

- Branch on Bit Set and Set (BBSS)
- Branch on Bit Clear and Clear (BBCC)
- Branch on Bit Set and Clear (BBSC)
- Branch on Bit Clear and Set (BBCS)

These instructions are particularly useful for keeping track of procedure completion or initialization, and for signaling the completion or initialization of a procedure to a cooperating process. In addition, there are two Branch on Bit Interlocked instructions that provide control variable protection:

- Branch on Bit Set and Set Interlocked (BBSSI)
- Branch on Bit Clear and Clear Interlocked (BBCCI)

The memory interconnect bus provides a memory interlock on these instructions. No other BBSSI or BBCCI operation can interrupt these instructions to gain access to the byte containing the control variable between the testing of the bit and the setting or clearing of the bit.

The processor offers three types of branch instructions that can be used to write efficient loops. The first type provides the basic subtract-one-and-branch loop. A counter variable (user-supplied) is decremented each time the loop is executed. In the Subtract One and Branch Greater Than (SOBGTR) instruction, the loop repeats until the counter equals zero. In the Subtract One and Branch Greater Than or Equal (SOBGEQ) instruction, the loop repeats until the counter becomes negative.

The counterpart to subtract-one-and-branch is add-one-and-branch. A counter and a limit must be supplied by the user. The counter is incremented at the end of the loop. In the Add One and Branch Less Than (AOBLSS) instruction, the loop repeats until the counter equals the user-defined limit. In the Add One and Branch Less Than or Equal (AOBLEQ) instruction, the loop repeats until the counter exceeds the user defined limit.

The third type of loop instruction efficiently implements the FORTRAN language DO statement and the BASIC language FOR statement: Add Compare and Branch (ACB). In this case, the user must supply a limit, a counter, and a step value. For each execution of the loop, the instruction adds the step value to the counter and compares the counter to the limit. The sign of the step value determines the logical relation of the comparison: the instruction loops on a less than or equal comparison if the step value is positive, on a greater than or equal comparison if the step value is negative.

The processor provides a branch instruction that implements higher-level language computed GO TO statements: the CASE instruction. To execute the CASE instruction, the user must supply a list of displacements that generate different branch addresses indexed by the value obtained as a selector. The branch falls through if the selector does not fall within the limits of the list.

Subroutine Branch, Jump, and Return Instructions

Two special types of branch and jump instruction are provided for calling subroutines: the Branch to Subroutine (BSB) and Jump to Subroutine (JSB) instructions. Both BSB and JSB instructions save the contents of the Program Counter on the stack before loading the Program Counter with the new address. With Branch to Subroutine,

the user supplies either a byte (BSBB) or word (BSBW) displacement. With Jump to Subroutine, regular addressing is used.

The subroutine call instructions are complemented by the Return from Subroutine (RSB) instruction. RSB pops the first longword off the stack and loads it into the Program Counter. Since the Branch to Subroutine instruction is either two or three bytes long, and the Return from Subroutine instruction is one byte long, it is possible to write extremely efficient programs using subroutines.

Procedure Call and Return Instructions

Procedures are general purpose routines that use argument lists passed automatically by the processor. The procedure Call instructions enable language processors and the operating system to provide a standard calling interface. They:

- save all the registers that the procedure uses, and only those registers, before entering the procedure
- pass an argument list to a procedure
- maintain the Stack, Frame, and Argument Pointer registers
- initialize the arithmetic trap enables to a given state

When issuing a Call procedure instruction, the address of the procedure being called, must be included. The first word of a procedure contains an entry mask that is used in the same way as the entry mask defined for the Push Registers instruction. Each set bit of the 12 low-order bits in the word represents one of the general registers, R0 through R11, that the procedure uses. The Call instruction examines this word and saves the indicated registers on the stack. In addition, the Call instruction also automatically saves the contents of the Frame Pointer, Argument Pointer, and Program Counter registers. This is an extremely efficient way to ensure that registers are saved across procedure calls. No general register is saved that does not have to be saved.

The Call Procedure with General Argument List (CALLG) instruction accepts the address of an argument list and passes the address to the procedure in the Argument Pointer register. The Call Procedure with Stack Argument List (CALLS) passes the argument list, (placed on the stack by the user) by loading the Argument Pointer register with its stack address.

When a procedure completes execution, it issues the Return from Procedure instruction (RET). Return uses the Frame Pointer register to find the saved registers that it restores, and to clean up any data left on the stack, including nested routine linkages. A procedure can return values using the argument list or other registers.

Miscellaneous Special Purpose Instructions

The processor has a number of special purpose instructions. They include:

- Cyclic Redundancy Check (CRC)
- Breakpoint Fault (BPT)
- Extended Function Call (XFC)
- No Operation (NOP)
- Halt

The Cyclic Redundancy Check (CRC) instruction calculates a cyclic redundancy check for a given string using any CRC polynomial up to 32 bits long. The user supplies the string for which the CRC is to be performed, and a table for the CRC function. The operating system library includes tables for standard CRC functions, such as CRC-16.

The Breakpoint Fault (BPT) instruction makes the processor execute the kernel mode condition handler associated with the Breakpoint Fault exception vector. BPT is used by the operating system debugging utilities, but can also be used by any process that sets up a Breakpoint Fault condition handler.

The Extended Function Call (XFC) instruction allows escapes to customer-defined instructions in writable control store. The NOP instruction is useful for debugging. The HALT instruction is a privileged instruction issued only by the operating system to halt the processor when bringing the system down by operator request.

COMPATIBILITY MODE

Under control of the operating system, the processor can execute PDP-11 instruction streams within the context of any process. When executing in compatibility mode, the processor interprets the instruction stream executing in the context of the current process as a subset of the PDP-11 instruction set.

In general, compatibility mode enables the operating system to provide an environment for executing most user mode programs written for a PDP-11 except stand-alone software. The processor expects all compatibility mode software to rely on the services of the native operating system for I/O processing, interrupt and exception handling, and memory management. There are some restrictions, however, on the environment that the native operating system can provide a PDP-11 program. For example, the PDP-11 memory management instructions Move To/From Previous Instruction/Data Space can not be simulated by the operating system since they do not trap to native mode software.

PDP-11 Program Environment

PDP-11 addresses are 16-bit byte addresses. There is a one-to-one correspondence between compatibility mode virtual addresses and the first 64K bytes of virtual address space available to native mode processes. As in the PDP-11, a compatibility mode program is restricted to referencing only these addresses. It is possible for the operating system to provide most of the PDP-11 memory management mechanisms. For example, compatibility mode automatically supports PDP-11 memory segment protection, but in 512-byte rather than 64-byte segments.

All of the PDP-11 general registers and addressing modes are available in compatibility mode. Compatibility mode registers R0 through R6 are the low-order 16 bits of native mode registers R0 through R6. Compatibility mode R7 (the Program Counter) is the low-order bits of native mode register 15 (the Program Counter). Native mode registers 8 through 14 are not affected by compatibility mode. Note that the compatibility mode register R6 acts as the Stack Pointer for program-local temporary data storage, but that the program-local stack is allocated address space in the program region, not the control region.

A subset of the PDP-11 Processor Status Word is defined for compatibility mode. Only the condition codes and the trace trap bit are relevant for the PDP-11 instruction stream.

All interrupts and exceptions that occur when the processor is executing in compatibility mode cause the processor to enter native mode. As in native mode, it is the operating system's responsibility to handle interrupts and exceptions. There are a few types of exceptions that apply only to compatibility mode. They include illegal instruction exceptions and odd address trap.

PDP-11 Instruction Set

The compatibility mode instruction set is that of the PDP-11 with the following exceptions:

- The privileged instructions (HALT, WAIT, RESET, SPL, and MARK) are illegal.
- The trap instructions (BPT, IOT EMT, and TRAP) cause the processor to enter native mode, where either the trap may be serviced, or the instruction simulated
- The Move From/To Previous Instruction/Data space instructions (MFPI, MTPI, MFPD, and MTPD) execute exactly as they would on a PDP-11 in user mode with instruction and data space overmapped. They ignore the previous access level and act as PUSH and POP instructions referencing the current stack.
- PDP-11 floating point instructions are emulated through software.

All other instructions execute as they would on a PDP-11/70 processor running in user mode.

PROCESSING CONCEPTS FOR SYSTEM PROGRAMMING

The processor is specifically designed to support a high-performance multiprogramming environment. The chief advantage of a multiprogramming system is its ability to get the most out of a computer that is being used for several different purposes concurrently. For example, multiprogramming enables the simultaneous execution of two or more application systems, such as process control and order entry. It is also possible to execute several application systems while simultaneously developing application programs. The characteristics of the hardware system that support multiprogramming are:

- rapid context switching
- priority dispatching
- virtual addressing and memory management

As a multiprogramming system, VAX not only provides the ability to share the processor among processes, but also protects processes from one another while enabling them to communicate with each other and share code and data.

Context Switching

In a multiprogramming environment, several individual streams of code can be ready to execute at any one time. Instead of allowing each stream to execute to completion serially (as in a batch-only system), the operating system can intervene and switch between the streams of code which are ready to execute.

To support multiprogramming for a high-performance system, the processor enables the operating system to switch rapidly between individual streams of code. The stream of code the processor is executing at any one time is determined by its *hardware context*. Hardware context includes the information loaded in the processor's registers that identifies:

- where the stream's instructions and data are located
- which instruction to execute next
- what the processor status is during execution

A process is a stream of instructions and data defined by a hardware context. Each process has a unique identification in the system. The operating system switches between processes by requesting the processor to save one process hardware context and load another. Context switching occurs rapidly because the processor instruction set includes save hardware context and load hardware context instructions. The operating system's context switching software does not have to individually save or load the processor registers which define the hardware context.

The actual scheduling mechanism for arbitrating among processes competing for processor time is left to the operating system software itself to give the system flexibility.

Priority Dispatching

While running in the context of one process, the processor executes instructions and controls data flow to and from peripherals and main memory. To share processor, memory and peripheral resources among many processes, the processor provides two arbitration mechanisms that support high-performance multiprogramming: exceptions and interrupts. Exceptions are events that occur synchronously with respect to instruction execution, while interrupts are external events that occur asynchronously.

The flow of execution can change at any time, and the processor distinguishes between changes in flow that are local to a process and those that are system-wide. Process-local changes occur as the result of a user software error or when user software calls operating system services. Process-local changes in program flow are handled through the processor's exception detection mechanism and the operating system's exception dispatcher.

System-wide changes in flow generally occur as the result of interrupts from devices or interrupts generated by the operating system software. Interrupts are handled by the processor's interrupt detection mechanism and the operating system's interrupt service routines. (System-wide changes in flow may also occur as the result of severe hardware errors, in which case they are handled either as special exceptions or high-priority interrupts.)

System-wide changes in flow take priority over process-local changes in flow. Furthermore, the processor uses a priority system for servicing interrupts. To arbitrate between all possible interrupts, each kind of interrupt is assigned a priority, and the processor responds to the highest priority interrupt pending. For example, interrupts from real-time I/O devices would take precedence over interrupts from mass-storage devices, terminals, line printers and other less time-critical devices.

The processor services interrupts between instructions, or at well-defined points during the execution of long, itera-

tive instructions. When the processor acknowledges an interrupt, it switches rapidly to a special system-wide context to enable the operating system to service the interrupt. System-wide changes in the flow of execution are handled in such a way as to be totally transparent to individual processes.

Virtual Addressing and Virtual Memory

The processor's memory management hardware enables the operating system to provide an execution environment that allows users to write programs without having to know where the programs are loaded in physical memory, and to write programs that are too large to fit in the physical memory they are allocated.

The processor provides the operating system with the ability to provide virtual addressing. A virtual address is a 32-bit integer that a program uses to identify storage locations in virtual memory. Virtual memory is the set of all physical memory locations in the system plus the set of disk blocks that the operating system designates as extensions to physical memory.

A physical address is an address that the processor uses to identify physical memory storage locations and peripheral controller registers. It is the physical address that the processor sends to the memory and peripheral adaptors.

The processor must be capable of translating virtual addresses provided by the executing program into the physical addresses recognized by the memory and peripherals. To provide virtual to physical address mapping, the processor has address mapping registers controlled by the operating system and an integrated address translation buffer.

The mapping registers enable the operating system to relocate programs in physical memory, to protect programs from each other, and share instructions and data between programs transparently or at their request. The address translation buffer ensures that the virtual address to physical address translation takes place rapidly.

SYSTEM PROGRAMMING ENVIRONMENT

Within the context of one process, user-level software controls its execution using the instruction sets, the general registers and the Processor Status Word. Within the multiprogramming environment, the operating system controls the system's execution using a set of special instructions, the Processor Status Longword, and the internal processor registers.

Processor Status Longword

A processor register called the Processor Status Longword (PSL) determines the execution state of the processor at any time. The low-order 16 bits of the Processor Status Longword is the Processor Status Word available to the user process. The high-order 16 bits provide privileged control of the system. Figure 4-4 illustrates the Processor Status Longword.

The fields can be grouped together by functions that control:

- the instruction set the processor is executing
- the access mode of the current instruction
- interrupt processing

The instruction set the processor executes is controlled by the compatibility mode bit in the Processor Status Longword. This bit is normally set or cleared by the operating system. For further information on compatibility mode, refer to the Operating System section.

The following paragraphs discuss access modes, the native instructions primarily used by the operating system, memory management, and interrupt processing.

Processor Access Modes

In a high-performance multiprogramming system, the processor must provide the basis for protection and sharing among the processes competing for the system's resources. The basis for protection in this system is the processor's access mode. The access mode in which the processor executes determines:

- instruction execution privileges: what instructions the processor will execute
- memory access privileges: which locations in memory the current instruction can access

At any one time, the processor is executing code in the context of a particular process, or it is executing in the system-wide interrupt service context. In the context of a process, the processor recognizes four access modes: kernel, executive, supervisor, and user. Kernel is the most privileged mode and user the least privileged.

The processor spends most of its time executing in user mode in the context of one process or another. When user software needs the services of the operating system, whether for acquisition of a resource, for I/O processing, or for information, it calls those services.

The processor executes those services in the same or one

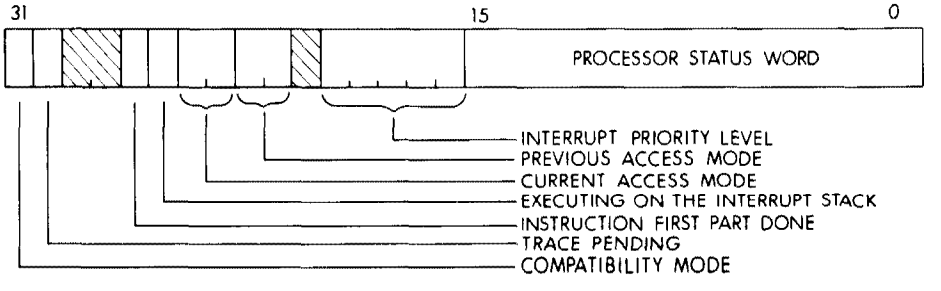


Figure 4-4
Processor Status Longword

of the more privileged access modes within the context of that process. That is, all four access modes exist within the same virtual address space. Each access mode has its own stack in the control region of per-process space, and therefore each process has four stacks: one for each access mode. Note that this makes it easy for the operating system to context switch a process even when it is executing an operating system service procedure.

In any mode except kernel, the processor will not execute the instructions that:

- halt the processor
- load and save process context
- access the internal processor registers that control memory management, interrupt processing, the processor console, or the processor clock

These instructions are privileged instructions that are generally reserved to the operating system.

In any mode, the processor will not allow the current instruction to access memory unless the mode is privileged to do so. The ability to execute code in one of the more privileged modes is granted by the system manager and controlled by the operating system. The memory protection the privilege affords is enforced by the processor. In general, code executing in one mode can protect itself and any portion of its data structures from read and/or write access by code executing in any less privileged mode. For example, code executing in executive mode can protect its data structures from code executing in supervisor or user mode. Code executing in supervisor mode can protect its data structures from access by code executing in user mode. This memory protection mechanism provides the basis for system data structure integrity.

Protected and Privileged Instructions

The processor provides three types of instructions that enable user mode software to obtain operating system services without jeopardizing the integrity of the system. They include:

- the Change Mode instructions
- the PROBE instructions
- the Return from Exception or Interrupt instruction

User mode software can obtain privileged services by calling operating system service procedures with a standard CALL instruction. The operating system's service dispatcher issues an appropriate Change Mode instruction before actually entering the procedure. Change Mode allows access mode transitions to take place from one mode to the same or more privileged mode only. When the mode transition takes place, the previous mode is saved in the Previous Mode field of the Processor Status Longword, allowing the more privileged code to determine the privilege of its caller.

A Change Mode instruction is simply a special trap instruction that can be thought of as an operating system service call instruction. User mode software can explicitly issue Change Mode instructions, but since the operating system receives the trap, non-privileged users can not write any code to execute in any of the privileged access modes. User mode software can include a condition handler for Change Mode to User traps, however, and this instruction is useful for providing general purpose services for user mode software. The system manager ultimately grants the privilege to write any code that handles Change Mode traps to more privileged access modes.

For service procedures written to execute in privileged access modes (kernel, executive, and supervisor), the processor provides address access privilege validation instructions. The PROBE instructions enable a procedure to check the read (PROBER) and write (PROBEW) access protection of pages in memory against the privileges of the caller who requested to access a particular location. This enables the operating system to provide services that execute in privileged modes to less privileged callers and still prevent the caller from accessing protected areas of memory.

The operating system's privileged service procedures and interrupt and exception service routines exit using the Return from Exception or Interrupt (REI) instruction. REI is the only way in which the privilege of the processor's access mode can be decreased. Like the procedure and subroutine return instructions, REI restores the Program Counter and the processor state to resume the process at the point where it was interrupted.

REI performs special services, however, that normal return instructions do not. For example, REI checks to see if any asynchronous system traps have been queued for the currently executing process while the interrupt or exception service routine was executing, and ensures that the process will receive them. Furthermore, REI checks to ensure that the mode to which it is returning control is the same as or less privileged than the mode in which the processor was executing when the exception or interrupt occurred. Thus REI is available to all software including user-written trap handling routines, but a program cannot increase its privilege by altering the processor state to be restored.

When the operating system schedules a context switching operation, the context switching procedure uses the Save Process Context (SVPCTX) and Load Process Context (LDPCTX) instructions to save the current process context and load another. The operating system's context switching procedure identifies the location of the hardware context to be loaded by updating an internal processor register.

Internal processor registers not only include those that identify the process currently executing, but also the memory management and other registers, such as the console and clock control registers. The Move to Processor Register (MTPR) and Move from Processor Register (MFPR) instructions are the only instructions that can explicitly access the internal processor registers. MTPR and MFPR are privileged instructions that can be issued only in kernel mode.

Memory Management

The processor is responsible for enforcing memory protection between access modes. Memory protection, however, is only a part of the processor's memory management function. In particular, the memory management hardware enables the operating system to provide an extremely flexible and efficient virtual memory programming environment. Virtual and physical address space defi-

nitions provide the basis for the virtual memory available on a system.

Virtual address space consists of all possible 32-bit addresses that can be exchanged between a program and the processor to identify a byte location in physical memory. The memory management hardware translates a virtual address into a physical address. A physical address can be up to 30 bits in length as in the case of the VAX-11/780. Other processor implementations may choose a smaller physical address. A physical address is the address exchanged between the processor and the memory and peripheral adaptors. Physical address space is the set of all possible physical addresses the processor can use to express unique memory locations and peripheral control registers. Figure 4-5 compares the structure of the common virtual address space with that of the VAX-11/750 and VAX-11/780 physical address spaces.

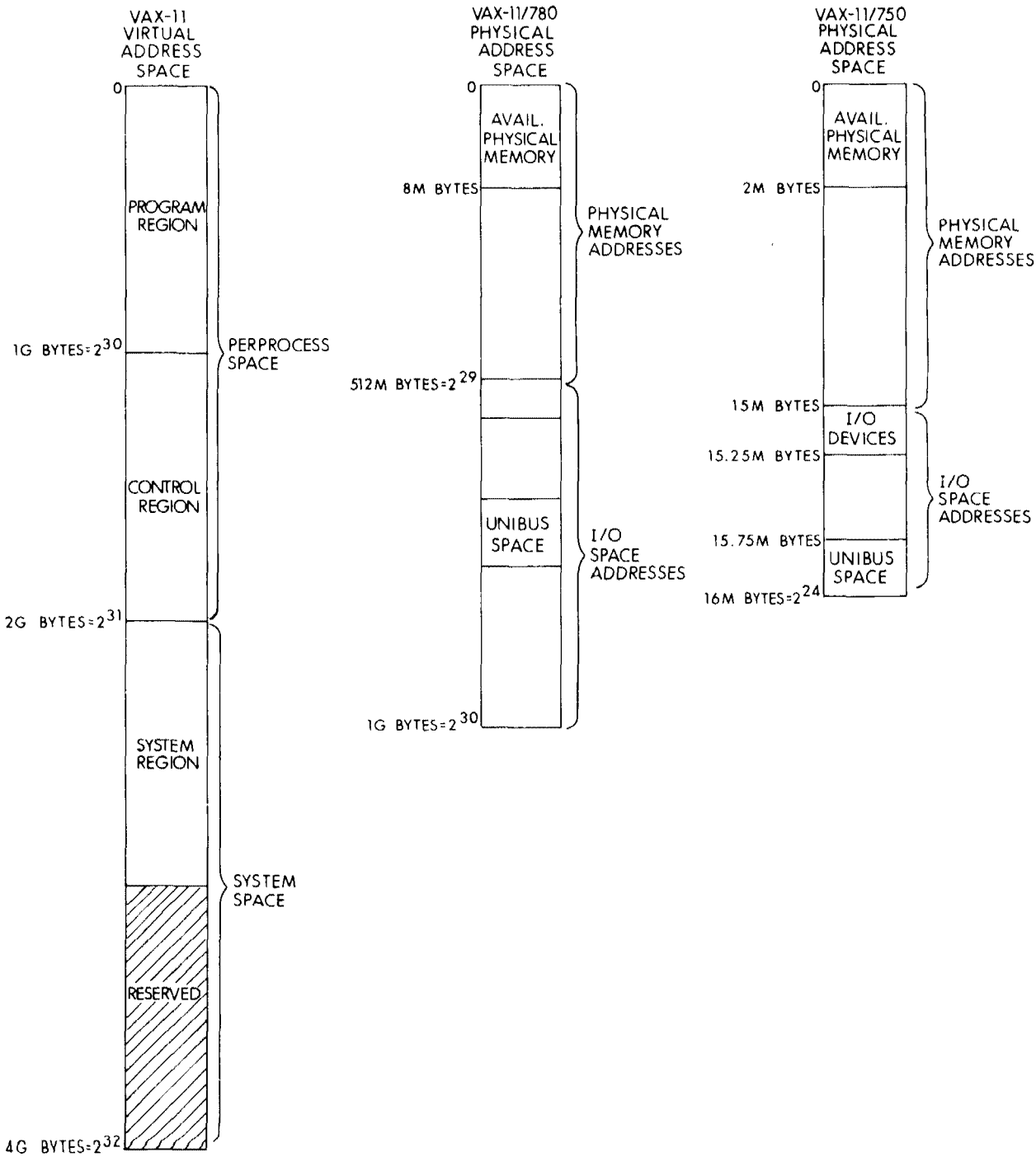


Figure 4-5

Virtual and Physical Address Space

On the VAX-11/780, physical address space is an array of addresses which can be used to represent 2^{30} byte locations, or approximately one billion bytes. Half of the addresses in VAX-11/780 physical address space can be used to refer to real memory locations and the other half can be used to refer to peripheral device control and data registers. The lowest-addressed half of physical address space is called **memory space**, and the highest-addressed half **I/O space**. On the VAX-11/750, physical address space is an array of addresses which can be used to represent 2^{24} byte locations, or approximately 16 million bytes. The first 15M bytes are dedicated to physical memory addresses while the last 1M byte is dedicated to I/O space.

The following section describes the way in which the memory management hardware enables the operating system to map virtual addresses into physical addresses to provide the virtual memory available to a process.

Virtual to Physical Page Mapping

Virtual address space is divided into pages, where a page represents 512 bytes of contiguously addressed memory. The first page begins at byte zero and continues to byte 511. The next page begins at byte 512 and continues to byte 1023, and so forth. For example, decimal and hexadecimal addresses of the first eight pages of virtual addresss space are:

PAGE	ADDRESS(10) decimal	ADDRESS(16) hexadecimal
0	0000-0511	0000-01FF
1	0512-1023	0200-03FF
2	1024-1535	0400-05FF
3	1536-2047	0600-07FF
4	2048-2559	0800-09FF
5	2560-3071	0A00-0BFF
6	3072-3583	0C00-0DFF
7	3584-4095	0E00-0FFF

The size of a virtual page exactly corresponds to the size of a physical page of memory, and the size of a block on disk.

To make memory mapping efficient, the processor must be capable of translating virtual addresses to physical addresses rapidly. Two features providing rapid address translation are the processor's internal address translation buffer and the translation algorithm itself.

Figure 4-6 compares the virtual address format to the physical address formats of the VAX-11/780 and VAX-11/750 processors. The high-order two bits of a virtual address immediately identify the region to which the virtual address refers. Whether the address is physical (processor specific) or virtual, the byte within the page is the same. Thus, the processor has to know only which virtual pages correspond to which physical pages.

The processor has three pairs of page mapping registers, one pair for each of the three regions actively used. The operating system's memory management software loads each pair of registers with the base address and length of data structures it sets up called **page tables**. The page tables provide the mapping information for each virtual page in the system. There is one page table for each of the three regions.

A page table is a virtually contiguous array of page table entries. Each page table entry is a longword representing the physical mapping for one virtual page. To translate a virtual address to a physical address, therefore, the processor simply uses the virtual page number as an index into the page table from the given page table base address. Each translation is good for 512 virtual addresses since the byte within the virtual page corresponds to the byte within the physical page.

Figure 4-7 shows the format of a page table entry. The high-order bits are used to indicate the page's status and protection. The page's protection can be set to prevent read and/or write access by any mode (kernel, executive, supervisor, or user). The page's status indicates what the remainder of the page table entry means. It may be, for example, a page address in physical address space, a disk sector, or a temporary pointer to a page shared by two or more processes. The system's **virtual memory** is a dynamic memory that is defined by the physical memory and disk pages that are virtually mapped by page table entries.

The operating system's memory management software maintains the page table entry protection and status bits, with the exception of the modified page bit. The processor sets the modified page bit to indicate that it has written into a physical page in memory. This is used to keep disk I/O to a minimum when paging a process.

The processor uses the page table base registers to locate the page tables, and uses the length registers as a validity check to ensure that any given virtual page is in the range of defined page table entries. Figure 4-8 summarizes and compares the page table structures.

All process page tables have virtual addresses in the system region of virtual address space, but the system region page table is located by its address in physical memory. That is, the system region page table base register contains the *physical address* of the page table base, while the process page table base registers contain the *virtual addresses* of their page table bases. Because a per-process page table entry is referred to by a virtual address in the system region, the hardware translates its virtual address using the system page table.

There are two advantages to using a virtual address as the base address of a per-process page table. The first advantage is that all page tables do not have to reside in physical memory. The system region page table is the only page table that needs to be resident in physical memory. All process page tables can reside on disk; that is, process page tables can themselves be paged and swapped as necessary.

The second advantage is that the operating system's memory management software can allocate per-process page tables dynamically, because the per-process page tables do not need to be mapped into contiguous physical pages. And although the system region page table must be mapped into contiguous physical pages, this requirement does not restrict physical memory allocation. The region is shared among processes, and therefore does not require redefinition from context to context.

To illustrate the efficiency of this memory mapping scheme, suppose that 16 processes, each of which is us-

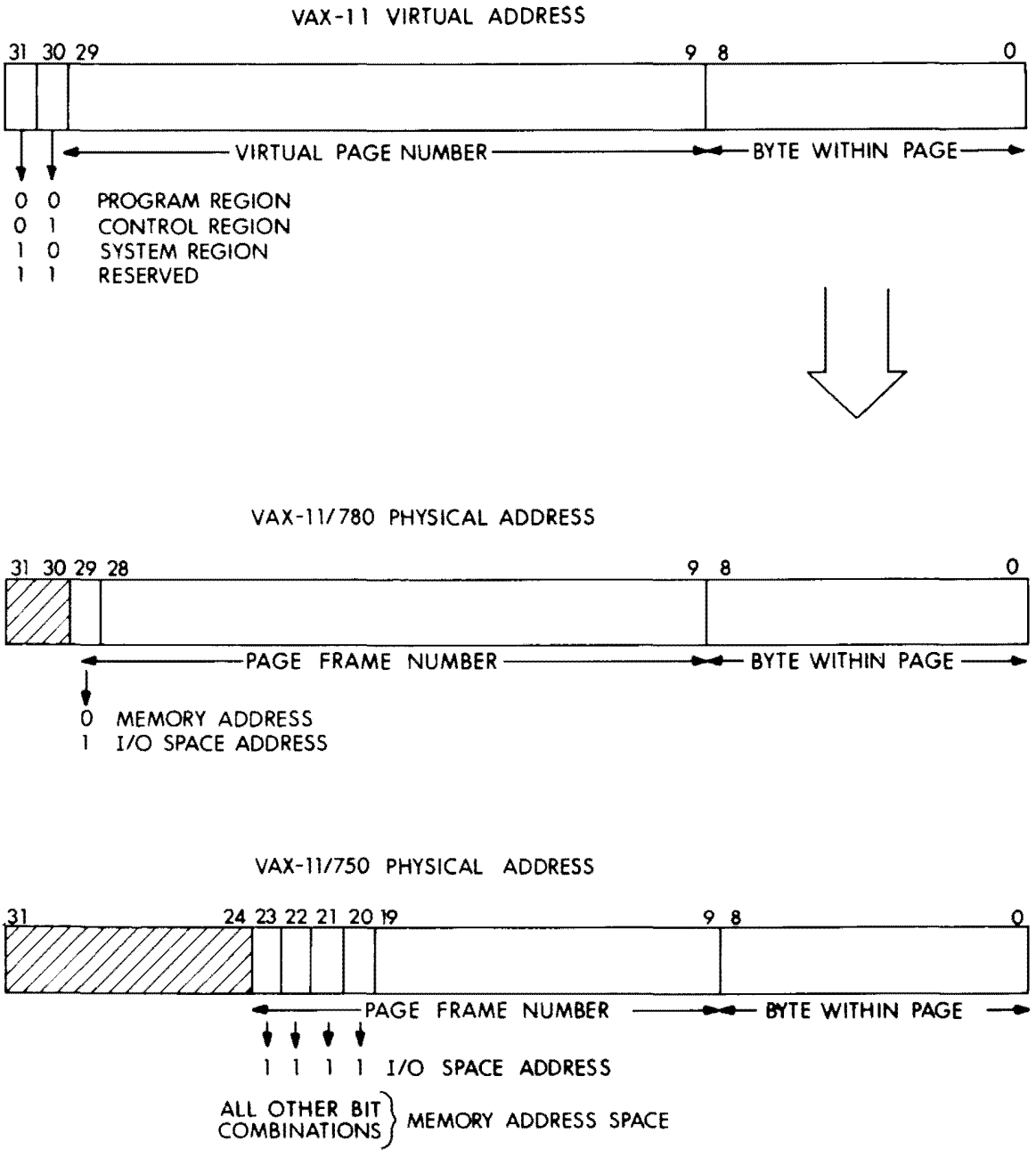


Figure 4-6

Virtual and Physical Address Formats

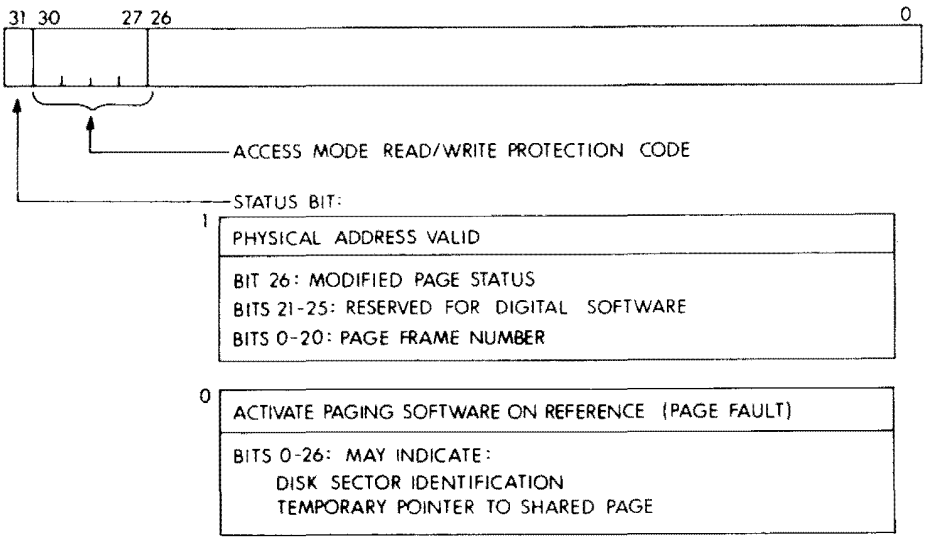


Figure 4-7

Page Table Entry

ing 4 million bytes of virtual address space, are known to the system at the same time (for a total of 64 Mb of virtual address space). One system page table entry maps one page of per-process page table entries, and one page of per-process page table entries maps 65,536 (64K) bytes of virtual address space (since it is possible to store 128 page table entries in a single page of memory). Therefore one page of system page table maps 128 pages of per-process page tables, which in turn maps 8 Mb of process virtual address space. Thus the system region page table needed to map these 16 processes requires approximately 8 physical pages (4K bytes) of memory.

Exception and Interrupt Vectors

The processor can automatically initiate changes in the normal flow of program execution. The processor recognizes two kinds of events that cause it to invoke conditional software: exceptions and interrupts. Some exceptions affect an individual process only, such as arithmetic traps, while others affect the system as a whole, for example, machine check. Interrupts include both device interrupts, such as those signaling I/O completion, and software-requested interrupts, such as those signaling the need for a context switch operation.

The processor knows which software to invoke when an exception or interrupt occurs because it references specific locations, called vectors, to obtain the starting address of the exception or interrupt dispatcher. The processor has one internal register, the System Control Block Base Register, which the operating system loads with the physical address of the base of the System Control Block, which contains the exception and interrupt vectors. The processor locates each vector by using a specific offset into the System Control Block. Figure 4-9 illustrates the vectors in the System Control Block. Each vector tells the processor how to service the event, and contains the system region virtual address of the routine to execute. Note that vector 14 (hex) can be used as a trap to writable control store to execute user-defined instructions, and the vector contains information passed to microcode.

Interrupt Priority Levels

Exceptions do not require arbitration since they occur synchronously with respect to instruction execution. Interrupts, on the other hand, can occur at any time. To arbitrate between interrupt requests that may occur simultaneously, the processor recognizes 31 interrupt priority levels.

The highest 16 interrupt priority levels are reserved for interrupts generated by hardware, and the lowest 16 interrupt priority levels are reserved for interrupts requested by software. Table 4-4 lists the assignment of each level, from highest to lowest priority. Normal user software runs at process level, which is interrupt priority level zero.

To handle interrupt requests, the processor enters a special system-wide context. In the system-wide context, the processor executes in kernel mode using a special stack called the interrupt stack. The interrupt stack cannot be referenced by any user mode software because the processor only selects the interrupt stack after an interrupt, and all interrupts are trapped through system vectors.

The interrupt service routine executes at the interrupt priority level of the interrupt request. When the processor receives an interrupt request at a level higher than that of the currently executing software, the processor honors the request and services the new interrupt at its priority level. When the interrupt service routine issues the REI (Return from Exception or Interrupt) instruction, the processor returns control to the previous level.

System Base Register
(contains the physical address of the first entry of the page table)

System Length Register
(contains the number of page table entries, N)

Program Region Base Register
(contains the virtual address of the first entry in the page table)

Program Region Length Register
(contains the number of page table entries, N)

Control Region Base Register
(contains the virtual address of base of the page table)

Control Region Length Register
(contains the virtual address of the first entry in the page table for virtual page number $2^{22}-N$, where N is the number of page table entries.)

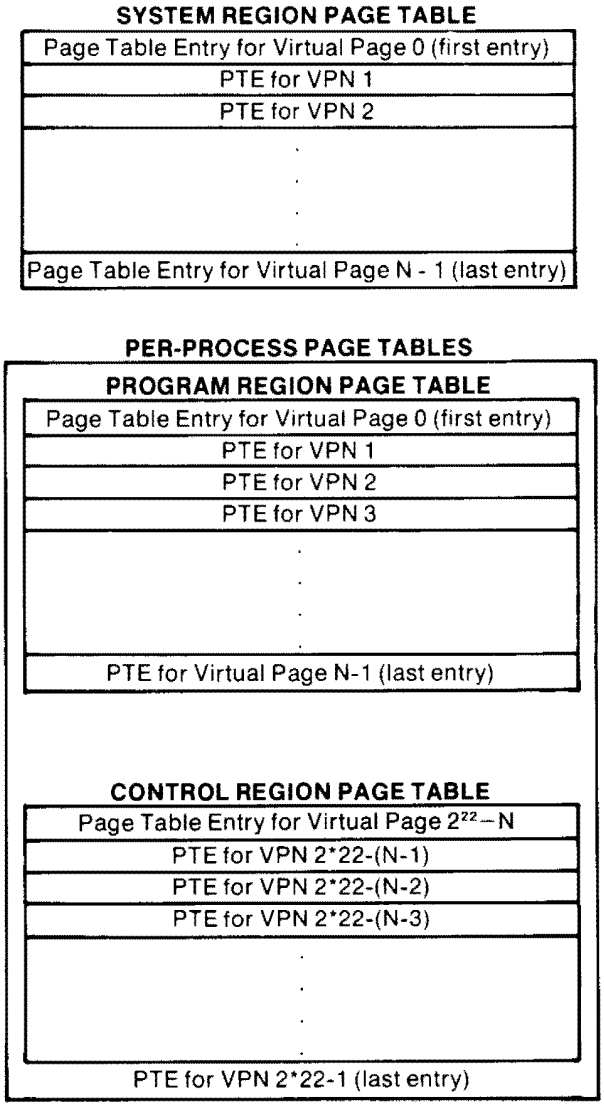


Figure 4-8
Page Tables

I/O Space and I/O Processing

An I/O device controller has a set of control/status and data registers. The registers are assigned addresses in physical address space, and their physical addresses are mapped, and thus protected, by the operating system's memory management software. That portion of physical address space in which device controller registers are located is called I/O space.

No special processor instructions are needed to reference I/O space. The registers are simply treated as locations containing integer data. An I/O device driver issues commands to the peripheral controller by writing to the controller's registers as if they were physical memory locations. The software reads the registers to obtain the controller status. The driver controls interrupt enabling and disabling on the set of controllers for which it is responsible. When interrupts are enabled, an interrupt occurs when the controller requests it. The processor accepts the interrupt request and executes the driver's interrupt service routine if it is not currently executing on a higher-priority interrupt level.

Process Context

For each process eligible to execute, the operating system creates a data structure called the **software process control block**. Within the software process control block is a pointer to a data structure called the **hardware process control block**. The hardware process control block is illustrated in Figure 4-10. It contains the hardware process context, that is, all the data needed to load the processor's

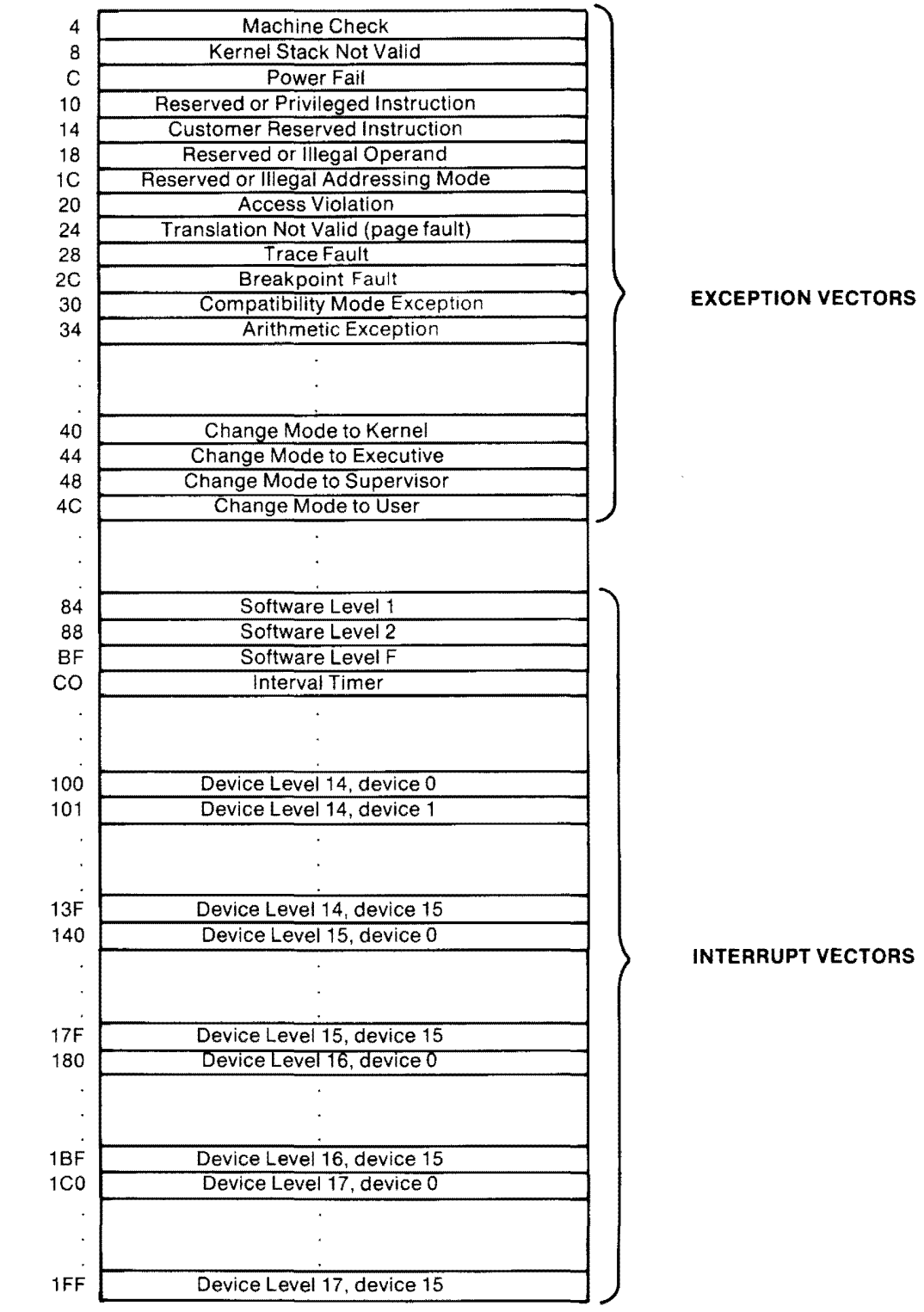


Figure 4-9

System Control Block

Table 4-4
Interrupt Priority Levels

PRIORITY Hex	PRIORITY Decimal	HARDWARE EVENT
1F	31	Machine Check, Kernel Stack Not Valid
1E	30	Power Fail
1D	29	Processor, Memory, or Bus Error
1C	28	
1B	27	
1A	26	
19	25	
18	24	Clock
17	23	UNIBUS BR7
16	22	UNIBUS BR6
15	21	UNIBUS BR6
14	20	UNIBUS BR4
13	19	Device Interrupt
12	18	
11	17	
10	16	
PRIORITY		**SOFTWARE EVENT**
0F	15	Reserved for DIGITAL
0E	14	
0D	13	
0C	12	
0B	11	Device Drivers
0A	10	
09	09	
08	08	
07	07	Timer Process
06	06	Queue Asynchronous System Trap (AST)
05	05	Reserved for DIGITAL
04	04	I/O Post
03	03	Process Scheduler
02	02	AST Delivery
01	01	Reserved for DIGITAL
00	00	User Process Level

process-specific registers when a context switch occurs. To give control of the processor to a process, the operating system loads the processor's Process Control Block Base register with the physical address of a hardware process control block and issues the Load Process Context instruction. The processor loads the process context in one operation and is ready to execute code within that context.

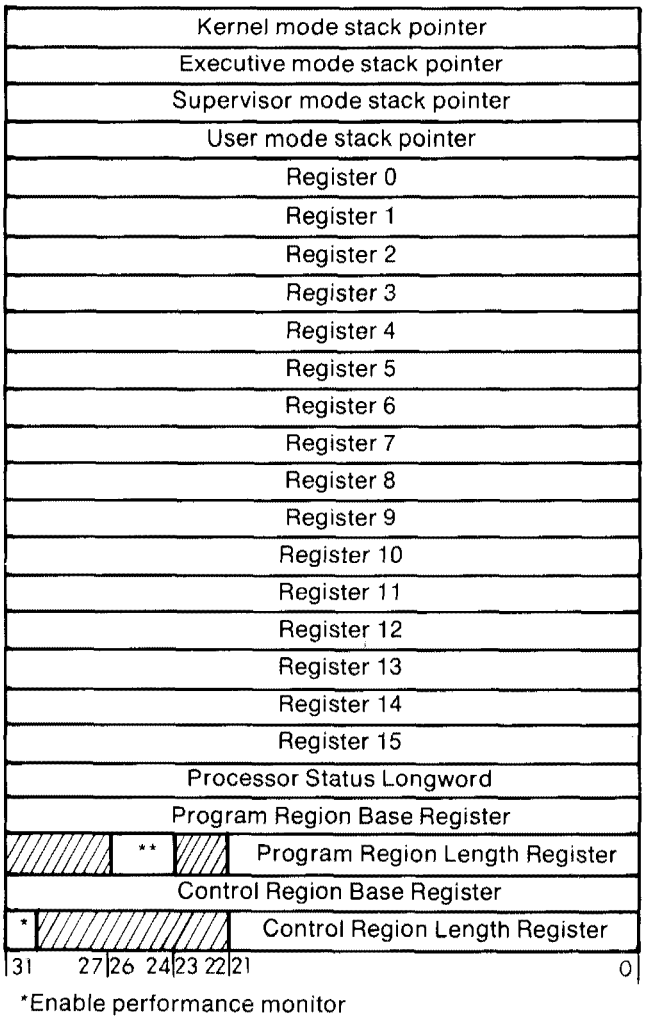
As can be seen from the illustration, a process control block not only contains the state of the programmable registers, it also contains the definition of the process virtual address space. Thus, the mapping of the process is automatically context-switched.

Furthermore, the process control block provides the mechanism for triggering asynchronous system traps to user processes. The Asynchronous System Trap field enables the processor to schedule a software interrupt to initiate an AST routine and ensure that they are delivered to the proper access mode for the process.

CONSOLE

The console is the operator's interface to the central processor. Using the console terminal, the operator can examine and deposit data in memory locations or the processor registers, halt the processor, step through instruction streams, and boot the operating system.

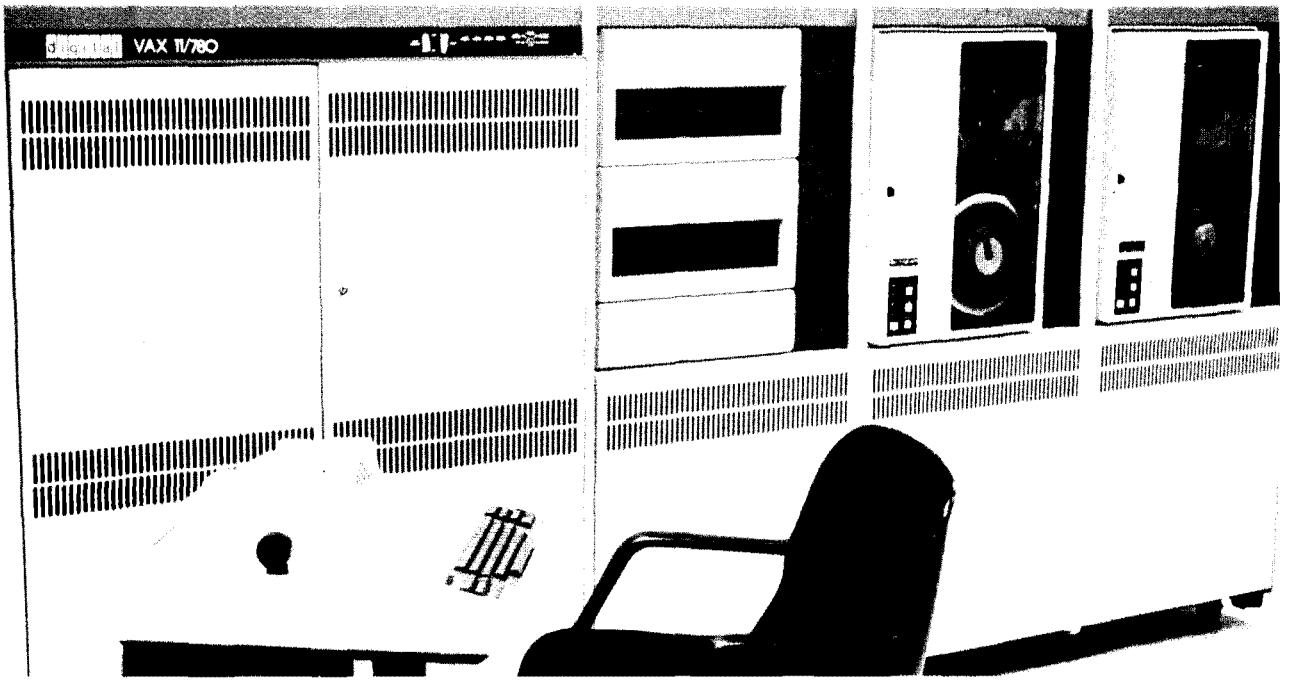
HARDWARE PROCESS CONTROL BLOCK



*Enable performance monitor
**Asynchronous System Trap pending

Figure 4-10

Hardware Process Control Block



THE VAX-11/780 PROCESSOR

INTRODUCTION

A VAX-11 processor is a specific set of hardware logic that performs the operations of the computer system according to the VAX-11 architecture.

This section describes the implementation-specific details of the VAX-11/780 processor. Its integrated components are:

- The Central Processing Unit (CPU) itself, including its cache, writable diagnostic control store, optional floating point accelerator, clocks and console.
- Main memory and main memory controllers.
- Input/output bus adaptors.
- Optional multiport memory.
- Optional high performance 32-bit interface.

These components communicate over a high-speed internal bus called the memory interconnect. Figure 4-11 illustrates the major processor components.

The Central Processing Unit performs the logical and arithmetic operations requested of the computer system. Its user programmable registers include sixteen 32-bit general purpose registers for data manipulation, and the Processor Status Longword for controlling the execution states of the CPU. The processor's instruction set is interpreted by the microcode contained in its control store.

The processor includes 12K bytes of writable diagnostic control store for updating the instruction set microcode. The writable diagnostic control is also used for storing microcode diagnostics, which can be loaded from the console's floppy disk.

The processor will also support 12K bytes of user writable control store (WCS). WCS is optionally available to the customer for augmenting the speed and power of the basic machine with customized functions.

The console enables the computer system operator to control the processor operation directly. The console actually consists of an LSI-11 microcomputer with 24K bytes of memory, a floppy disk system, and a terminal. A serial line interface is optionally available for remote diagnosis.

Two memory controllers can be be connected to the memory interconnect. Each controller handles up to 4096K bytes of semiconductor memory, for a system total of 8192K bytes of memory. The memory controllers employ an error detecting and correcting technique that ensures correction of all single-bit errors and detection of all double-bit errors.

In addition, VAX-11/780 supports the multiported memory option. Multiported memory supports very high throughput interprocessor communications. Multiported memory is discussed more fully in the Peripherals section.

Three I/O bus adaptors can be interfaced to the memory interconnect: an adaptor for the MASSBUS, which connects high-speed disk and magnetic tape devices to the processor; an adaptor for the UNIBUS, which connects lower-speed devices to the processor, including disks, communications lines, and I/O peripherals such as terminals, line printers, and card readers; and an optional adaptor for the high performance 32-bit interface. The high per-

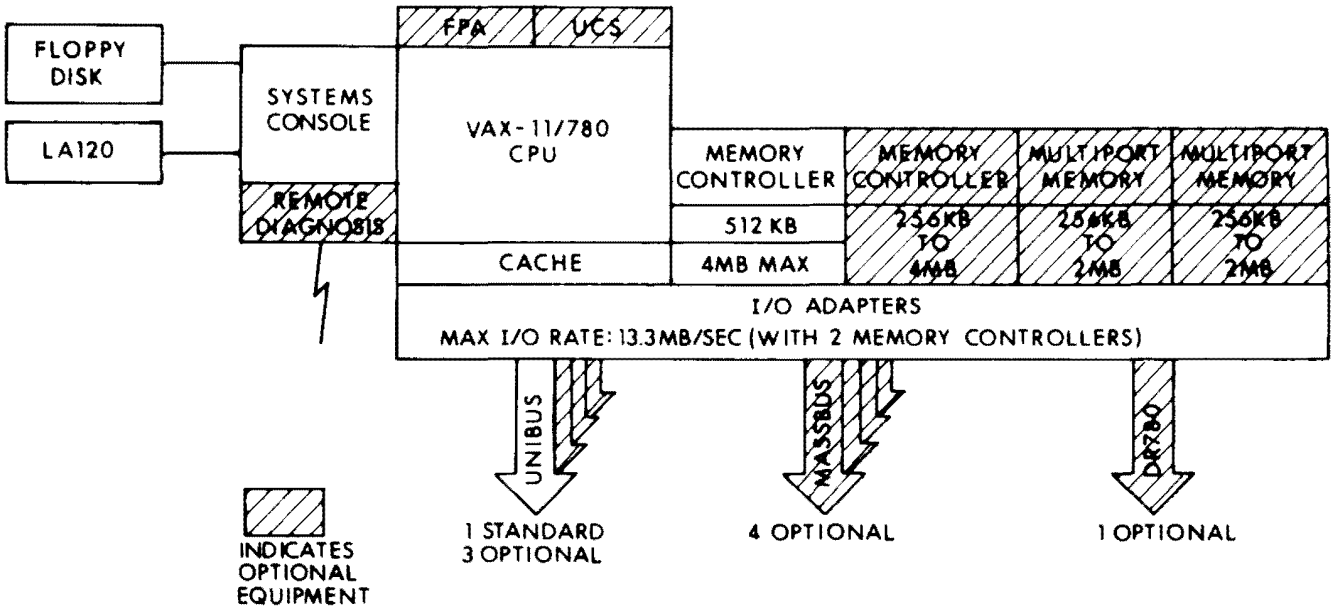


Figure 4-11
VAX-11/780 Processor

formance interface enables the user to interface custom devices directly to the memory interconnect, or to connect two VAX-11/780 systems together. The high performance interface will be discussed more thoroughly in the Peripherals section.

VAX-11/780 PROCESSOR COMPONENTS

Described below are the major hardware components of the VAX-11/780 processor.

VAX-11/780 Console

The VAX-11/780's integrated console consists of an LSI-11 microcomputer with 16K bytes of read/write memory and 8K bytes of ROM (used to store the LSI diagnostic, the LSI bootstrap, and fundemental console routines), a floppy disk system (for the storage of basic diagnostic programs and software updates), a hard-copy terminal, and an optional remote diagnosis port.

The console is further used for updating the software with maintenance releases and for loading optional software products distributed on floppy disk.

The operator communicates with the VAX-11/780 console via a set of user-oriented, English-like commands known as the console command language (CCL).

An EIA serial line interface and modem can be added to the console to provide remote diagnosis.

VAX-11/780 MEMORY INTERCONNECT

The memory interconnect is the system's internal bus, conveying addresses, data, and control information between the processor and memory, and between memory and the I/O controllers. The memory interconnect has a cycle time of 200 nanoseconds and can transfer 32 bits each cycle. Data transfers use two consecutive cycles to transfer 64 bits at a time. The maximum memory interconnect transfer rate is 13.3 million bytes per second. The memory interconnect provides an unusual degree of throughput and reliability because it uses:

- time-division multiplexing
- distributed priority arbitration
- parity and protocol checking on every transfer
- transaction history recording

The protocol, or sequence in which operations occur on the memory interconnect, is time-division multiplexed to increase the effective bus bandwidth. Time-division multiplexing means that the transactions which constitute one transfer operation are interleaved with the transactions which constitute another transfer operation. Thus, several operations can be in progress over the same period of time. For example, the CPU can ask a memory controller to read some data; the same memory controller might then transfer previously requested data to an I/O device before it transfers the requested data to the CPU.

In some systems, the processor bus can be tied up for each transfer because a requester acquires the bus to send an address and then keeps the bus while it waits for the requested data. In VAX-11/780, the bus is not held inactive during the data access time because bus ownership is relinquished after every cycle. A requester acquires the bus to specify an operation and send an address, and then relinquishes the bus. At some time later the responder acquires the bus to send back the requested data. In the interim, any number of other transactions can be initiated or completed. This and the fact that transactions are buffered

make it possible for the bus to operate at its full bandwidth.

Arbitration on the memory interconnect is distributed, which ensures that no unit is critical to bus operation. Every unit on the memory interconnect has its own arbitration line. Arbitration lines are ordered by priority and every unit monitors all the arbitration lines each cycle to determine if it will get the next cycle. Unlike some bus systems, any unit on the memory interconnect (except the CPU clock) can fail without causing a failure of the entire bus.

To ensure the integrity of the signals transmitted, the memory interconnect includes several error checking and diagnostic mechanisms, such as:

- parity checking on data, addresses, and commands
- protocol checking in each interface
- a history silo of the last 16 memory interconnect cycles

VAX-11/780 MAIN MEMORY AND CACHE SYSTEMS

The processor includes both main memory systems and cache memory systems. Transactions between main memory and the processor take place over the memory interconnect. The cache memory systems are internal to the processor.

Main Memory

Main memory consists of arrays of MOS RAM integrated circuits with a cycle time of 600 nanoseconds. A memory controller can access a maximum of 4,194,304 bytes (4M bytes). Two memory controllers can be connected to the memory interconnect, yielding a maximum of 8M bytes of physical memory that can be available on the system. The maximum total physical address space is 2^{29} or approximately 512 million bytes. However, the minimum required memory is 256K bytes, which is then expandable in increments of 256K bytes.

A memory controller will buffer one command while it processes another to increase system throughput. Main memory can also be interleaved (where two memory controllers are each addressing the same amount of memory) to increase the available memory bandwidth. The memory system employs error checking and correction (ECC) that corrects all single bit errors and detects all double bit errors.

When the system is powered down, an ac standby current is normally used to retain the contents of memory. In case of temporary AC power interruption, an optional backup battery is also available to provide 10 minutes of power for up to 4M bytes of memory so that the contents of main memory are not destroyed. Two backup batteries provide power for up to 8M bytes of memory.

Data are fetched from main memory 64 bits at a time (two memory interconnect cycles) and cached in the processor's internal memory systems. The internal memory systems include a main memory cache, an address translation buffer, and an instruction lookahead buffer.

Memory Cache

The memory cache is the primary cache system for all data coming from memory, including addresses, address translations, and instructions. The memory cache is an 8K byte, two-way set associative, write-through cache.

Write-through provides reliability because the contents of main memory are updated immediately after the processor performs a write. Most write-through cache systems tie up the processor while main memory is updated. However, the VAX-11/780 processor buffers data to be written to memory to avoid waiting while main memory is updated from the cache. Therefore, while providing the reliability of a write-through cache, this system also provides much the same performance as a write-back cache.

Memory cache significantly reduces processor wait time since practically all of the time, (greater than 95%), the data are in the cache. The cache memory system carries byte parity for both data and addresses for increased integrity.

Address Translation Buffer

The address translation buffer is a cache of virtual to physical address translations. It significantly reduces the amount of time spent by the CPU on the repetitive task of dynamic address translation. The cache contains 128 virtual-to-physical page address translations which are divided into equal sections: 64 system space page translations and 64 process space page translations. Each of these sections is two-way associative. There is byte parity on each entry for increased integrity.

Instruction Buffer

The 8-byte instruction buffer improves CPU performance by prefetching data in the instruction stream. The control logic continuously fetches information from memory or cache, where possible, to keep the 8-byte buffer full. It effectively eliminates the time spent by the CPU waiting for two memory cycles where bytes of the instruction stream cross 32-bit longword boundaries. In addition, the instruction buffer processes operand specifiers in advance of execution and subsequently routes them to the CPU.

I/O CONTROLLER INTERFACES

Peripherals can be connected to the processor's memory interconnect bus in either of two ways: through the MASSBUS, for high-speed disk and/or magnetic tape devices, or through the UNIBUS, for a variety of I/O devices, including line printers, disks, card readers, terminals, and interprocessor communication links.

VAX-11 MASSBUS Interface

The processor interface for a MASSBUS peripheral is the **MASSBUS adaptor**. The MASSBUS adaptor performs control, arbitration, and buffering functions. Up to four MASSBUS adaptors can be connected to the memory interconnect. The MASSBUS is typically used to attach high-speed disk or magnetic tape devices.

Each MASSBUS adaptor includes its own address translation map that permits scatter/gather disk transfers. In scatter/gather transfers, physically contiguous disk blocks can be read into or written from discontiguous blocks of memory. The translation map contains the addresses of the pages, which may be scattered throughout memory, from or to which the contiguous disk transfer takes place.

Each MASSBUS adaptor includes a 32-byte silo data buffer. Data are assembled in 64-bit quadwords (plus parity) to make efficient use of the memory interconnect bandwidth. On transfers from memory to a MASSBUS peripheral, the

MASSBUS adaptor anticipates upcoming MASSBUS data transfers by fetching the next 64 bits from memory before all of the previous data have been transferred to the peripheral.

On-line diagnostics and built-in loop-back testing enable fault isolation of the MASSBUS adaptor for any of its function circuits without a drive on the MASSBUS.

VAX-11/780 UNIBUS Interface

All devices other than the high-speed disk drives and magnetic tape transports are connected to the UNIBUS, an asynchronous bidirectional bus. These include all DIGITAL- and user-developed real-time peripherals. The UNIBUS is connected to the memory interconnect through the **UNIBUS adaptor**. The UNIBUS adaptor does priority arbitration among devices on the UNIBUS. Up to four UNIBUS adapters can be placed on the memory interconnect.

The UNIBUS adaptor provides access from the VAX-11/780 processor to the UNIBUS peripheral device registers by translating UNIBUS addresses, data transfer requests, and interrupt requests to their memory interconnect equivalents, and vice versa. The UNIBUS adaptor address translation map translates an 18-bit UNIBUS address to a 30-bit memory interconnect address. The map provides direct access to system memory for nonprocessor request UNIBUS peripheral devices and permits scatter/gather disk transfers.

The UNIBUS adaptor enables the processor to read and/or write the peripheral controller registers. In some cases this constitutes the transfer.

To make the most efficient use of the memory interconnect bandwidth, the UNIBUS adaptor provides buffered direct memory access data paths for up to 15 nonprocessor request (NPR) devices. Each of these channels has a 64-bit buffer (plus byte parity) for holding four 16-bit transfers to and from UNIBUS devices. The result is that only one memory interconnect transfer (64 bits) is required for every four UNIBUS transfers. The maximum aggregate transfer rate through the buffered data paths is 1.35 million bytes per second. On memory interconnect-to-UNIBUS transfers, the UNIBUS adaptor anticipates upcoming UNIBUS requests by pre-fetching the next 64-bit quadword from memory as the last 16-bit word is transferred from the buffer to the UNIBUS. By the time the UNIBUS device requests the next word, the UNIBUS adaptor has it ready to transfer.

Any number of unbuffered direct memory access transfers are handled by one Direct Data Path. Every 8- or 16-bit transfer requires one 32-bit transfer on the memory interconnect. The maximum transfer rate through the Direct Data Path is 500,000 bytes per second.

The UNIBUS adaptor permits concurrent program interrupt, unbuffered, and buffered data transfers. The aggregate throughput rate of the Direct Data Path, plus the 15 buffered data paths, is 1.35 million bytes per second.

Data Throughput

VAX-11/780 includes many features that support high data throughput, including silo data buffers for MASSBUS peripheral controllers, buffered direct memory access for the UNIBUS peripherals, and 64-bit data transfers and prefetching.

Memory bandwidth matches that of the processor's internal bus — 13.33 million bytes per second, including time for refresh cycles. This is primarily because of the memory controller request buffers, which substantially increase memory throughput and overall system throughput, and decrease the need for interleaving for most configurations. Memory interleaving, which is enabled and disabled under program control, can be used effectively when more than two MASSBUS peripheral controllers are connected and the MASSBUS and UNIBUS devices are transferring at very high rates — greater than one million bytes per second.

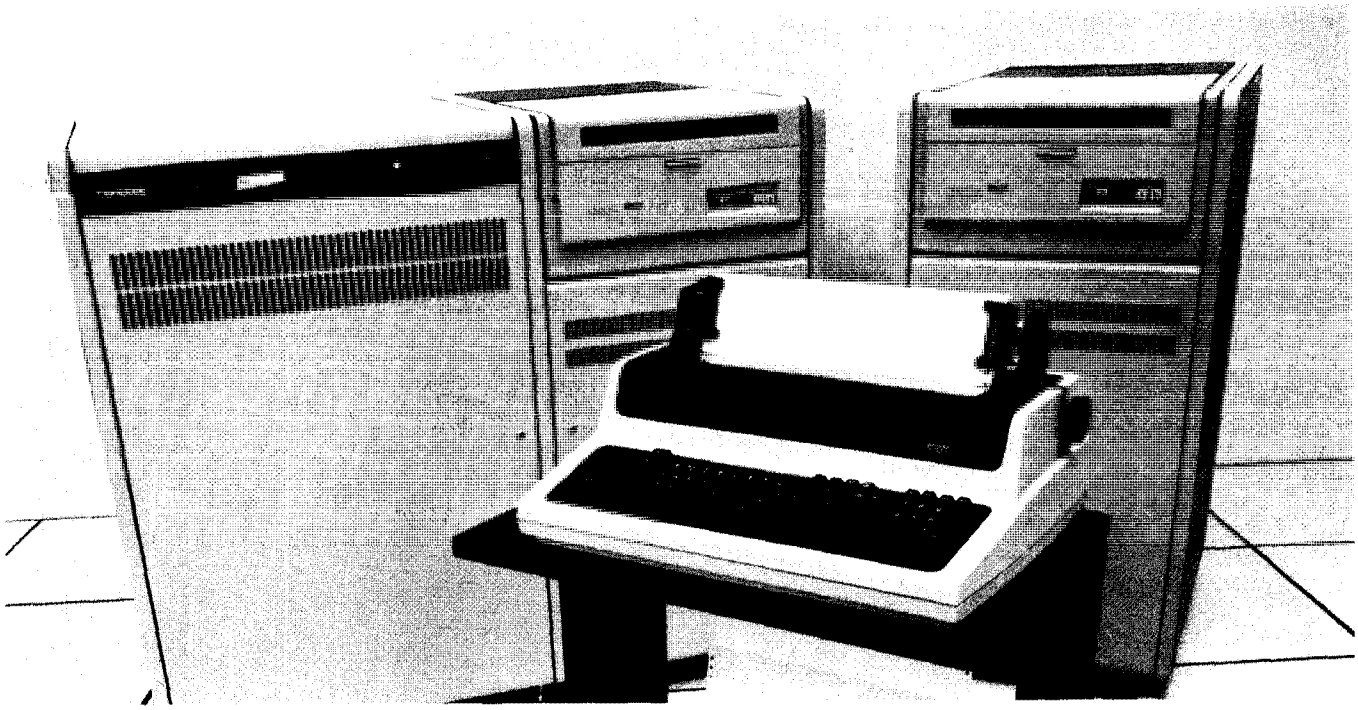
The operating system supports the hardware throughput in its I/O request processing software. The software uses the processor's multiple hardware priority levels to increase I/O response time, and keeps each disk controller as busy as possible by overlapping seek requests with I/O transfers.

VAX-11/780 FLOATING POINT ACCELERATOR

The floating point accelerator (FPA) is an optional high-speed processor enhancement. When included in the processor, the floating point accelerator accelerates the execution of the addition, subtraction, multiplication, and division instructions that operate on single- and double-precision floating point operands. This includes the special EMOD and POLY instructions in both single- and double-precision formats. Additionally, the floating point accelerator enhances the performance of the 32-bit integer multiply instruction _MUL.

The processor does not have to include the floating point accelerator to execute floating point operand instructions; the FPA increases the execution speed of floating point instructions. The floating point accelerator can be added or removed without changing any existing software.

When the floating point accelerator is included in the processor, a floating point register-to-register add instruction takes as little as 800 nanoseconds to execute. A register-to-register multiply instruction takes as little as one microsecond. The inner loop of the POLY instruction takes approximately one microsecond per degree of polynomial.



THE VAX-11/750 PROCESSOR

INTRODUCTION

A VAX-11 processor is a specific set of hardware logic that performs the operations requested of the computer system according to the VAX-11 architecture.

This section describes the implementation specific details of the VAX-11/750 processor. Its integrated components are:

- the Central Processing Unit (CPU) itself, including its cache, optional user control store, clocks and console
- main memory and main memory controllers
- peripheral bus adaptors

Figure 4-12 illustrates the major VAX-11/750 processor components.

The central processing unit performs the logical and arithmetic operations requested of the computer system. Its user programmable registers include sixteen 32-bit general purpose registers for data manipulation, and the Processor Status Word for controlling the execution states of the CPU. The processors instruction set is defined by the microcode contained in its control store.

The optional User Control Store includes 10K bytes (1Kbytes of 80 bit microwords) of writeable storage. This allows customers to augment the speed and power of the basic machine with customized microcode functions. Digital offers a loadable microcode package for extended precision floating point arithmetic operations (G- and H-floating point data types) on the 11/750.

The console enables the computer system operator to control the processor operation directly. The console subsystem consists of the console terminal (LA38 DECwriter), the front panel, the user oriented console command language, and a TU58 Tape Cartridge Drive. Also optionally available for the console is the remote diagnosis interface.

The main memory subsystem consists of ECC MOS memory, which is interfaced to the system via the memory controller. MOS memory may be added to the system in increments of 256K bytes to a maximum of 2M bytes.

The I/O subsystem consists of the UNIBUS and MASSBUS devices connected to the system via special buffered interfaces called adaptors. Each VAX-11/750 system contains one UNIBUS adapter for standard peripherals and up to a maximum of three MASSBUS adapters for high speed peripherals.

VAX-11/750 PROCESSOR COMPONENTS

Described below are the major hardware components of the VAX-11/750 processor.

VAX-11/750 Console

The console enables the computer system operator to control the processor operation directly. The console subsystem consists of the console terminal (LA38), the front panel, the user oriented console command language, and a TU58 Tape Cartridge Drive. Simple console commands, entered through the console terminal, replace the traditional toggle switches and provide operational control (i.e., bootstrapping, initialization, self testing, examining and depositing data in memory,etc.). When not performing op-

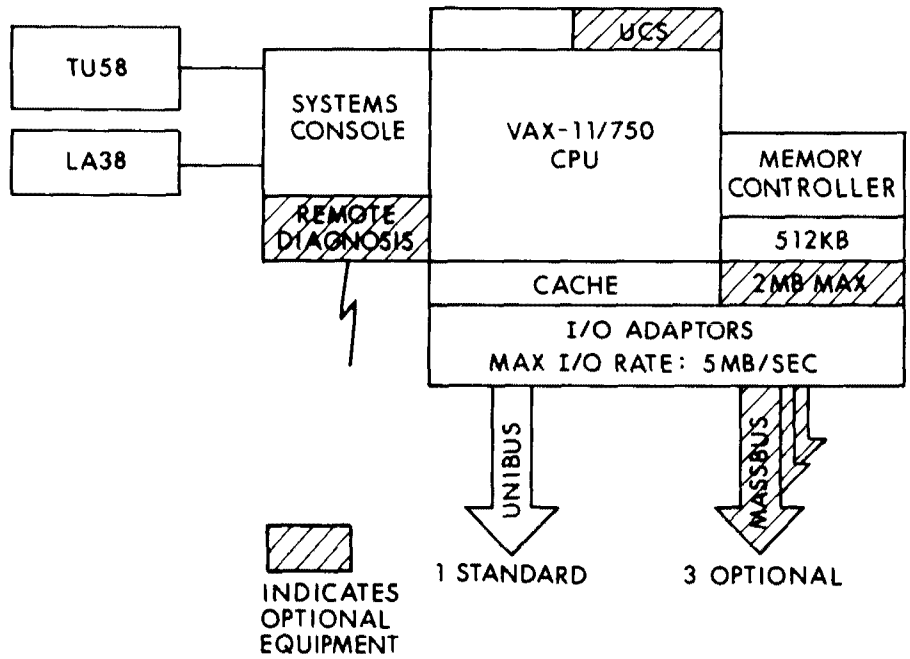


Figure 4-12
VAX-11/750 Processor

erator functions or error logging, the same terminal can be available to authorized users for normal system operations.

The VAX-11/750 console subsystem and the console command language also facilitate the loading of diagnostics and software updates from the TU58 Tape Cartridge. For those customers subscribing to a DIGITAL maintenance contract, the console subsystem may also be equipped with a remote diagnosis module (RDM) allowing the VAX-11/750 to interface to a host computer at a DIGITAL Diagnostic Center for remote fault detection or preventive maintenance procedures.

VAX-11/750 Main Memory

The VAX-11/750 main memory is built using 16K MOS RAM (random access memory) LSI chips. Physical memory is organized into an array of 32-bit longwords plus an additional 7 bits per longword dedicated to ECC (error correcting code). ECC allows the correction of all single-bit errors and the detection of all double bit errors to insure data integrity. Main memory is interfaced to the VAX system via the memory controller. The VAX-11/750 can be easily field upgraded to 2M bytes of main memory by simply adding 256K byte expansion modules.

VAX-11/750 Cache Systems

The VAX-11/750 CPU provides three cache systems: the main memory cache, the address translation buffer, and the instruction buffer.

- Main Memory Cache

Memory cache (typically 90% hit rate) provides the central processor with high-speed data access by storing frequently referenced addresses, data and instruction items. The memory cache typically reduces memory access time in half.

The VAX-11/750 memory cache is a 4K byte, direct mapped, write-through cache. It is used for all data coming from memory, including addresses and instructions. The write-through feature protects the integrity of memory because memory contents are updated immediately after the processor performs a write. For increased integrity, the cache memory system carries byte parity for both data and addresses. Cache locations are allocated when data is read from memory or when a longword is written to memory. Memory cache also watches I/O transfers and updates itself appropriately. Therefore, no operating system overhead is needed to synchronize the cache with I/O operations, i.e., memory cache is transparent to all software.

- Instruction Buffer

The instruction buffer is an 8 byte buffer that enables the CPU to fetch and decode the next instruction while the current instruction completes execution. The instruction buffer in combination with the parallel data paths (which can perform integer arithmetic and shifting operations simultaneously) significantly enhances the VAX-11/750's performance because the CPU is not held in a wait state.

- Address Translation Buffer

The address translation buffer is a cache of the most frequently used 512 physical address translations. It significantly reduces the amount of time the CPU spends on the repetitive task of dynamic address translation. The cache contains 512 virtual-to-physical page address translations which are divided into equal sections: 256 system space page translations and 256 process space page translations. Each of these sections is two-way associative and has parity on each entry for increased integrity.

Peripheral Controller Interfaces

Peripherals can be connected to the processor in either of two ways: through the MASSBUS, which conveys signals

to and from high-speed disks or magnetic tape devices, or through the UNIBUS, which conveys signals to and from a variety of I/O devices, including line printers, disks, card readers, tapes, terminals, and interprocessor communication links.

VAX-11 MASSBUS Interface

The processor interface for a MASSBUS peripheral is the **MASSBUS adaptor**. The MASSBUS adaptor performs control, arbitration, and buffering functions. There may be a total of three MASSBUS adapters on each VAX-11/750 system.

Each 11/750 MASSBUS adaptor includes its own address translation map that permits scatter/gather disk transfers. In scatter/gather transfers, physically contiguous disk blocks can be read into or written from discontiguous blocks of memory. The translation map contains the addresses of the pages, which may be scattered throughout memory, from or to which the contiguous disk transfer takes place.

Each 11/750 MASSBUS adaptor includes a 32-byte silo data buffer. Data are assembled in 32-bit longwords (plus parity) to make efficient use of the system bus. On transfers from memory to a MASSBUS peripheral, the MASSBUS adaptor anticipates upcoming MASSBUS data transfers by fetching the next 32 bits from memory before all of the previous data are transferred to the peripheral.

On-line diagnostics and loop-back enable adaptor fault isolation without requiring the use of a drive on the MASSBUS.

VAX-11/750 UNIBUS Interface

General purpose peripherals and customer developed devices are connected to the VAX-11/750 system via the UNIBUS. Since the 11/750 memory deals in 24-bit physical addresses (16M byte physical address space), 18-bit UNIBUS addresses must be translated to 24 bit memory addresses. This mapping function is performed by the UNIBUS adapter (a special hardware interface between memory and the UNIBUS) which translates UNIBUS addresses to their memory equivalents, and vice versa.

The UNIBUS adapter performs priority arbitration among devices on the UNIBUS, a function handled by the central processor in PDP-11 systems. The address translation map permits contiguous disk transfers to and from noncontiguous pages of memory (these are called scatter/gather operations). Interrupts on the VAX-11/750 UNIBUS are directly vectored into the appropriate process handler.

The UNIBUS adapter allows two kinds of data transfers; program interrupt and direct memory access (DMA). To make the most efficient use of the memory bandwidth, the UNIBUS adapter facilitates high-speed DMA transfers by providing buffered DMA data paths for up to 3 high-speed devices at one time. Each of these channels has a 32-bit buffer (plus byte parity) for holding two 16-bit transfers to or from UNIBUS devices. The result is that only one memory transfer (32 bits) is required for every two UNIBUS transfers. The maximum aggregate transfer rate through the buffered data paths is 1.5M bytes per second.

Any number of unbuffered DMA transfers are handled by one direct DMA data path. Every 8- or 16-bit transfer on the UNIBUS requires a 32-bit memory transfer (although only 16 bits are used). The maximum transfer rate through the direct data path is 1M bytes per second.

It should be noted that the UNIBUS adapter permits program interrupts, unbuffered and buffered data transfers to occur concurrently.

5
The Peripherals

The VAX system supports high-performance mass storage devices for on-line data retrieval, unit record equipment for data processing, terminals and line interfaces for the interactive user, direct memory access interfaces for real-time users and a line interface for interprocessor communications.

The mass storage systems provide large capacity and high throughput. Each MASSBUS adapter can support up to eight disk drives or seven disk drives and one magnetic tape controller. In addition, up to eight medium-capacity disk drives can be connected to the system's UNIBUS. VAX/VMS overlaps seeks on all multiple-drive disk configurations, performs multiple-block I/O transfers, and allows the user to control buffering, positioning, and blocking.

Card readers and line printers can be spooled input and output devices managed by operator-controlled queues. The LP11 and LA11 series line printers provide a range of high-speed and low-cost printer models. Up to four LP11 printers and up to 16 LA11 printers can be used on the system.

The system supports full-duplex handling for both hard copy and video terminals. The LA120 is a hard-copy terminal which offers moderate throughput and advanced print features; the VT100 video terminal offers a variety of controllable character and screen attributes including 24 lines by 80 columns or 14 lines by 132 columns screen sizes, smooth scrolling, and split screen. The system can support up to 96 terminals.

The DMC11 serial synchronous communications line provides high-performance point-to-point interprocessor connection using the DIGITAL Data Communications Message Protocol (DDCMP). The DMC11 ensures reliable data transmission and relieves the host processor of the details of protocol operation. For very high-performance interprocessor communications, the VAX-11/780 offers both multiport memory (MA780) and a high-speed channel interface (DR780). The DR780 can also be used for interfacing customer devices which require transfer rates of up to 6.67M bytes/second.

For real-time applications, VAX supports the LPA11-K and DR11-B direct memory access (DMA) interfaces. These devices reduce CPU involvement in I/O operations and speed the transfer of data between external devices and computer memory. The LPA11-K is an intelligent (dual-microprocessor) controller which provides high speed data sampling, operates in both dedicated and multirequest mode, and supports a number of peripheral devices. The DR11-B is a general purpose interface which performs high speed block data transfers between the VAX memory and user peripheral devices.

All equipment is integrated with the software system, and is supported by both on-line error logging and diagnostics. Each component includes extensive error checking and correction features. The software provides power failure and error recovery algorithms.

COMPONENTS

VAX supports four types of peripheral subsystems:

- Mass storage peripherals such as disk and magnetic tape
- Unit record peripherals such as line printers and card readers
- Terminals and terminal line interfaces
- Interprocessor communications links

All peripheral device control/status registers (CSRs) are assigned addresses in physical I/O space. No special processor instructions are needed for I/O control. In addition, all device interrupt lines are associated with locations that identify each device's interrupt service routine. When the processor is interrupted on function request completion, it immediately starts executing the appropriate interrupt service routine. There is no need to poll devices to determine which device needs service.

Devices use either one of two types of data transfer techniques: direct memory access or programmed interrupt request. The mass storage disk and magnetic tape devices and the interprocessor communications link are capable of direct memory access (DMA) data transfers. The DMA devices are also called **non-processor request** (NPR) devices because they can transfer large blocks of data to or from memory without processor intervention until the entire block is transferred.

The unit record peripherals and terminal interfaces are called **programmed interrupt request** devices. These devices transfer one or two bytes at a time to or from assigned locations in physical address space. Software then transfers the data to or from a buffer in physical memory.

MASS STORAGE PERIPHERALS

The mass storage peripherals include various capacity moving head disk drives and various speed magnetic tape transports:

- the high speed, large capacity RP06 and RM05 disk drives
- the high speed, medium capacity RM03 disk drive
- the medium speed, smaller capacity RL02, and RK07 disk drives
- the RX02 floppy disk
- the TE16, TU45, and TU77 magnetic tape transports
- the TS11 magnetic tape transport

The RM03, RP06, and RM05 disks and the TE16, TU45, and TU77 magnetic tape controllers are MASSBUS peripheral devices. The RX02 floppy disk, the RL02, and RK07 disk drives, and the TS11 tape subsystem are UNIBUS peripheral devices. Each MASSBUS can support up to eight device controllers; eight disk controllers with one drive each or seven disk drives and one magnetic tape formatter with up to eight tape transports.

To support the performance and reliability features of the system's disk and magnetic tape devices, the operating system's disk and magnetic tape device drivers provide:

- overlapped seeks for increased throughput on controllers with multiple disk drives
- overlapped magtape operations (write on one transport while another rewinds, for example)
- multiple block non-contiguous I/O transfers for file-structured devices
- read and write checks on a per-request, per-file, and/or volume basis
- extensive error recovery algorithms (e.g., ECC and offset recovery for disk, NRZI error correction for magnetic tape)
- logging of all device errors
- dynamic bad block support for file-structured disk devices
- volume mount verification after a change in drive status (off/on-line, powerfail)
- powerfail recovery for on-line drives, including repositioning of magnetic tape transports

Table 5-1
Disk Devices

DISK	RX02	RL02	RK07	RM03	RP06	RM05
Pack capacity:	512 Kbytes	10.4 Mbytes	28 Mbytes	67 Mbytes	176 Mbytes	300 Mbytes
Peak transfer rate (/sec):	55 Kbytes	512 Kbytes	538 Kbytes	1200 Kbytes	806 Kbytes	1200 Kbytes
Ave. seek time:	263ms	55ms	36.5ms	30ms	30ms	30ms
Ave. rotational latency:	83ms	12.5ms	12.5ms	8.3ms	6ms	8.3ms

For applications requiring special data reliability checks, the programmer can implement user written error recovery procedures without having to write unique device driver routines. The operating system driver's normal error recovery retry and error logging operations can be inhibited. If any error occurs when the recovery functions are inhibited, the driver immediately terminates the I/O operation and returns a failure status. User software can then perform its own recovery or logging procedures, since all the hardware diagnostic operations are available to jobs granted the diagnostic privilege by the system manager.

Disks

The disk subsystems provide high performance and reliability. They feature accurate servo positioning, error correction, and offset positioning recovery. Table 5-1 summarizes the capacities and speeds of the disk devices.

All disk drives use top-loading removable media. The RM03, RP06, and RM05 disk drives can be mixed on the same MASSBUS.

The UNIBUS accepts RK06, RK07, and RL02 disk drives and the RX02 floppy disk. The RX02 is the smallest capacity disk available, while the RK07 is the largest capacity disk. Up to eight RX02, RL02, RK06, and RK07 disk drives can be mixed in any combination on the same controller. In small system configurations where the RK06 or RK07 is used as the systems device, two drives are required in the configuration.

To decrease the effective access time and increase throughput, the operating system's disk device drivers provide overlapped seeks for all disk units on a controller. All I/O transfers, including write checks, are preceded by a seek, except when the seek is explicitly inhibited by diagnostic software. On MASSBUS devices, seeks to any unit can be initiated at any time and do not require controller intervention. During seeks, the controller is free to perform a transfer on any unit other than the one on which the seek is active. If a data transfer was in progress at the time of completion, the driver processes the attention interrupts caused by seek completion when the controller is free.

The device unit notifies the driver when it detects a read error that can be recovered using its error correction code (ECC). It provides the position and pattern of any error burst of up to 11 bits within the data field. The driver applies the error correction to the data in memory. The transfer continues as if the error had not occurred.

In addition to overlapping seeks with data transfers, the driver also overlaps offset error recovery with normal controller operation. Offset recovery enables the driver to reposition the head on the track to pick up a stronger signal on a sector during a read operation. Provided that retry is not inhibited, the driver performs offset recovery automatically when a read error occurs that can not be corrected using the hardware ECC.

The driver logs all errors, including those from which it successfully recovers. The driver also supplies dynamic bad block handling for virtual I/O (Files-11 file-structured) operations. When a bad block is detected, the information is stored in the file header. The bad block is recorded in the bad block file when the file is deleted.

In addition to the driver's dynamic bad block handling, the system includes an on-line static bad block utility and on-line diagnostics for verifying drive level functions.

Magnetic Tape

The TE16, TU45, and TU77 are high performance MASSBUS tape storage subsystems, which share the following characteristics:

- program-selectable 1600 or 800 bpi, 9-track data storage
- industry compatible data formats
- reading in reverse (as well as forward)
- parity, longitudinal, and cyclic redundancy checking
- NRZI error correction

The TU45 and TE16 are identical in capacity: both allow up to 40 million bytes per tape reel and both allow up to 8 tape drives per formatter. However, the TU45 offers substantially higher speed and throughput. Read/write speeds for the TU45 and TE16 are 75 and 45 inches/second respectively; their data transfer speeds are 120K and 72K bytes per second.

The TU77 tape storage system can perform read/writes of data at the rate of 125 inches/second, while allowing a peak transfer rate of 200K bytes/second. These features make the TU77 ideally suited for heavy duty cycle applications such as disk to tape backup and transaction processing.

The TS11 is a medium performance UNIBUS magnetic tape subsystem containing 9 tracks with a density of 1600 bpi. It performs read/writes at a speed of 45 inches/second. The TS11 subsystem can handle a maximum data transfer rate of up to 72K bytes/second.

The operating system's magnetic tape device driver supports the read reverse operation, which enables a program to request a sequential read of the block preceding the block at which the tape is positioned. Writing occurs only while the tape is moving forward.

The operating system's file system can read and write file-structured magnetic tape volumes using the current ANS magnetic tape standard. The system also supports multivolume files, program-controlled blocking factors, and unlabeled magnetic tapes.

UNIT RECORD PERIPHERALS

The operating system normally treats line printers and card readers as spooled shareable devices managed by multiple operator-controlled queues. The devices can also be allocated to individual programs.

The operating system's line printer handling includes line and page counting for job accounting. The user can specify carriage control as: one line per record, FORTRAN conventions, contained within the record itself, or general pre- and post-spacing (within the limits of the hardware capabilities).

The operating system's card reader driver interprets the encoded punched information using the American National Standard 8-bit card code. The driver uses a special punch outside the data representation to indicate end-of-file.

LP11 Line Printers

LP11 series line printers can be connected to the VAX system. The LP11 series printers are impact-type, rotating drum, serial interface line printers. They feature full line buffering, a static eliminator, and a self-test capability.

All models are 132-column printers that can accept paper 4 to 16-3/4 inches wide with up to 6-part forms. They print 10 characters per inch horizontally, and 6 or 8 lines per inch vertically (switch selectable). They include a vernier adjustment for horizontal and vertical paper position. All models are available with either upper (64) or upper/lower (95) character sets (including numbers and symbols). Most models have optional scientific or EDP character sets.

The low-cost models print one line every two revolutions (300 lines per minute with the 64-character set, 230 lines per minute with the 95-character set), or one line every revolution (600 lines per minute with the 64-character set, 460 lines per minute with the 95-character set). A higher-speed version that includes a noise-reduction cabinet, ribbon guide, and a high-speed paper puller offers 900 line-per-minute printing with the 64-character set, or 660 lines per minute with the 95-character set.

For systems requiring even greater printer throughput, LP11 models are available that print up to 1200 lines per minute with the 64-character set or 800 lines per minute with the 95-character set.

LA11 Line Printer

The LA11 is an extremely low-cost, highly-reliable parallel interface printer. The LA11 prints at speeds up to 180 characters per second. The print set consists of the ASCII characters, including 95 upper and lower case letters, numbers, and symbols. Characters are printed using a 7 × 7 matrix with horizontal spacing of 10 characters per inch and vertical spacing of six lines per inch.

Adjustable pin-feed tractors allow for a variable-form width of 3 to 14-7/8 inches (up to 132 columns). A forms length switch sets the top-of-form to any of 11 common lengths, with fine adjustment for accurate forms placement. The printer can accommodate multipart forms (with or without carbons) of up to six parts.

CR11 Card Reader

CR11 card readers can be connected to the system as programmed interrupt request devices. The CR11 reads up to 285 80-column punched cards per minute. The card reader has a high tolerance for cards that have been nicked, warped, bent, or subjected to high humidity. The card reader uses a short card path, with only one card in the track at a time. It uses a vacuum pick mechanism and keeps cards from sticking together by blowing a stream of air through the bottom half-inch of cards in the input hopper. The input hopper holds up to 400 cards, and cards can be loaded and unloaded while the reader is operating.

TERMINALS AND INTERFACES

Interactive terminals can be connected to the VAX system. The operating system's terminal driver provides full duplex handling for both hard copy and video terminals.

Programs can control terminal operations through the terminal driver. The terminal driver supports many special operating modes for terminal lines. A program can enable or disable the following modes by calling a system service:

- SLAVE All unsolicited data are discarded. This mode is used to establish application-controlled terminals.
- NO ECHO Data entered on the terminal keyboard are not printed or displayed on the terminal. This mode is used, for example, to read passwords typed on the terminal.
- PASS ALL All data entered on the terminal are transmitted to the program as 8-bit binary information without any interpretation, except where a line terminator or terminators are specified. This mode enables programs to perform their own interpretation of control characters instead of using the VAX/VMS interpretation.
- ESCAPE Escape sequences entered on the terminal are recognized as read terminators, validated, and passed to a program for interpretation.
- TERMINAL/HOST SYNCHRONIZATION Data sent to the terminal are controlled by terminal-generated XOFF and XON. These functions are generated by typing CTRL/S and CTRL/Q on command terminals and are interpreted as requests to stop and resume output to the terminal.
- HOST/TERMINAL SYNCHRONIZATION All read operations are explicitly solicited with XON and terminated with XOFF. XON and XOFF are also used to keep the type-ahead buffer from filling.

Input from a command terminal is always independent of concurrent output. This capability is called *type-ahead*. Data typed at the terminal are retained in a type-ahead buffer until a program issues a read request. At that time the data are transferred to the program buffer and echoed

on the terminal (provided that echoing is not disabled). If a read is already in progress, the echo and data transfer are immediate. Deferring the echo until a read operation is active allows the program to specify the mode of the terminal, such as No Echo or Convert Lower Case to Upper, to modify the read operation.

A line entered on a command terminal is terminated by any of several special characters, for example, the RETURN key. A program reading from a terminal can optionally specify a particular line terminator or class of line terminators for read requests (including read PASS ALL requests).

Terminal characteristics are initially established during system generation. Users operating command terminals can modify the characteristics of the particular terminal being used. For example, the user can set the baud rate (transmission speed) or change the terminal line width.

LA120 Hard Copy Terminal

The LA120 is a hard copy terminal which offers exceptional throughput and a number of advanced keyboard-selectable formatting and communication features. It uses a contoured typewriter-styled keyboard and includes an additional numeric keypad and a prompting LED display for infrequently used features.

The LA120 achieves high throughput owing to several features:

- 180 character per second print speed
- 14 data transmission speeds ranging up to 9600 baud
- 1K character buffer to equalize differences between transmission speeds and print speeds
- smart and bidirectional printing so that printhead always takes shortest path to next print position
- high speed horizontal and vertical skipping over white space

In addition to its throughput, the LA120 is distinguished by its printing features. The terminal offers eight font sizes, ranging from expanded (5 characters per inch) to compressed font (16.5 characters per inch). Hence a user could, for example, select a font size of 16.5 cpi and print 132 columns onto an 8½-inch-wide sheet. Other print features include six line spacings ranging from 2 to 12 lines per inch, user-selectable form lengths up to 14 inches, left/right and top/bottom margins and horizontal and vertical tabs.

The LA120 is designed for easy use. Terminal characteristics are selected via clearly labeled keys and simple mnemonic commands. Once the selections have been made, the operator can check his settings by depressing the STATUS key. The terminal will then print a listing of the selected settings.

LA36 Hard Copy Terminal

The LA36 is an exceptionally reliable hard copy terminal. It is a lower-priced device than the LA120 with lower throughput (30 cps vs. 165 cps) and fewer print features.

The LA36 uses a typewriter-like keyboard which produces 128 ASCII characters, consisting of 95 upper- and lower-case printing characters and 33 control characters, and is available with optional special character sets, including various foreign language character sets.

Characters are printed using a 7 × 7 matrix with horizontal spacing of ten characters per inch and vertical spacing of six lines per inch. To ensure clear visibility of the printed line, the print head automatically retracts out of the way when not in operation. Adjustable pin-feed tractors allow for a variable-form width from 3 to 14-7/8 inches (up to 132 columns). The print mechanism will accommodate multipart forms (with or without carbons) of up to six parts.

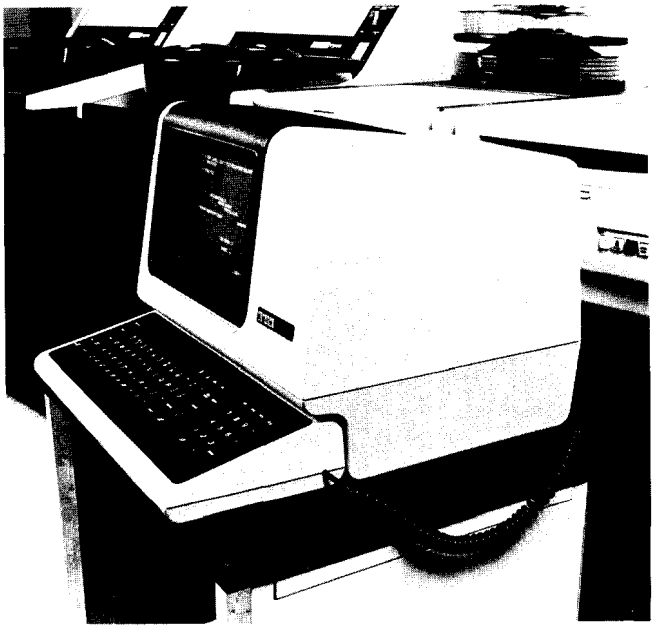
The LA36 operates at speeds of 110, 150, or 300 baud (10, 15, or 30 characters per second). Printable characters are stored in a buffer during the carriage return operation. While more than one character is in the buffer, the printer mechanism operates at an effective speed of 60 characters per second.

VT100 Video Terminal

The VT100 Video Terminal is an upper- and lower-case ASCII terminal which offers a variety of controllable character and screen attributes. The VT100 features a typewriter-like detachable keyboard which includes a standard numeric/function keypad for data entry applications. Also featured are seven LEDs, four of which are program-controlled, used as operator information and diagnostic aids.

The VT100 offers a number of advanced features. The most important of these are:

- ability to select either of two screen sizes: 24 lines by 80 columns or 14 lines by 132 columns
- ability to select either double-width single-height characters or double-width double-height characters on a line by line basis
- smooth scrolling and split screen capability
- ability to set baud rates, tabs, and Answer Back messages from the keyboard and to store these in RAM (Random Access Memory)
- special line drawing graphic characters providing the ability to display simple graphics for business or laboratory applications
- ability to select black-on-white characters or white-on-black characters on a full screen basis



In addition, several options further extend the capabilities of the VT100. These include the advanced video option, which adds selectable blinking, underline, and dual intensity characters to the existing reverse video attribute; the provision of space, power, and interconnects for the future addition of a terminal processor; and additional RAM allowing 24 lines of 132 characters.

DZ11 Terminal Line Interface

The DZ11 is a serial line multiplexer whose character formats and operating speeds are programmable on a per-line basis. A DZ11 connects the UNIBUS with up to a maximum of 8 or 16 asynchronous serial lines, depending on the configuration. Each line can run at any one of 15 speeds.

Local operation with EIA terminals is possible at speeds up to 9600 baud. Remote dial-up terminals can operate full-duplex at speeds up to 300 baud using Bell 103 or 113 modems, or up to 1200 baud using the Bell 212 modem.

The DZ11 optionally generates parity on output and checks parity on input. Incoming characters are buffered using a 64-character silo buffer. Outgoing characters are processed on a programmed interrupt request basis.

REAL-TIME I/O DEVICES

To enhance real-time performance, VAX supports the DR-11B, the LPA11-K direct memory access (DMA) interfaces, and the DR780 32-bit high performance parallel interface (VAX/11-780 only). These devices allow data to be transferred from a peripheral device to memory and vice-versa without the intervention of computer programs except at the initialization and completion of transfers. The result is that CPU involvement in I/O operations is greatly reduced. Further, since these devices are capable of "driving" large blocks of information at high speeds, their usage can greatly increase I/O bandwidth, i.e., the capacity of the system to sustain a total data transfer load. I/O bandwidth is an important performance measure in real-time applications, since such applications require data transfers between external devices and computer memory.

LPA11-K

The LPA11-K is an intelligent (dual-microprocessor) direct memory access controller that buffers real-time data and transfers them to VAX memory in efficient blocks (rather than a word at a time). Since the LPA11-K has automatic buffer switching capability, transfers may occur continuously. Via a system call, the programmer can instruct the LPA11-K to take samples from a data source at specified time intervals. Sampling is handled by the microprocessors, without the intervention of the CPU. Under VMS, the LPA11-K can be accessed via VAX-11 FORTRAN, VAX-11 BASIC, VAX-11 BLISS-32, and MACRO.

The LPA11-K operates in two distinct modes: dedicated mode and multirequest mode.

In multirequest mode, up to eight requests can be active concurrently. Each user's sampling rate is a user-specified multiple of the common real-time clock rate; thus independent rates can be maintained for each user. Each request specifies the device so that A/D, D/A or digital I/O can be synchronously sampled; the transition of a bit in a digital word can synchronize the sampling with a user event. In multirequest mode, throughput is determined by the number and types of requests. The aggregate throughput rate for all users is typically 15,000 samples per second.

In dedicated mode, one user can sample from analog-to-digital converters, or output to a digital-to-analog converter. Two analog-to-digital converters can be controlled simultaneously. Sampling is initiated by an overflow of the real-time clock, or by an external signal. Two sampling algorithms are implemented. One, at each overflow, samples both analog-to-digital converters in parallel, allowing two channels to be sampled simultaneously. The other algorithm samples the two converters on an interleaved basis, beginning with the first whose sampling begins on alternate clock overflows.

The LPA11-K supports the following I/O devices on VAX:

- AA11K (four-channel 12-bit D/A converter)
- AD11K (eight-channel 12-bit A/D converter)
- AM11K (multiplexer board)
- DR11K (16-bit parallel, general device interface)
- KW11K (real-time clock)

DR11-B

The DR11-B is a general purpose, direct memory access (DMA) digital interface to the UNIBUS. The DR11-B, rather than using programmed controlled data transfers, operates directly to or from memory, moving data between the UNIBUS and the user device. The DR11-B, like the LPA11-K, is a block transfer device. However, it is less expensive than the LPA11-K, does not include a microprocessor, and can only handle a single task for a single programmer.

The DR11-B interface consists of four registers: command and status, word count, bus address, and data. Operation is initialized under program control by loading word count with the 2's complement of the number of transfers, specifying the initial memory or bus address where the block transfer is to begin and by loading the command/status register with function bits. The user device recognizes these function bits and responds by setting up the control inputs. If the user device requests data from memory of a UNIBUS device, the DR11-B performs a UNIBUS Data In transfer (DATI) and loads its data register with the information held at the referenced bus address. The outputs of this register are available to the user device; this output data is buffered. If the user device requests data to be written into memory, the DR11-B performs a UNIBUS Data Out transfer (DATO), moving data from the user device to the referenced bus address; this input data is not buffered. Transfers normally continue at a user-defined rate until the specified number of words is transferred. The DR11-B has the capability of transferring data at a rate of 500,000 bytes/second, but actual transfer rates depend upon the particular configuration.

DR780

The DR780 is a high performance general purpose interface adaptor that enables users to directly interface custom devices to a VAX-11/780 system or to connect two VAX-11/780 systems. This high performance, general purpose interface provides a 32-bit parallel data path capable of transferring data up to 6.67 megabytes/second.

The architecture of the DR780 uses separate interconnect structures for transfer of control information and data. The control interconnect is an asychronous 8-bit bidirectional path for transferring control information to and from the user device. The 8-bit width of the control interconnect makes it possible to have up to 256 individual registers in the user device. The data interconnect is a synchronous 32-bit bidirectional path synchronized to a single clock (either the internal DR780 clock or a clock provided in the user device). By using the DR780 internal clock, the transfer rate is selectable under program control from .156 to 6.67M bytes/sec.

The DR780 provides the high performance interface to utilize the system bandwidth of the VAX-11/780. However, to achieve DR780 bandwidths over 2.0 Mb/second, it is required that the system include two interleaved memory controllers.

Typical applications of the DR780 are high-speed data collection, CPU to CPU communications, signal processing, and interfacing to graphics and array processors.

INTERPROCESSOR COMMUNICATIONS LINK

VAX permits interprocessor communications via the DMC11 communications link or via the MA780, multiport (shared) memory. MA780 is supported by the VAX-11/780 processor only.

DMC11

The DMC11 communications link is designed for high-performance point-to-point serial interprocessor connection based on the DIGITAL Data Communications Message Protocol (DDCMP). The DMC11 provides local or remote interconnection of two computers over a serial synchronous link. Both computers can include the DMC11 and DECnet software, or both computers can include the DMC11 and implement their own communications software. For remote operations, a DMC11 can also communicate with a different type of synchronous interface provided that the remote system has implemented the DDCMP protocol.

By implementing the DDCMP protocol in its high-speed microprocessor, the DMC11 ensures reliable data transmission and relieves the host processor of the details of protocol operation (including character and message synchronization, header and message formatting, error checking, and retransmission control). The DDCMP protocol detects errors on the channel interconnecting the system using a 16-bit Cyclic Redundancy Check (CRC-16). Errors are corrected by automatic retransmission. Sequence numbers in message headers ensure that messages are delivered in proper order with no omissions or duplications.

The DMC11 supports full- or half-duplex operation. Full-duplex operation offers the highest throughput and is used when the communications facilities permit two-way operation. The DDCMP protocol permits continuous simultaneous transmission of data messages in both directions when buffers are available and there are no errors on the channels.

Where both computers are located in the same facility, the DMC11 permits transmission at speeds of up to 1,000,000 bits per second over coaxial cable up to 6,000 feet long, or speeds of up to 56,000 bits per second over coaxial cable up to 18,000 feet long. The necessary modems for local interconnection are built in. Where the computers are located remotely and connected using common carrier facilities, the DMC11 permits transmission of up to 19,200 bits per second using an EIA interface. A DMC11 can interface to synchronous modems such as the Bell models 208 and 209, or other synchronous modems conforming to the RS232-C standard.

MA780

MA780 multiport memory is a bank of MOS semiconductor memory with error correcting code (ECC) that can be shared by up to four VAX-11/780 systems. Each system can randomly access all of the shared memory in exactly the same way it accesses its local memory.

Each MA780 can be expanded from a minimum of 256K bytes to a maximum of 2M bytes. This storage is in addition to each system's local memory, which can be as large as 8M bytes. Since there can be up to two MA780s connected to a CPU, a VAX-11/780 system can now directly address up to 12M bytes of physical memory.

Extensions to VAX/VMS make access to the shared memory transparent to the programmer. That is, processes can be moved from one CPU to another with transparency to the programs involved.

The MA780 can be thought of as a very fast communication device between VAX-11/780 systems. Specifically, VAX/VMS provides support for interprocessor communications through the sharing of data regions, VMS mailboxes, and common event flags. VAX/VMS also allows code to be shared among CPUs.

The MA780 can be used to configure multiple computer systems for very high throughput. Depending on the application, the CPUs can be arranged in either a parallel or pipeline manner, as described in Figure 5-1 below:

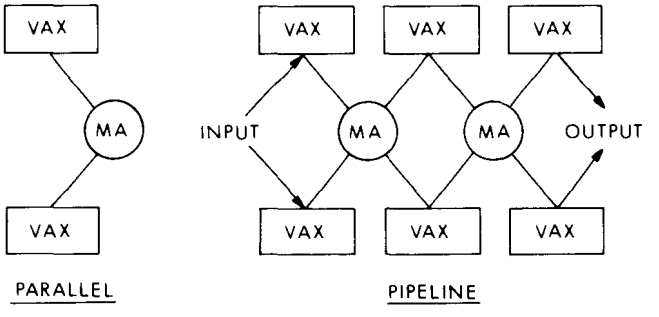


Figure 5-1
Multiported Memory Configurations

In parallel systems, two or more appropriately programmed CPUs can divide a task. This allows the CPUs to effectively pool their power to finish the job quickly. Pipeline systems can increase throughput by allowing instantaneous data exchange between CPUs that are handling sequential parts of an application.

CONSOLE STORAGE DEVICES

The VAX-11/780 console utilizes the RX01 floppy disk

while the VAX-11/750 console utilizes the TU58 magnetic tape cartridge.

RX01 Floppy Disk Cartridge

The RX01 floppy disk is an integral part of the VAX-11/780 console subsystem, storing microdiagnostics and system software. This feature facilitates fast diagnosis (initiated both locally and remotely), simplifies bootstrapping and initialization and improves software update distribution.

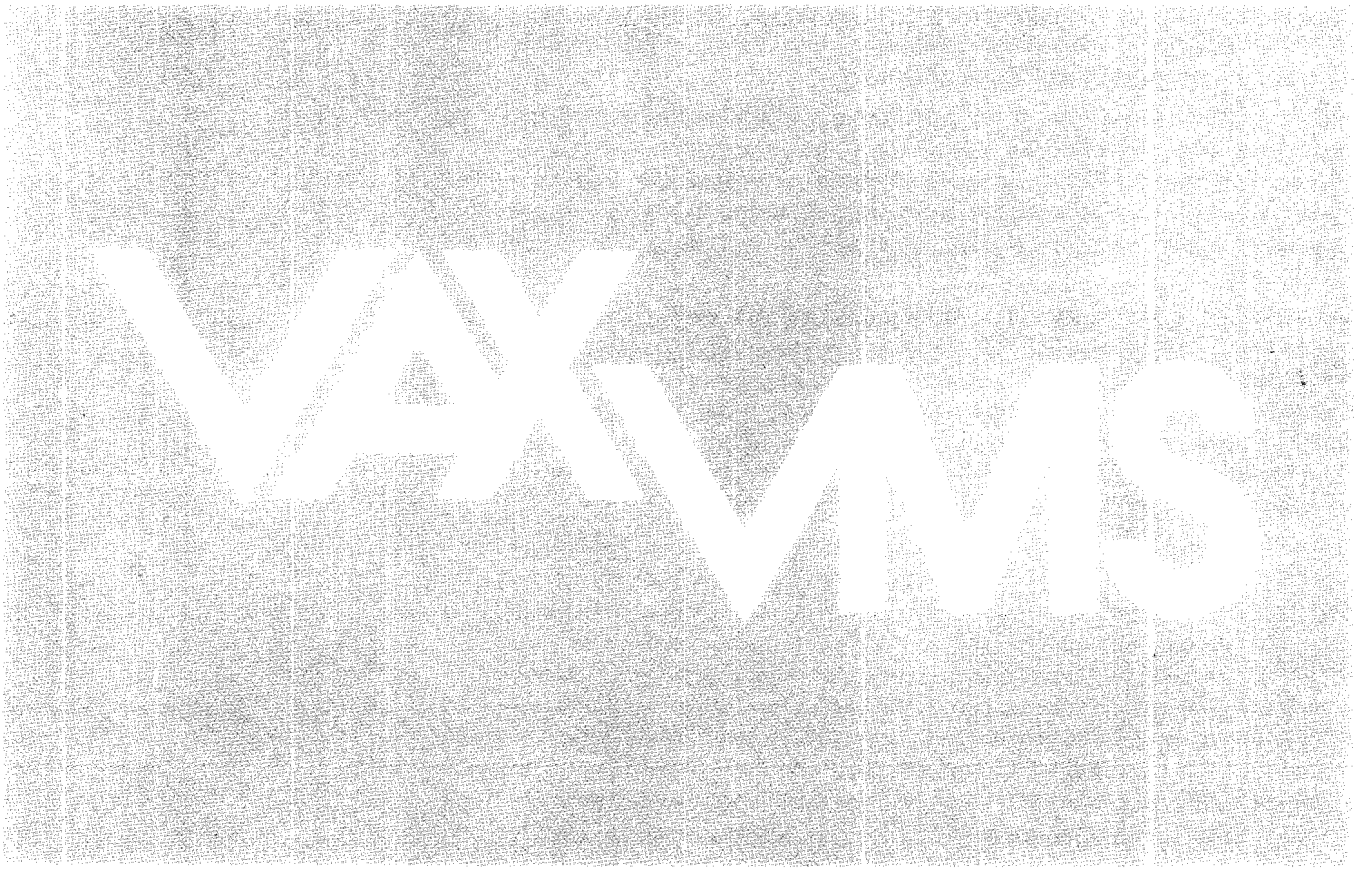
The RX01 is a random access mass memory subsystem that stores data in fixed length blocks on a flexible diskette with preformatted, industry standard headers. The RX01 is a single drive floppy capable of storing 256K bytes of data. The RX02 floppy disk system can also read/write data formatted for the RX01 floppy disk.

TU58 tape cartridge

The TU58 Tape Cartridge Drive is an important part of the VAX-11/750 console subsystem. Because the TU58 is connected directly to the CPU, it maintains the capability to administer diagnostics even with some system components inoperative. This feature significantly increases system reliability. The TU58 may also be used to boot the system, to load files into physical memory, and to store files which describe and execute site-specific bootstrap procedures.

The tape cartridge is preformatted to store 2048 records, each containing 128 bytes. The controller provides random access to any record. The TU58 searches at 60 inches per second (ips) to find the file requested, then reads at 30 ips. Data read from the tape are verified through checksums at the end of each record or header. All data transfers between the TU58 and the host are in 512 byte blocks, with the TU58 concatenating four 128 byte records to accomplish this. Data are transferred to the CPU at approximately 2 KB per second.

6 The Operating System



VAX/VMS is the general purpose operating system for VAX systems. It provides a reliable, high-performance environment for the concurrent execution of multiuser timesharing, batch, and real-time applications. VAX/VMS provides:

- virtual memory management for the execution of large programs
- event-driven priority scheduling
- shared memory, file, and interprocess communication data protection based on ownership and application groups
- programmed system services for process and subprocess control and interprocess communication

VAX/VMS uses memory management features to provide swapping, paging, and protection and sharing of both code and data. Memory is allocated dynamically. Applications can control the amount of physical memory allocated to executing processes, the protection pages, and swapping. These controls can be added after the application is implemented.

VAX/VMS schedules CPU time and memory residency on a pre-emptive priority basis. Thus, real-time processes do not have to compete with lower priority processes for scheduling services. Scheduling rotates among processes of the same priority.

VAX/VMS allows real-time applications to control their virtual memory paging and execution priority. Real-time applications can eliminate services not needed to reduce system overhead. Processes granted the privilege to execute at real-time scheduling levels, however, do not necessarily have the privilege to access protected memory and/or data structures.

VAX/VMS includes system services to control processes and process execution, control real-time response, control scheduling, and obtain information. Process control services allow the creation of subprocesses as well as independent detached processes. Processes can communicate and synchronize using mailboxes, shared areas of memory, shared files or multiple common event flag clusters. A group of processes can also communicate using multiported memory.

Applications designers can use the VAX/VMS protection and privilege mechanisms to implement system security and privacy. VAX/VMS provides memory access protection both between and within processes. Each process has its own independent virtual address space which can be mapped to private pages or shared pages. A process cannot access any other process's private pages. VAX/VMS uses the four processor access modes to read and/or write protect individual pages within a process. Protection of shared pages of memory, files, and interprocess communication facilities,such as mailboxes and event flags, is based on User Identification Codes individually assigned to accessors and data.

INTRODUCTION

VAX/VMS is built for executing high-performance applications where:

- Event-driven interprocess communication and procedure and data sharing are important. Order entry and teller transaction systems often consist of many cooperating processes that synchronize record creation and modification.
- Priorities of resource allocation can be set for currently executing jobs. Both real-time processes and resource-sharing processes can execute in the same environment, as in a communications network. High-speed links can be serviced on demand, while interactive terminal users and batch jobs share processor time and peripherals.
- Large programs can be developed to execute in a physical memory smaller than the program's total memory requirements. Engineering computation programs such as simulators often build data arrays which require a large address space to describe the arrays.

The VAX/VMS operating system provides the run time services for executing high-performance application systems. Operations managers and systems programmers have considerable flexibility in designing and controlling data and program flow.

Applications can be divided into several independent subsystems where data and code are protected from one another, and yet which have general communication and data sharing facilities. Jobs can communicate using general, group, or local communication facilities.

Applications which require an immediate response to some external event can be scheduled as real-time processes. When a real-time process is ready to execute, it executes until it becomes blocked or another higher priority real-time process needs the resources of the processor. Normal jobs can be scheduled using a modified pre-emptive algorithm that ensures that they receive processor and peripheral resources at regular intervals commensurate with their processing needs.

If insufficient memory is available for keeping concurrently executing jobs resident, the operating system will swap jobs in and out of memory to allocate each its share of processor time. Real-time processes can be locked in memory to ensure that they can be started up rapidly when they need to execute.

The operating system provides a dynamic virtual memory programming environment. Large programs can be executed in a portion of physical memory that is considerably smaller than the program's memory requirements, without requiring the programmer to define overlays. The operating system optimizes its virtual memory system for program locality and provides tools that support optimization. It makes program performance predictable and controllable by allowing the programmer to restrict physical memory usage, and by bringing in large amounts of a program at one time. Processes executing under VAX/VMS page against themselves and not against the entire system; thus heavily paging processes executing large programs do not affect the paging of other processes.

The operating system provides sophisticated peripheral device management for sharing, protection, and throughput. Devices can be shared among all jobs or reserved for exclusive use by particular jobs. Input and output for low-speed devices are spooled to high-speed devices to increase throughput. Files on mass storage devices can be protected from unauthorized access on an individual, group, or volume basis.

Furthermore, the I/O request processing system is optimized for throughput and interrupt response. The operating system provides the programmer with several data accessing methods, from logical record accessing for easy, device-independent programming to direct I/O accessing for extremely rapid data processing. Files can be stored in any of several ways to optimize subsequent processing.

VAX/VMS provides the programming tools, scheduling services, and protection mechanisms for multiuser program development. Programmers can write, execute, and debug programs on the system interactively, and also create batch command files that perform repetitive program development operations without requiring their attention.

Although it provides a multiuser program development environment, VAX/VMS is unlike traditional program development timesharing systems. VAX/VMS is an application-oriented operating system that optimizes total system throughput and response to high-priority activities. As in a timesharing system, interactive jobs can be given equal opportunities for resource acquisition. In addition, the system can be executing real-time applications while program development jobs run, since higher priority activities always have the ability to pre-empt lower priority activities.

COMPONENTS AND SERVICES

The operating system is the collection of software that organizes the processor and peripherals into a high-performance system. The operating system's basic components include:

- processes that control initial resource allocation, communicate with the system operator, and log errors
- the command interpreters
- user-callable process control services
- memory management routines
- shared run time library routines
- scheduling routines and swapper
- file and record management services
- interrupt and I/O processing routines
- compatibility mode executive routines
- hardware and software exception dispatching

The operating system's jobs run as independent activities on the system. They include the Job Controller, which initiates and terminates user processes and manages spooling; the Operator Communications Manager, which handles messages queued to the system operators; the Swapper, which controls the swapping of a processes working set in and out of main memory; and the Error Logger, which collects all hardware and software errors detected by the processor and operating system.

A command interpreter executes as a service for interactive and batch jobs. It enables the general user to request

the basic functions that the operating system provides, such as program development, file management, and system information services.

Both hardware-detected and software-detected exception conditions are tracked through the exception dispatcher. The exception dispatcher passes control to user-programmed condition handlers or, in the case of system-wide exception conditions, to operating system condition handlers.

The operating system's memory management routines include the image activator, which controls the mapping of virtual memory to system and user jobs; the pager, which moves portions of a process in and out of memory as required; and various system services, callable by users that want to manage their virtual address space directly. They respond to a program's dynamic memory requirements, and enable programs to control their allocated memory, share data and code, and protect themselves from one another.

The scheduler controls the allocation of processor time to system and user jobs. The scheduler always ensures that the highest priority, ready-to-execute real-time process receives control of the processor until it relinquishes it. When no real-time processes are ready to execute, the scheduler dynamically allocates processor time to all other jobs according to their priorities and resource requirements. The swapper works in conjunction with the scheduler to move entire jobs in and out of memory when memory requirements exceed memory resources. The swapper ensures that the jobs most likely to execute are kept in memory.

The operating system's I/O processing software includes interrupt service routines, device-dependent I/O drivers, device-independent control routines, and user-programmed record processing services. The I/O system ensures rapid interrupt response and processing throughput, and provides programming interfaces for both special purpose and general purpose I/O processing.

The next few sections discuss some of the concepts basic to understanding the operating system's functions and services. They are followed by descriptions of the services available to individual and cooperating processes, and descriptions of memory management and scheduling for the systems programmer.

PROCESSING CONCEPTS

To support high-performance multiprogramming application environments, the operating system provides the applications programmer with the tools to implement:

- shared programs
- shared files and data
- interprocess communication and control

To enable the programmer to write shared programs easily, the operating system treats a program independently of the context in which a program executes. The context defines the privileges assigned by the system manager to a particular user. Users with different privileges can share programs, and the operating system will enforce protection independently of the program.

The operating system controls privilege and accounts for resource allocation by job. A job can be performing processing operations under the direction of one user at a terminal, or it can be performing processing operations for several users at multiple terminals. A job can consist of one or several independently executing processes that share the resource allocations for that job. Jobs can be grouped into application subsystems that share files and communication channels that are protected from other application subsystems.

Programs and Processes

The four concepts important for understanding how the operating system supports multiprogrammed application systems are:

- **image**, or executable program
- **process**, or image context and address space
- **job**, or detached process and its subprocesses
- **group**, or set of jobs that can share resources

These concepts are for the most part transparent to the general user whose only contact with the system is the operating system's command language interpreter or an application's command interface. They are, however, significant concepts for the applications programmer. Figure 6-1 illustrates the concepts of groups, jobs, processes, and images.

An image is an executable program. It is created by translating source language modules into object modules and linking the object modules together. An image is stored in a file on disk. When a user runs an image, the operating system reads the image file into memory to execute the image.

The environment in which an image executes is its context. The complete context of an image not only includes the state of its execution at any one time (known as its hardware context), it also includes the definition of its resource allocation quotas, such as device ownership, file access, and maximum physical memory allocation. These resource allocation quotas are determined by the quotas given to the user who runs the image.

Two or more users can execute the same image concurrently; that is, image code can be shared, in which case the image is executing in two or more different contexts. An image context, including the address space used by an image, is called a **process**. The operating system schedules processes, and a process provides a context in which an image executes.

The distinction between an image and a process is a significant one. We can speak of two processes, each executing the FORTRAN compiler. There may be only one copy in physical memory of the FORTRAN compiler's image, but two different contexts in which the image executes. In one context, the compilation may have just begun; in the other context, it may be almost complete. In one context, the compiler may be reading and writing files listed in one directory; in the other context, the compiler may be reading and writing files listed in another directory.

A process executes only one image at a time, but it provides the context for serially executing any number of different images. For example, when a user logs on the sys-

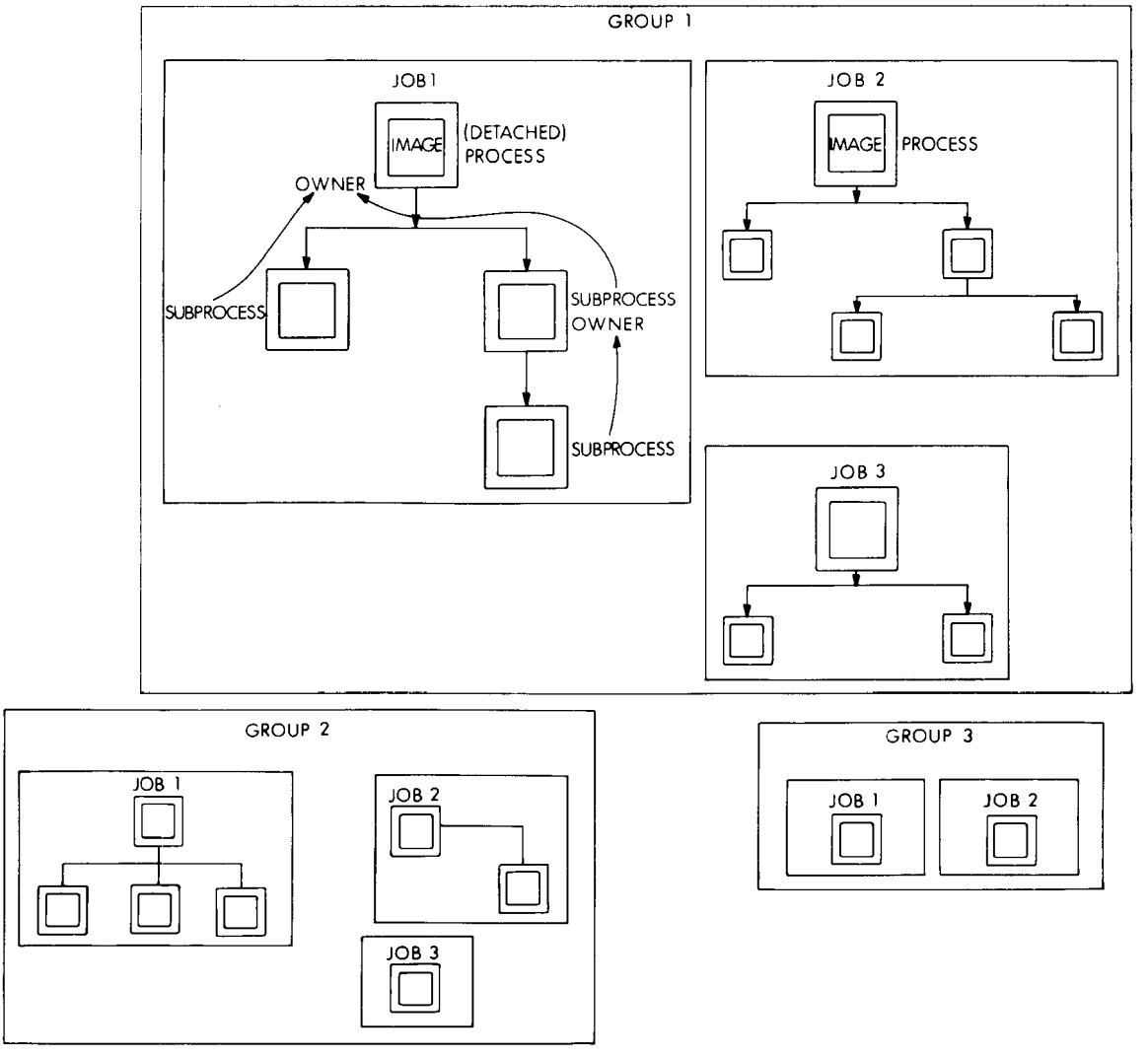


Figure 6-1

Programs and Processes

tem at an interactive terminal, the operating system creates a process for that user. If the user edits a file, the editor image executes in the context of that user's process. If the user then compiles a program, the compiler image executes in the context of that user's process. A process thus acts as a continuous "envelope" for a user's activities.

An image executing in the context of a process can create **subprocesses**. A subprocess can be thought of as an auxiliary process in which a given image executes. When an image creates a subprocess, it identifies the image to be executed in the context of the subprocess or the source of commands to be interpreted in that process. An image executing in a subprocess context can in turn create other subprocesses.

The process executing the image that creates a subprocess is an **owner process**. An owner process has complete control over the execution of the subprocesses it creates. It determines which of its privileges it will allow a subprocess to have. Each detached process and all subprocesses created below it heirarchically share a common pool of quotas. The owner process can control the scheduling of its subprocess, and it can delete the subprocess. When an owner process terminates, all of the subprocesses it owns are terminated.

A **detached process** is the process created by the operating system on behalf of a user who logs onto the system and requests services of the system using a command interpreter. A detached process has no owner. Normally, only the operating system can create detached processes, but a suitably privileged application program could also create a detached process and start up an application command interface to execute images serially for the detached process or any subprocess it creates.

A **job** consists of a detached process and all the subprocesses it creates, and all the subprocesses they create, etc. Jobs are the accounting entities that the system uses to control resource allocation. All processes constituting a job are scheduled independently (they can compete for

processor time individually to overlap processing), but they share the total resources allocated to the detached process. Each job has a set of resources that it can use, i.e., authorized quotas. Subprocesses share these quotas with the detached process.

Jobs can be associated in **groups**. Groups are the basis for the definition and development of application subsystems. Groups are mutually exclusive, that is, if a job belongs to one group, it does not belong to any other group. A process with appropriate privilege can control the execution of other processes in the same group. Processes in the same group can synchronize their activities using protected group communication facilities.

Resource Allocation

The resources of the system are the processor, memory, and peripherals. The system handles many jobs simultaneously, and each job can have different resource requirements. The operating system enables jobs to share the resources according to their individual needs, and the operating system protects each job and its data from other jobs on the system.

The operating system controls resource allocation dynamically through its scheduling, memory management, device allocation, and I/O processing software, and statically through the authorization of users.

The system manager is responsible for creating an authorization file entry for each user of the system. The authorization file provides the operating system with the resource quotas and limits for each job. For example, there are quotas and limits that control:

- total processor time usage
- number of subprocesses a job can create
- number of simultaneously open files
- process virtual and physical memory usage
- number of simultaneous I/O transfers

Separate authorization files located on each disk volume control disk usage quotas.

Privileges

In addition to providing job quotas, the user authorization file provides the base definition for each user's privileges. There are potentially 64 distinct privileges that can be individually granted or withheld. Among them are privileges that give the job the right to:

- alter the priority of a process
- execute a user-written program at a more privileged access mode
- execute operator functions
- create detached processes
- set up the communication facilities used by cooperating processes
- control other processes in the same group

Whenever the user executes an image, the image can at most acquire only those privileges and quotas granted directly to that user's job by the authorization file, unless the image is a **known image**. Known images are installed by the system manager, and while they execute they provide a second, dynamic set of privileges granted a user.

When the user executes a known image, the process has the privileges and quotas granted to the user in the authorization file, *plus* those run time execution privileges granted specifically to that image. While that image executes, the user may have the privilege to perform operations not granted when executing any other image. For example, one known image is the operating system's LOGIN image, which enables a user to log on the system. The LOGIN image has the privilege required to access the user authorization file to obtain the user's privileges and quotas.

Protection

The basis for data protection in the VAX system is the user identification code (UIC). A UIC consists of two numbers: a group number and a member number. The system manager assigns each user a user identification code (UIC) in the user authorization file. Images that the user executes are given or denied data access privileges based upon the user's UIC.

When a file or an interprocess communication facility is created, it is assigned a UIC and a protection code. The UIC determines which group of users or programs, and which members within the group, have controlled access to that data. The protection code provides the access control.

The protection code applies to four types of access: **read, write, execute,** and **delete**. Each type of access can be given or denied to:

- the owner: the user whose UIC is the same as the UIC assigned to the data.
- the group: every user whose UIC group number is the same as that assigned the data.
- the world: every user whose UIC group number is different or the same from that assigned the data. (everyone on the system)

- the system: every user or program with the privilege SYSPRV, and those whose UIC group number is a system privileged group number (1-X, where X is a number specified by the sysgen parameter, MAXSYSGROUP).

For example, in a common application of the protection scheme, a user can create a program image file and assign it the same UIC as the user's own UIC (the default case). The user can give it a protection code to:

- enable the user (and all other users with the same UIC) to read, write, execute, and delete the file
- enable other users in the group to execute the program image, but prevent them from reading, writing or deleting the file
- prevent all users outside the group (other than privileged system users) from reading, writing, executing, or deleting the file
- enable the the privileged system users to read the file (so that it can be backed up, for example)

Read and write access applies to both files and interprocess communication facilities. Delete access applies only to files, and execute access applies only to program image files. (The privileges and quotas granted in the user authorization file control creation and deletion for interprocess communication facilities.)

USER PROCESS ENVIRONMENT

The user program environment is the process, which is the entity the operating system schedules for execution. Each process has its own independent address space in which an image executes. Each image executing in a process can call system service procedures to acquire resources and request special processing services from the operating system. The following paragraphs introduce program virtual address allocation and the fundamental system service procedures available to user programs directly, as well as indirectly through the more complex programmed requests provided by the operating system.

Virtual Address Space Allocation

Process virtual address space is the set of 32-bit addresses that an image executing in the context of a process uses to identify byte locations in virtual memory. For the purpose of allocating virtual memory to processes, the operating system divides process virtual address space into four sets of virtual addresses. The first three sets of addresses are called the program region, the control region, and the system region. The fourth set of addresses is unused.

Figure 6-2 illustrates the general allocation of virtual address space for each process. Addresses in the first two regions are used for process code and data, where the first region is generally used for image-specific code and data, and the second for stacks, process permanent data, buffers, and operating system code. Addresses in the system region are also used for code and data maintained by the operating system, but in this case the addresses refer to the same locations in every process context. The system region addresses provide a set of locations whose addresses are independent of process context, and therefore do not have to be context switched.

When a user program is translated and linked, the image is allocated addresses starting with address 512 and continuing up. The first page is not normally allocated (although it can be) because it helps catch programming errors caused by improperly initialized pointers, by branching or jumping to 0, or by passing 0 or other small addresses as arguments. The linker allocates the remainder of address space to image sections according to whether they are shared or private, position-independent or position-dependent, and read-only or read/write, such that memory protection can be used to full advantage in preventing and isolating programming errors.

The addresses in the control region are used to identify the locations containing temporary image control information and data such as the stacks, permanent process control information such as I/O channel allocations, and code provided by the operating system. These addresses are allocated from address $2^{31}-1$ *down*. One reason this method of allocating in reverse is convenient is because the control region contains the process stacks, and stacks grow to lower addresses as data are added, and to higher addresses as data are removed.

There are four stack areas reserved in the control region, one for each access mode protection level that the processor provides for software executing in the context of a process. (Refer to the VAX Processors section for a description of access modes.) The stack seen by the image executing in the program region is the user stack. All other stack areas are protected from that image. These stacks are used by operating system software executing in the context of the process on behalf of the image the process is executing. For example, command interpreters use the supervisor stack, the record management services use the executive stack, and the exception dispatcher and some exception handlers use the kernel stack.

The system region addresses, which start at address 2^{31}, are used to identify the locations containing the entry vectors for system service procedures, followed by locations containing privileged operating system code and data. The system service entry vectors are permanently reserved virtual addresses so that no relinking is required if system services are modified. Other addresses in the system region are not generally used by the image allocated to the program region, and access to areas mapped by these addresses is restricted.

System Services

An image requests services of the operating system directly through calls to the **system services**. The system services are to the operating system what the instruction set is to the processor. They provide all the primary resource request activities, such as I/O processing and interprocess communication. Other programmed requests available to the user are often derived from system services. For example, the record management services use the I/O processing system services as the basis for logical record processing functions.

Images that use the system services can be written in assembly language or any native programming language that has a Call statement. (Refer to the Languages section and the section on the RSX-11M Programming Environment for system services available for compatibility mode

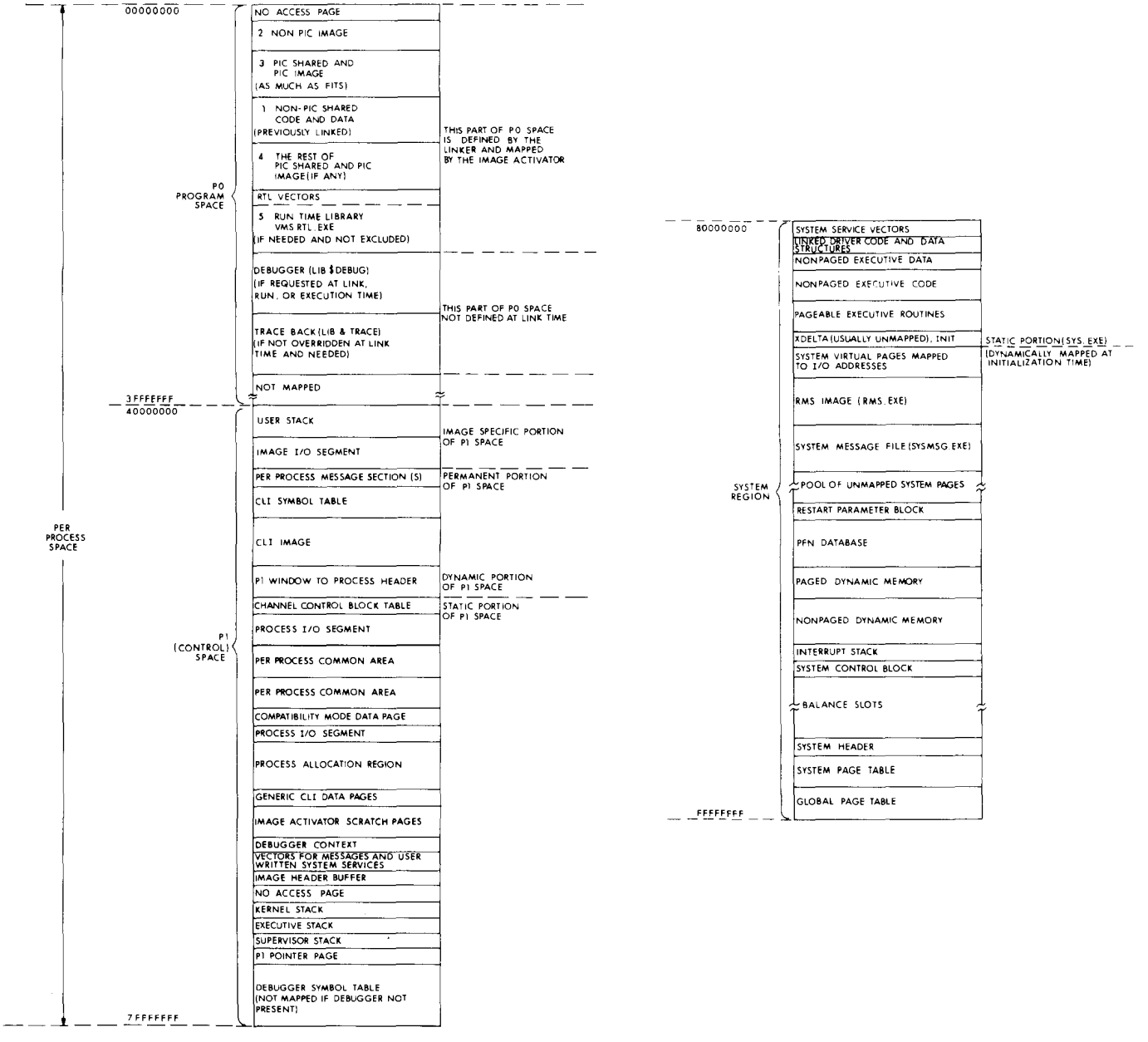


Figure 6-2

Virtual Address Space Allocation

programming languages.) The Call interface is the same independent of the programming language selected with the exception of CORAL 66.

Table 6-1 summarizes the system services available to the applications programmer, some of which are controlled by privilege. The table also lists those system services used primarily by the operating system but which can also be used by suitably privileged application system code.

Table 6-1

System Services

I/O SERVICES FOR DEVICE-DEPENDENT I/O	
Assign I/O Channel ($ASSIGN)	Establish a path for an I/O request. Establish a path for network operations.
Deassign I/O Channel ($DASSGN)	Release linkage for an I/O path. Release a path from the network.
Get I/O Channel Information ($GETCHN)	Provide information about a device to which an I/O channel has been assigned.
Get I/O Device Information ($GETDEV)	Provide information about a physical device.
Allocate Device ($ALLOC)	Reserve a device for exclusive use by a process and its subprocesses.
Deallocate Device ($DALLOC)	Relinquish exclusive use of a device.
Queue I/O Request ($QIO)	Initiate an input or output operation and continue processing while I/O is in progress.
Queue I/O Request and Wait ($QIOW)	Initiate an input or output operation and cause the process to wait until the I/O is complete before continuing execution.
Cancel I/O on Channel ($CANCEL)	Cancel pending I/O requests on a channel.
Formatted ASCII Output ($FAO)	Perform ASCII string substitution, and convert numeric data to ASCII representation and substitute in output.
Formatted ASCII Output with List Parameter ($FAOL)	
I/O SERVICES FOR MAILBOXES AND MESSAGES	
Create Mailbox and Assign Channel ($CREMBX)	Create a temporary mailbox. Create a permanent mailbox.
Delete Mailbox ($DELMBX)	Mark a permanent mailbox for deletion.
Broadcast ($BRDCST)	Send a high-priority message to a specified terminal or terminals.
Send Message to Accounting Manager ($SNDACC)	Control accounting log file activity. Write an arbitrary message to the accounting log file.
Send Message to Symbiont Manager ($SNDSMB)	Request symbiont manager to initialize, modify, or delete a printer, device, or batch job queue. Request symbiont manager to queue a batch or print file, or delete or change characteristics of a queued file.
Send Message to Operator ($SNDOPR)	Write a message to designated operator(s) terminal(s). Enable or disable an operater's terminal, send a reply to a user request or initialize the operator's log file.
Send Message to Error Logger ($SNDERR)	Write arbitrary data to the system error log file.
Get Message ($GETMSG)	Return text of system or application error message from message file.
Put Message ($PUTMSG)	Write a system or application error message to the current output and error devices.
LOGICAL NAME SERVICES	
Create Logical Name ($CRELOG)	Place logical name/equivalence name pair in process logical name table. Place logical name/equivalence name pair in group logical name table. Place logical name/equivalence name pair in system logical name table.
Delete Logical Name ($DELLOG)	Remove logical name/equivalence name pair from process logical name table. Remove logical name/equivalence name pair from group logical name table. Remove logical name/equivalence name pair from system logical name table.
Translate Logical Name ($TRNLOG)	Search logical name table for a specified logical name and return its equivalence name when the first match is found.

Table 6-1 (con't)
System Services

EVENT FLAG PROCESSING

Associate Common Event Flag Cluster ($ASCEFC)	Create a temporary common event flag cluster. Create a permanent common event flag cluster. Create a common event flag cluster in memory shared by multiple processors. Establish association with an existing common event flag cluster.
Disassociate Common Event Flag Cluster ($DACEFC)	Cancel association with a common event flag cluster.
Delete Common Event Flag Cluster ($DLCEFC)	Mark a permanent common event flag cluster for deletion.
Set Event Flag ($SETEF)	Turn on an event flag in a process-local event flag cluster. Turn on an event flag in a common event flag cluster.
Clear Event Flag ($CLREF)	Turn off an event flag in a process-local event flag cluster. Turn off an event flag in a common event flag cluster.
Read Event Flags ($READEF)	Return the status of all event flags in a process-local event flag cluster. Return the status of all event flags in a common event flag cluster.
Wait for Single Event Flag ($WAITFR)	Place the current process in a wait state pending the setting of an event flag in a process-local event flag cluster. Place the current process in a wait state pending the setting of an event flag in a common event flag cluster.
Wait for Logical OR of Event Flags ($WFLOR)	Place the current process in wait state pending the setting of any one of a specified set of flags in a process-local event flag cluster. Place the current process in a wait state pending the setting of any one of a specified set of flags in a common event flag cluster.
Wait for Logical AND of Event Flags ($WFLAND)	Place the current process in a wait state pending the setting of all specified flags in a process-local event flag cluster. Place the current process in a wait state pending the setting of all specified flags in a common event flag cluster.

AST (ASYNCHRONOUS SYSTEM TRAP) SERVICES

Set Power Recovery AST ($SETPRA)	Establish AST routine to receive control following power recovery condition.
Set AST Enable ($SETAST)	Enable or disable the delivery of ASTs.
Declare AST ($DCLAST)	Queue an AST for delivery.

CONDITION HANDLING SERVICES

Set Exception Vector ($SETEXV)	Define condition handler to receive control in case of hardware- or software-detected exception conditions.
Set System Service Failure Exception Mode ($SETSFM)	Request or disable generation of a software exception condition when a system service call returns an error or severe error.
Unwind from Condition Handler Frame ($UNWIND)	Delete a specified number of call frames from the call stack following a nonrecoverable exception condition.
Declare Change Mode or Compatibility Mode Handler ($DCLCMH)	Designate a routine to receive control when change mode to user instructions are encountered. Designate a routine to receive control when change mode to supervisor instructions are encountered. Designate a routine to receive control when compatibility mode exceptions occur.

PROCESS CONTROL SERVICES

Create Process ($CREPRC)	Create a subprocess. Create a detached process.
Set Process Name ($SETPRN)	Establish a text string to be used to identify the current process.
Get Job/Process Information ($GETJPI)	Return information about the current process. Return information about the current process context of other processes in the same group. Return information about any other process in the system.
Delete Process ($DELPRC)	Delete the current process, or a subprocess. Delete another process in the same group. Delete any process in the system.
Hibernate ($HIBER)	Make the current process dormant but able to receive ASTs until a subsequent wakeup request.
Schedule Wakeup ($SCHDWK)	Wake a process after a specified time interval or at a specific time.

Table 6-1 (con't)
System Services

Cancel Wakeup ($CANWAK)	Cancel a scheduled wakeup request.
Wake ($WAKE)	Restore executability of the current process or a hibernating subprocess.
	Restore executability of a hibernating process in the same group.
	Restore executability of any hibernating process in the system.
Suspend Process ($SUSPND)	Make the current process or a subprocess nonexecutable and unable to receive ASTs until a subsequent resume or delete request.
	Make another process in the same group nonexecutable and unable to receive ASTs until a subsequent resume or delete request.
	Make any process in the system nonexecutable and noninterruptible until a subsequent resume or delete request.
Resume Process ($RESUME)	Restore executability of a suspended subprocess.
	Restore executability of a suspended process in the same group.
	Restore executability of any suspended process in the system.
Exit ($EXIT)	Terminate execution of an image and returns to command interpreter.
Force Exit ($FORCEX)	Cause image exit for the current process or a subprocess.
	Cause image exit for a process in the same group.
	Cause image exit for any process in the system.
Declare Exit Handler ($DCLEXH)	Designate a routine to receive control when an image exits.
Cancel Exit Handler ($CANEXH)	Cancel a previously established exit handling routine.
Set Priority ($SETPRI)	Change the execution priority for the current process or a subprocess.
	Change the execution priority for a process in the same group.
	Change the execution priority for any process in the system.
Set Resource Wait Mode ($SETRWM)	Request wait, or that control be returned immediately, when a system service call cannot be executed because a system resource is not available.
Set Privileges ($SETPRV)	Allow a process to enable or disable specified user privileges.

TIMER AND TIME CONVERSION SERVICES

Get Time ($GETTIM)	Return the date and time in system format.
Convert Binary Time to Numeric Time ($NUMTIM)	Convert a date and time from system format to numeric integer values.
Convert Binary Time to ASCII String ($ASCTIM)	Convert a date and time from system format to an ASCII string.
Convert ASCII String to Binary Time ($BINTIM)	Convert a date and time in an ASCII string to the system date and time format.
Set Timer ($SETIMR)	Request setting of an event flag or queuing of an AST, based on an absolute or delta time value.
Cancel Timer Request ($CANTIM)	Cancel previously issued timer requests.
Schedule Wakeup ($SCHDWK)	Schedule a wakeup for the current process or a hibernating subprocess.
	Schedule a wakeup for a hibernating process in the same group.
	Schedule a wakeup for any hibernating process in the system.
Cancel Wakeup ($CANWAK)	Cancel a scheduled wakeup request for the current process or a hibernating subprocess.
	Cancel a scheduled wakeup request for a hibernating process in the same group.
	Cancel a scheduled wakeup request for any hibernating process in the system.
Set System Time ($SETIME)	Set or recalibrate the current system time.

MEMORY MANAGEMENT SERVICES

Adjust Working Set Limit ($ADJWSL)	Change maximum number of pages that the current process can have in its working set.
Expand Program/Control Region ($EXPREG)	Add pages at the end of the program or control region.
Contract Program/Control Region ($CNTREG)	Delete pages from the end of the program or control region.
Create Virtual Address Space ($CRETVA)	Add pages to the virtual address space available to an image.
Delete Virtual Address Space ($DELTVA)	Make a range of virtual addresses unavailable to an image.

Table 6-1 (con't)
System Services

Create and Map Section ($CRMPSC)	Identify a disk file as a private section and establish correspondence between virtual blocks in the file and the process' virtual address space.
	Identify a disk file containing shareable code or data as a temporary global section and establish correspondence between virtual blocks in the file and the process' virtual address space.
	Identify a disk file containing shareable code or data as a permanent global section and establish correspondence between virtual blocks in the file and the process' virtual address space.
	Identify a disk file containing shareable code or data as a system global section and establish correspondence between virtual blocks in the file and the process' virtual address space.
	Identify one or more page frames in physical memory as a private or global section and establish correspondence between the page frames and the process' virtual address space.
Update Section File on Disk ($UPDSEC)	Write modified pages of a private or global section into the section file.
Map Global Section ($MGBLSC)	Establish correspondence between a global section and a process' virtual address space.
Delete Global Section ($DGBLSC)	Mark a permanent global section for deletion.
	Mark a system global section for deletion.
Set Protection on Pages ($SETPRT)	Control access to a range of virtual addresses.
Lock Pages in Working Set ($LKWSET)	Specify that particular page cannot be paged out of the process' working set.
Unlock Pages from Working Set ($ULWSET)	Allow previously locked pages to be paged out of the working set.
Purge Working Set ($PURGWS)	Remove all pages within a specified range from the current working set.
Lock Page in Memory ($LCKPAG)	Specify that particular pages may not be swapped out of memory.
Unlock Page in Memory ($ULKPAG)	Allow previously locked pages to be swapped out of memory.
Set Process Swap Mode ($SETSWM)	Control whether or not the current process can be swapped out of the balance set.
CHANGE MODE SERVICES	
Change Mode to Executive ($CMEXEC)	Execute a specified routine in executive mode.
Change Mode to Kernel ($CMKRNL)	Execute a specified routine in kernel mode.
Adjust Outer Mode Stack Pointer ($ADJSTK)	Modify the current stack pointer for a less privileged access mode.

I/O System Services

The operating system provides the programmer with two request interfaces for performing input/output operations: the I/O system services and the record management services. Record management services, discussed in the Data Management Facilities section, provides a general purpose file and record programming interface that satisfies most I/O processing needs, and allows the programmer to implement I/O processing quickly. The I/O system services provide the programmer with direct control over the I/O processing resources of the operating system. In particular, the I/O system services enable the programmer to:

- perform both device-independent and device-dependent I/O processing
- read and write blocks on mass storage media using physical (device-oriented), logical (volume-relative), or virtual (file-relative) addressing

The I/O system recognizes several types of devices, and within the extents of their capabilities, all devices are programmed in the same manner. All devices can be sequentially accessed, including mass storage devices such as disks and magnetic tapes, and record-oriented devices, such as terminals, card readers and line printers. In addition, disk volumes can be accessed randomly.

Mass storage volumes can be either file-structured or non-file-structured, according to the choice of the user. The I/O system services enable programmers to use either the physical (device assigned) address or a logical (driver assigned) address for directly addressing blocks on **foreign** mass storage volumes. A foreign volume can be either non-file-structured, or structured with the user's own file structure. If the volume is structured using the operating system's Files-11 disk file or ANSI magnetic tape structure, the I/O system services enable the programmer to address blocks directly using virtual (file system assigned) addresses.

A special type of record-oriented device is the **mailbox**, which is a virtual device that a process creates for the receipt of messages from other processes. Mailboxes are

treated like any other record-oriented device: they can be read from and written to using either the I/O system services or record management services. Mailboxes are discussed further in the section on Interprocess Communication.

Before a process requests I/O to a device, it obtains a channel assignment from the operating system. A process can use a device name or a **logical name** in a channel assignment request to identify the device for which the channel is desired.

A device name is a unique name assigned by the operating system to a particular physical device. The name identifies the type of device and its controller and line or unit number, as applicable. For example, DMA3: is the operating system's device name for the RK06 disk drive unit 3 on controller A, and TTA12: is the operating system's name for the terminal on line 12 on multiplexer A.

A logical device name is any string of characters a user or program assigns to a device name assigned by the operating system. The Create Logical Name system service not only enables a process to define logical names for device names, but it enables a process to assign logical names to any portion of a file specification, or to other logical names. Furthermore, logical names can be assigned on a per-process, per-group, or system-wide basis. (For more information on logical names, refer to the Data Management Facilities Section.)

Once a channel is obtained, a process can issue I/O requests on that channel. The Queue I/O Request system service is a general I/O request interface. All I/O using system services is asynchronous: both I/O and computation can be taking place simultaneously. An I/O request is simply queued to the device driver and control is normally returned to the requesting process before the I/O operation is complete.

The process is responsible for synchronizing with I/O completion. The process can simulate synchronous I/O processing by using the Queue I/O Request and Wait system service, or it can continue to execute during the I/O operation and request I/O completion notification using the general purpose event flag or asynchronous system trap notification mechanisms.

Real-time interface extensions (connect-to-interrupt and map-to-I/O page) provide the real-time programmer (MACRO and BLISS-32) a technique of more simply interfacing to user-specialized devices. The connect-to-interrupt facility can be used to cause an interrupt to be delivered directly to the user's program. As a consequence of this approach, response to the interrupt occurs in the shortest time possible without writing an I/O driver.

The map-to-I/O page is a complement to the connect-to-interrupt facility by allowing the user program to access the device registers. Before these extensions were available, the user had to write both a device driver and an application program to achieve the same results.

Local Event Flags

An event flag is a status bit used for posting an event, such as I/O completion or elapsed time interval. Event flags are an extremely efficient means of starting up or synchronizing procedures.

Each process has available for its own use two local event flag clusters, each of which contains 32 event flags. Eight flags in the first cluster are reserved by the operating system. A process can set, clear, and read individual event flags, as well as wait for one or more event flags to be set. The advantage to having two clusters of event flags is that the flags in each cluster can be treated as a related group. A process can wait until any of a specified set of flags in a particular cluster is set, or wait until all of a specified set of flags in a particular cluster are set.

Aside from their use with I/O processing and timer scheduling, a process can assign its own meanings to local event flags. Event flags can be used to coordinate several asynchronous events, such as multiple I/O request completions, or to simplify asynchronous processing. For example, a program may wish to know if a terminal user has typed a CTRL/C (indicating the desire to interrupt execution) only at well-defined points during processing. An asynchronous system trap routine can set an event flag to indicate that a CTRL/C has been received.

Asynchronous System Traps

An asynchronous system trap (AST) is a software-simulated interrupt used for event notification within a process. An asynchronous system trap routine is a procedure that handles an AST. AST routines provide an efficient means for processing events that can occur at any time during processing (such as terminal input) because they eliminate the need for polling.

For example, a program can specify AST routines for I/O request processing, timer scheduling, and power recovery. When the I/O operation completes, time interval expires, or power is restored, the operating system declares an AST. When the AST is delivered, the operating system interrupts the process and executes the AST routine. A process can be hibernating and still receive ASTs declared for it.

Code executing at one processor access mode can declare an AST for code executing at the same or a less privileged access mode. The operating system automatically disables AST delivery while an AST routine is executing, and code executing at a given access mode can explicitly disable AST delivery. While ASTs are disabled, the operating system queues any ASTs waiting to be delivered to that access mode in the order in which they were declared. When AST delivery is again enabled for that acess mode, the ASTs are delivered in the order in which they were queued.

Exception Conditions and Condition Handlers

A program may request the processor or a system service to do operations they cannot perform correctly. For example, a program might inadvertently issue a divide instruction using a divisor of zero. Normally there is no way to recover and the program cannot continue. In this system, however, it is possible for a program to continue if it declares a **condition handler** that can correct the situation. If a user program declares a condition handler, control transfers to the condition handler when an **exception condition** occurs.

This system treats all errors or special events that occur synchronously with respect to a program's execution as

exception conditions, and provides a general purpose mechanism for dispatching condition handlers. Exception conditions include:

- errors from which the processor cannot normally recover, such as the divide by zero arithmetic trap
- special conditions for which a program does not wish to test continually, for example, the floating point overflow arithmetic trap or unsuccessful system service completion

Some of the exceptions detected by the processor are handled automatically by the operating system. For example, the pager is a condition handler for translation-buffer-not-valid faults.

In addition to processor and system service detected exception conditions, any software procedure can define cases for which it will fail or produce an exception by calling a system library procedure that signals an exception condition. The search sequence for a condition handler is independent of the nature of the exception condition: the search sequence is the same whether an exception condition is detected by hardware or software.

A process can declare two kinds of condition handlers: those that are process-wide and those that are applicable to individual procedures. Process-wide condition handlers are declared using the Set Exception Vector system service, which enables a process to declare a primary and a secondary condition handler. Condition handlers applicable to individual procedures are declared by the procedure when it is called using one instruction.

When an exception condition occurs, the exception dispatcher does not differentiate between exception conditions, it simply transfers control to the first condition handler it can find that wants to handle the exception condition. This method for handling exception conditions is an efficient means of transferring control to the appropriate condition handler rapidly, since condition handling is defined by the module or modules in which an exception condition may occur.

For programs written in high-level languages, each language may have different definitions of what is and what is not an exception condition. As the user program calls language functions, the exception conditions for those functions can be handled locally with the procedure. And where exception conditions should be handled on a process-wide basis, the primary and secondary exception vectors provide a top level exception condition trap. For example, when a user program is linked with the debugger, the debugger uses the primary exception vector to declare a process-wide condition handler.

INTERPROCESS COMMUNICATION AND CONTROL

This system supports both simple and complex job definitions. A simple job is a detached process created by the operating system on behalf of the user who logs in at a terminal, or for the purpose of executing a batch job. A simple job serially executes images, but it does not create subprocesses.

A complex job is one in which a detached process creates subprocesses in which designated images execute. These subprocesses can also create their own subprocesses, and so on. The advantage of a complex job over a simple job is that a complex job performs parallel processing operations because it has control over several images executing concurrently.

The following sections describe the services that enable a process to control and communicate with other processes.

Process Control Services

The system services provide process control by enabling a process to:

- create and delete subprocesses
- hibernate, then reactivate, a process via the Hibernate/Wake and Suspend/Resume system services

The ability to create subprocesses is granted to a user by the system manager, where the number of subprocesses a job can create is a resource limit. When a process creates a subprocess, it can give the subprocess all or some of its privileges, and its resource quotas and limits are shared with the subprocess. Other resource quotas are shared between the creator and the subprocess.

The Hibernate/Wake and Suspend/Resume mechanisms are methods of process control which are especially efficient in real-time applications. They allow the user to prepare an image for execution and then place it into a wait state until some event occurs which requires its activation.

The Hibernate system service provides the greatest flexibility in sequencing processes for execution. When a Hibernate system service is invoked, normal execution can be resumed only by issuance of a $WAKE system service (or a variant, $SCHDWK, which allows wake-up at an absolute time or at a fixed time interval). However, a hibernating process can be interrupted temporarily by the delivery of an AST (Asynchronous System Trap). When the AST service routine completes execution, the process continues hibernation. If, however, the process calls the $WAKE system service during execution of the AST service routine, the process wakes itself after the service routine completes. Figure 6-3 shows an example of a program which uses the hibernate and wake system services.

Using the $SUSPEND system service, a process can place itself or another process into a wait state similar to hibernation. However, a suspended process cannot be as easily activated as a hibernating one. It cannot, for example, be interrupted by delivery of an AST. Nor can it wake itself, but can only resume normal execution following issuance of a $RESUME system service by another process. Table 6-1 summarizes the differences between hibernation and suspension.

The interprocess system services can be used by a process to control another process executing in the same group. While only an owner process can create and delete subprocesses, a process can be given the privilege to suspend, resume, and wake other processes in its group.

Jobs with sufficient privilege can also create detached processes, and delete, suspend, resume, or wake any process in the system. These privileges are normally reserved for the operating system or the system manager.

Interprocess Communication Facilities

In addition to providing process control services, the oper-

```
Process: GEMINI

ORION: .ASCID 'ORION'                          ;SUBPROCESS NAME
FASTCOMP: .ASCID 'COMPUTE.EXE'                 ;IMAGE

                .
                .
1       $CREPRC_S PRCNAM=ORION,—
                  IMAGE=FASTCOMP               ;CREATE ORION - HE'LL
                                               ;SLEEP
          BLBC    R0,ERROR                     ;BRANCH IF SERVICE ERROR
                                               ;CONTINUE
                .
                .
                .
3         $WAKE_S PRCNAM=ORION                 ;WAKE ORION
          BLBC    R0,ERROR                     ;BRANCH IF SERVICE ERROR
                .
                .
                .
          $WAKE_S PRCNAM=ORION                 ;WAKE ORION AGAIN
          BLBC    R0,ERROR                     ;BRANCH IF SERVICE ERROR

Process: ORION

FASTCOMP:
2         .WORD   0                            ;ENTRY MASK
10$       $HIBER_S                             ;SLEEP 'TIL GEMINI WAKES
                                               ;ME
          BLBC    R0,ERROR                     ;BRANCH IF SERVICE ERROR
                .                              ;PERFORM...
                .
                .
          BRB     10$                          ;BACK TO SLEEP
```

Notes:

1. Process GEMINI creates the process ORION, specifying the image name FASTCOMP.
2. The image FASTCOMP is initialized, and ORION issues the $HIBER system service.
3. At an appropriate time, GEMINI issues a $WAKE request for ORION. ORION continues execution following the $HIBER service call. When it finishes its job, it loops back to repeat the $HIBER call and to wait for another wake.

Figure 6-3
Program Using Hibernate/Wake System Services

ating system provides process communication facilities for synchronizing execution, for sending messages, and for sharing common data. The three techniques that cooperating processes can use to communicate are:

- common event flags
- mailboxes
- shared areas of memory

Common event flags are available by group association to processes within jobs. Mailboxes and shared areas of memory are more general purpose facilities which can be limited or unlimited in scope. They can be limited to a specific member family within a group or to a specific group of jobs, or they can be extended to all jobs in the system.

Common Event Flags

In addition to the local event flags available to each process, cooperating processes can communicate using common event flags. Every group in the system can define any number of common event flag clusters. Each cluster contains 32 flags. The flags can be assigned any meaning for the processes in the group.

Each process in a group can associate with up to two of its group's common event flag clusters at one time. A process can read, set, clear, or wait for common event flags to be set. The ability to read, set, or clear event flags is controlled by the protection code and User Identification Code assigned to the common event flag cluster.

Common event flag clusters can also be used by cooperating processes on different processors in a multiport memory configuration.

Mailboxes

A mailbox is a record-oriented virtual I/O device created by a process. Mailboxes can be used to pass status information, return codes, messages, or any other data from one process to another. A process can protect its mailboxes from read and/or write access by any process outside

its member family or outside its group. Mailboxes can also be used by processes to communicate with other processes on different processors in a multiport memory configuration.

All of the I/O system services and record management services can be applied to mailboxes. Other processes write messages to a process' mailbox by queuing write requests for the device. A process reads messages in its mailbox by queuing read requests for the device. A process can request AST notification when anything is written to its mailboxes, and it can assign mailboxes logical names.

Shared Areas of Memory

The system supports a high degree of code and data sharing through the use of global sections. A global section is a copy of all or a portion of an image or data file that can be mapped in a process virtual address space at run time. Global sections can be used for shared data structures, as communication regions for cooperating processes, or they can be used simply to eliminate multiple copies of image code or data.

Global sections can be created dynamically by a process or they can be permanently present in the system. Dynamically created global sections are mapped into processes that reference them, and deleted when no more references are made to them. Permanent global sections are created by a sufficiently privileged process, and remain until they are explicitly deleted. They are loaded into and removed from memory dynamically as references are made to them.

Each process that maps a global section into its virtual address space can have a different access privilege to a global section. When a global section is created, it is assigned a User Identification Code (UIC) identifying the group and member family to which the global section belongs, and a protection code identifying the read and write access privileges of processes in the system. Global sections can be shared by all processes in the system, or shared only by processes within a particular group, or shared only by processes within a particular job. One or more controlling processes can have write privileges while other processes in the system, group, or job have only read privileges.

A process can map to a global section explicitly by issuing a Map Global Section system service, or it can be mapped implicitly by referring to a shareable image. If an image references a shareable image, the linker does not normally include the shareable image in the image. The shareable image is installed as a global section or set of global sections. When the image is executed, the image activator calls the Map Global Section system service on behalf of the image. For example, the Common Run Time Procedure Library is a shareable image consisting of library procedures that is mapped as a system-wide permanent global section. The use of permanent global sections significantly reduces the size of programs using common library procedures and the overall system memory requirements.

Interprocessor Communication Facility

VAX/VMS support for the multiport memory subsystem means that both user data and subroutines may reside in shared memory for access from multiple processors connected to the multiport memory. All three of the interprocess communication facilities, i.e., common event flags, mailboxes, and global sections, may reside in multiport memory, thus providing interprocessor communication facilities. Through the use of logical names, common event flags, mailboxes, and global sections may be placed either in local or multiport memory, transparently to the program. Common event flags, mailboxes, and global sections are the communication facilities permitted in multiport memory configurations.

MEMORY MANAGEMENT

In a multiprogramming system, many processes coexist simultaneously in main memory. The system switches between these processes, giving each some time to execute. In most multiprogramming environments, however, the number, size, and kind of concurrently executing processes change rapidly, while the amount of memory available for processes remains constant. Users log on and off the system, production activities vary periodically, and special production jobs occur. Since it is generally inefficient to have available the maximum amount of memory that might ever be needed at one time, it becomes the task of the operating system to provide a dynamic memory that responds to the changing multiprogramming environment.

VAX/VMS uses two interdependent complementary techniques to allocate limited memory to competing processes: paging and swapping. These techniques relieve the general programmer of concern for memory allocation while still allowing system programmers to optimize program performance in limited configurations. This section and the following section on scheduling discuss how this system's paging and swapping techniques extend limited memory resources with minimum effect on the system or programs when the system has sufficient memory to hold all concurrently executing processes.

Mapping Processes into Memory

The operating system's memory management software is responsible for creating and maintaining the information used to map the virtual addresses used in a program to physical memory addresses. The unit mapped is the **page**, which is a block of 512 contiguous byte locations in physical memory.

Virtual addresses are also grouped into 512-byte pages, and each page of virtual addresses can be mapped to a page of real memory locations. Any number of virtual pages can be mapped to one physical page. Unlike systems that partition or statically allocate portions of physical memory, this system dynamically allocates physical memory, with the result that pages of a process may be scattered anywhere throughout memory. It is never the concern of the programmer to determine how physical memory is allocated. To illustrate this, Figure 6-4 shows how two processes might be mapped into physical memory.

When a process is created, the operating system sets up its mapping information, called **page tables**. Each process has its own page tables mapped by system region virtual

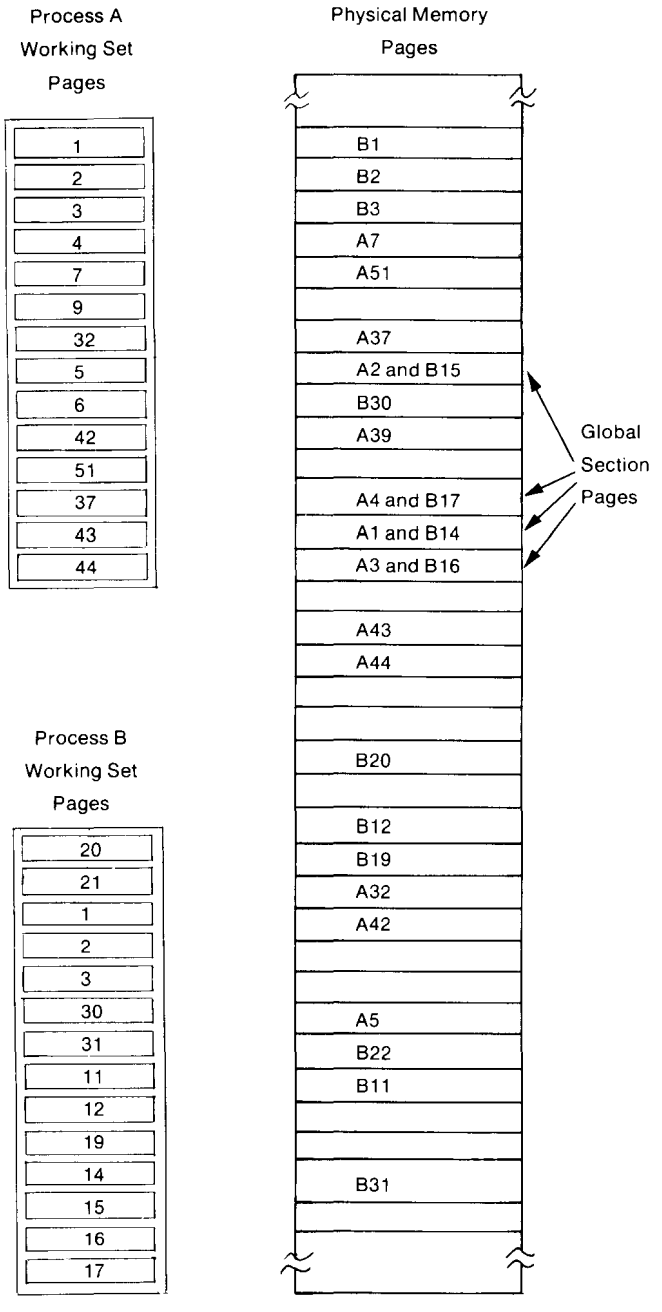


Figure 6-4

Mapping Processes into Memory

addresses. (Refer to the Processor section on memory management for a complete description.) Initially, a process page table simply maps those pages of the control region that define the permanent process context.

When a program is linked, the addresses the linker assigns in the image are always virtual addresses. The linker has no knowledge of how physical memory is allocated. Its primary function is to build descriptions of the size and protection requirements of the program's code and data areas.

When an image is executed, the memory management software uses the linker's descriptions of code and data areas to map image virtual pages to physical pages. The operating system's image activator builds the image's mapping information in the process' page tables.

Process Virtual Memory and Working Set

The total virtual memory requirement of a process is called **process virtual memory**. Process virtual memory consists of all the pages of the process program region and control region which are mapped by the process page tables.

At any one time, some of the pages of process virtual memory may be mapped to disk and some to physical memory. The physical memory requirement of a process is the **process working set**. When a process is executing, a process working set consists of all the pages of a process' virtual memory residing in physical memory that the process can directly access without incurring a page fault, *plus* any actively used portions of the process page tables and process header information.

The working set is a dynamic characteristic of a process that has both minimum and maximum size limits. The system designates a required minimum number of pages that has to be in a process working set, and the system manager defines the maximum number of pages allowed in any one job's working set in the user authorization file. The size of a process working set affects its paging and swapping performance, as well as affecting the number of process working sets that can be resident when the process working set is resident.

A process may increase or decrease its working set, within the authorized limits, through the use of command language commands or system service calls.

Under version 2.0 of VAX/VMS, working set size adjustments are made automatically by the operating system. This facility, when enabled by the system manager, monitors the page fault rate of a process and automatically increases or decreases the working set (again within authorized limits) to optimize performance and memory usage. This automatic adjustment provides a more immediate response in system reaction/performance.

Paging

Through its paging technique, the operating system can execute programs that are too large to fit in the amount of physical memory allocated to a process, without requiring the programmer to define overlays. Inactive portions of a program are automatically stored on disk while the active portions are resident in memory. When the program references a disk-resident portion of the program, the operating system reads in, or **pages** in, the referenced portion, moving out other portions of the program to disk if necessary. This system's paging technique has several features that distinguish it from other techniques:

- clustering, or the ability to read in several pages at one time
- paging processes against themselves, not against the entire system
- maintaining an available page pool from which processes can recover recently discarded pages without incurring disk I/O

- writing back to disk only the modified pages that are released from a process working set and only writing them when several have accumulated
- activating a process waiting for page fault I/O to execute AST routines when they are delivered

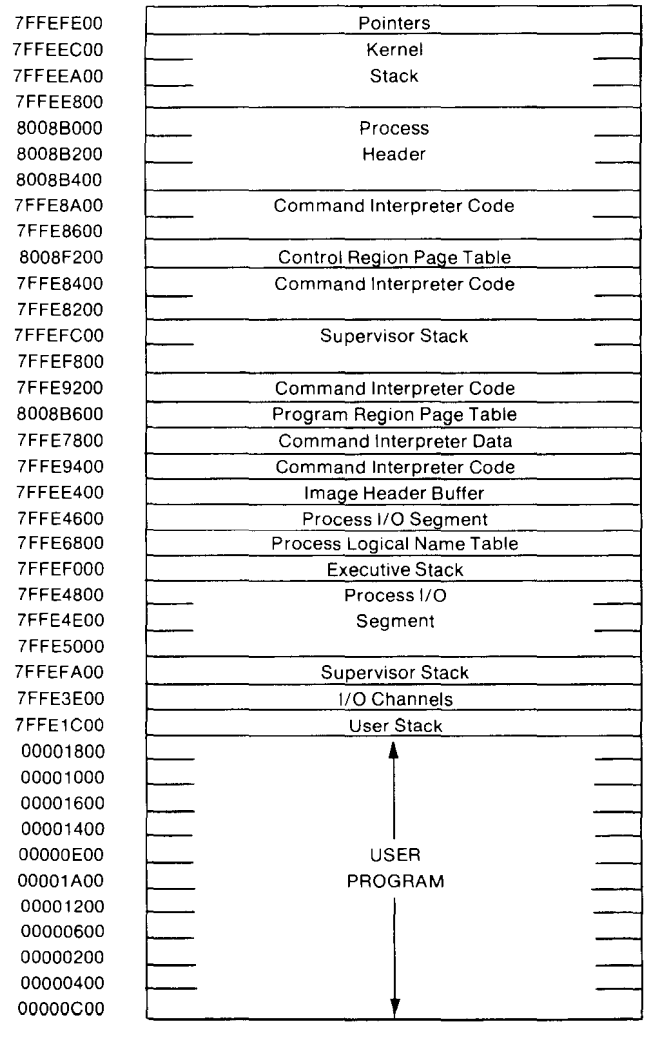
When the operating system activates an image for the first time, a number of pages are read into memory from the image file on disk. The number of pages read in the first time can be controlled by a cluster factor the programmer can assign optionally per image. The ability to read in several pages at once allows the image to execute for some time without incurring page faults, and provides significantly improved responsiveness in starting programs.

A process is subsequently paged only when it executes an image that needs more pages than the process is allowed to have in its working set. If the number of pages in the image plus the number of pages for the remainder of the process is less than the working set size limit, all the pages are read in and the process is never paged.

If all the pages are not read in initially, at some point the image will reference the pages that have not been read in. At that time, the process incurs a page fault, that is, a reference to a page not mapped in the process working set.

The operating system's pager is a condition handler that executes when a process incurs a page fault. If the working set size limit has not yet been reached, the pager reads in the faulted page from disk, plus any additional pages, again according to a cluster factor for that section of the image.

If a page fault occurs when the working set size limit is reached, the pager obtains a page from a pool of available pages to read in the faulted page, and releases the least recently faulted page from the process working set into the pool and writes it to disk if it has been modified. Figure 6-5 illustrates two different size working sets for a process running the same program in each case. The illustration shows the order in which the pages were faulted. (Refer to Figure 6-2 to see how the pages appear in virtual address space.)



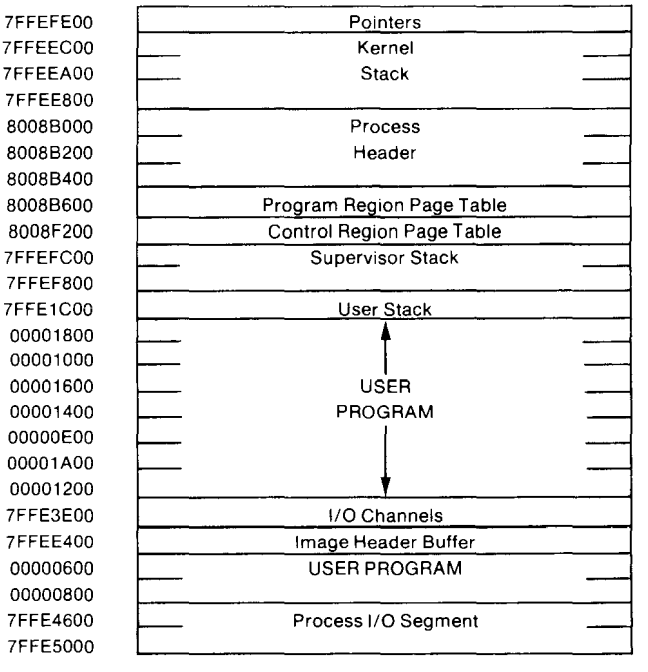


Figure 6-5

Process Working Sets

The pager pages a process only against itself. It does not release pages of one process to satisfy another's needs. This ensures that only those processes that need paging are affected by paging. Other processes in the system need not be affected by another process's memory requirements.

The list of available pages works as a cache of pages that effectively extends a working set size above its limit when few processes are competing for memory resources and there are many pages in the list. If a process faults a page that was released and is still in the list, the page does not have to be read in from disk, it is simply taken from the list and remapped into the working set.

When a page is released, it is placed on one of two lists: the free page list or the modified page list. Modified pages are pages the process has written into and, if they need to be added to the free page list to be used by another process, must be saved on disk. Modified pages are only written to disk when the modified page list exceeds a threshold size or when an image's execution is terminated and the files containing the modified pages must be closed. When modified pages are written, they are writen in clusters to increase system performance.

Virtual Memory Programming

The processor provides the programmer with a large virtual address space and rapid address translation, and the operating system provides the programmer with extremely efficient mapping and paging algorithms. Furthermore, these memory management mechanisms are totally transparent to the application programmer. It is not necessary for a programmer to be concerned with address allocation or page mapping: the high-level language compilers and the linker take advantage of the memory management mechanisms to set up the memory allocation optimal for most programming requirements.

For those systems with limited memory or special processing requirements, however, this system enables users to control and optimize memory management. The system manager can control the memory allocation requirements of the system as a whole by initialization parameters such as desired and minimum acceptable number of available pages, and of individual jobs by user authorization parameters such as paging file usage limit and maximum working set size. It is possible to have a process avoid paging entirely by making its working set size equal to its virtual memory requirements, or to reduce paging by choosing a working set size that satisfies the average demand for pages over time.

The programmer also has the ability to control memory allocation for images in two ways: through properly coded programs and through the memory management system services. This system's memory management software optimizes for program locality. Programs that incur paging infrequently are those in which the code and data used during each stage of processing are contained in the fewest possible number of virtually contiguous pages.

For the most part, the linker allocates virtual addresses so that images require the minimum mapping information possible. The programmer can also ensure that images that process large data structures require the least possible mapping information and potential paging by organizing data structures as if they were disk-resident files. In general, the programmer need not be concerned with data structures such as tables and arrays whose elements are virtually contiguous and sequentially processed.

Large data structures that are randomly accessed can, however, be optimized. For example, processing down a linked chain in which the chain elements are spaced far apart with no useful data in between requires that an image reference a large number of pages in a short period of time. If all of the pages cannot fit in the process working set at the same time, the references to successive chain elements will incur disk I/O.

On the other hand, a large data structure can be efficiently accessed using directory trees, where a page or set of consecutive virtual pages contains all one kind of information. One page can contain all of the information that points to randomly arranged, but virtually contiguous, pages containing the data processed at that locality.

VAX/VMS Memory Management Services

For those who have special processing requirements, there are system services that control memory management within the quotas and limits assigned by the system manager. These memory management system services enable a process to:

- modify the working set size limit
- add or delete pages from process virtual memory
- expand or contract the program region or control region
- lock pages in the working set
- lock pages in physical memory

A program can impose a limit on process working set size anywhere between the minimum required by the system and the maximum specified by the system manager. The limit can be adjusted in accordance with program behavior and real-time requirements. By maintaining the smallest working set size consistent with an acceptable paging rate, a program that temporarily requires a large working set can reduce its impact on the system. For example, a process control program or simulator might use a small working set while processing interactive initialization commands. Once real-time processing is underway, the program can expand its process working set size to reduce paging. When real-time processing is finished, the program can contract the working set.

A process can add selected pages to and delete selected pages from its virtual memory dynamically. Deleting a page is in effect saying that the image is no longer going to use those virtual addresses, and the operating system does not need to map them to pages in virtual memory. Deleting read/write pages (such as those used for interprocess communication) as soon as they are no longer used eliminates the need for the system to write them out as modified pages to a paging file. When an image has reached its paging file quota, it can delete pages in order to map other pages in its virtual address space.

A process can request an extension to the amount of virtual memory allocated to its program region. The operating system will map zero-filled pages into the process virtual address space following the highest addressed page allocated for the program region. This service is useful for dynamically creating data arrays whose size is not known be-

forehand, and it eliminates the need for allocating a data area in a program image. A process can also extend the initial allocation of pages for the user stack by requesting the operating system to map zero-filled pages into process virtual address space preceding the lowest addressed page allocated for the control region. In Version 2.0 of VAX/VMS, the system will automatically extend the stack if the process references unmapped addresses in the control region.

In unusual situations, a process can lock pages in its working set. Locking a page in the working set is useful when a process does not reference a particular page regularly, but the page needs to be in the working set to increase the performance of the code in that page. For example, it might be desirable to keep the page containing asynchronous system trap routines in a working set to ensure that the routines are started up rapidly when an AST is delivered. Note, however, that locking a page in the working set causes other pages to be paged more frequently, since the page will not be paged out, no matter how long it has been in the working set. A page can be unlocked when it is no longer necessary to keep it in the working set.

It is also possible to lock pages in memory. A page locked in memory is not only locked in the working set, it is not swapped out with the process. This service is useful for real-time processes that need to keep buffers in memory for I/O transfers.

PROCESS SCHEDULING

VAX/VMS features event-driven scheduling based on process priority. Unlike traditional timeshared scheduling systems, this system's ability to respond to events enables it to dispatch real-time processes efficiently as well as to share processing time among normal processes competing for resources. Furthermore, priority assignment enables the user to bias processor time allocation based on process activity, to bias the allocation absolutely for certain processes, or to mix both allocation methods.

The operating system's scheduler and swapper are responsible for ensuring that the processes executing in the system receive processor time commensurate with their priority, which is controlled by assignment, and with their ability to execute, which is controlled by system events.

System Events and Process States

In VAX/VMS, dispatching a process for execution involves little decision making. The selected process is always the highest priority executable process. The real scheduling decisions are made as the result of system events that make processes executable.

A system event is an event that affects the ability of a process in the system to execute. System events include events external to the process currently executing, such as I/O completion or timer interrupt. System events also include events internal to the process currently executing. The process may issue a wait request or a hibernate request, or it may request or release a system resource, for example, a page of memory.

Every active process in the system is listed in one of several state queues that identifies whether or not a process is executable, and if not, the event or resource for which the process is waiting. Whenever a system event occurs, the scheduler adjusts the process state queues accordingly. For example, the scheduler adds a process to the executable state queue when a resource for which it is waiting becomes available, or removes it when it requests an event or resource for which it must wait.

The executable state queue supplies the scheduler with a list of processes that are eligible to execute. Priority determines which process among those eligible executes. Rescheduling occurs when a system event makes executable a process with higher priority than the one currently executing.

Unlike timeshared scheduling, therefore, event-driven scheduling is based on the activities of the processes themselves, not on a time limit imposed by the scheduler. Because scheduling intervals are determined by system events, the interval between rescheduling is random. Quantum keeping and requested timer events provide a minimum level of event activity but, in practice, the average interval between events is determined by the duration of the typical I/O operation.

Priority: Real-Time and Normal Processes

The scheduler recognizes 32 scheduling priorities, where priority 31 is high and 0 is low. Priorities 31-16 are for real-time processes, and priorities 15-0 are for normal processes. When a process is created, the system assigns it a scheduling priority. A program image that the process executes can modify the process priority using a system service. The system manager grants jobs the privilege to execute at real-time priorities.

The scheduler maintains a queue for each scheduling priority. Processes having the same priority are listed in the same queue. The priority assigned to a process when it is created is its base priority. The scheduler does not alter the priority of a real-time process during execution. The scheduler may temporarily increase the priority of a normal process during its execution, but its priority never drops below its base priority.

Scheduling by strict priority for real-time processes and by potentially modifying priority for normal processes allows the scheduler to achieve maximum overlap of compute and I/O activities while still remaining responsive to high-priority real-time applications.

Scheduling Real-Time Processes

When a system event occurs that makes a real-time process eligible to execute, it receives control of the processor unless another higher priority process is currently executing. A real-time process retains control of the processor until it finishes execution, enters a wait state, or is pre-empted by a higher priority process. (Note that under VAX/VMS, real-time processes actually have a higher priority than system processes, thus ensuring that real-time processing will never be encumbered by system overhead.)

A higher priority real-time process can pre-empt any lower priority process whenever a system event occurs that makes it eligible to execute. For example, a device interrupt may occur that signals the completion of an I/O transfer requested by the higher priority real-time process.

When a real-time process is pre-empted to dispatch a

process of higher priority, the pre-empted process is placed at the end of its priority queue. This rotates processes within a priority, with the result that available processor time is distributed among processes of the same priority.

Scheduling Normal Processes

When no real-time processes are executing, the scheduler distributes processor time among the processes on the normal priority levels. As with real-time processes, the scheduler selects the highest priority ready-to-execute normal process. That process executes until it finishes execution, enters a wait state, or is pre-empted by a higher priority process. Unlike real-time process scheduling, however, the scheduler modifies normal process priority whenever a system event occurs for a normal process and whenever a normal process is scheduled.

When a system event occurs that affects a normal process, the scheduler increases the priority of the normal process (but not to more than the maximum priority of 15) and places the process at the tail of the queue for its new priority. The amount of priority increment depends on the nature of the event. For example, the scheduler increases the priority of a normal process on the following events:

- terminal input completed
- terminal output completed
- resource available
- wake, resume, delete request received
- nonterminal I/O completion, page fault completion, or other event

In this case, the terminal I/O events receive the highest priority increments to enable the system to be most responsive to the interactive terminal user. When the scheduler increases a normal process's priority, that process gets control of the processor if its new priority is higher than that of the process currently executing.

Each time a normal process is scheduled, the scheduler decreases its priority by one (unless it is already in its base priority queue) and places it at the end of that priority queue. The effect of dynamically increasing and decreasing normal process priority ensures maximum overlap of computation and I/O.

Swapping and the Balance Set

It is the job of the swapper to keep the scheduler supplied with the highest priority executable processes in configurations that do not have a sufficient amount of physical memory to keep all process working sets memory-resident. The balance set is the set of all process working sets that are currently in memory. The swapper ensures that the balance set always contains the highest priority executable processes by moving low priority or nonexecutable memory resident process working sets to a swap area on disk, and moving high priority or executable process working sets into memory.

Swapping is a very efficient way of extending limited memory resources when many processes are executing concurrently. Process working sets for small processes (less than 64K bytes or 128 pages) can be swapped in and out of memory in one disk I/O operation. Where paging extends limited memory resources on a per-process basis and is limited to moving few pages in and out of memory, swapping balances the memory requirements of the system as a whole.

The swapper is activated whenever a system event occurs that can make a nonresident process resident, a nonresident process executable, or a resident process nonexecutable. For example, a resident process might release sufficient memory to enable the swapper to move in a nonresident process. An I/O completion event might make a nonresident process executable. A resident process might enter a wait state and become nonexecutable. In any case, the swapper uses three conditions to determine which processes should be swapped in and which should be swapped out:

- which processes are executable and which are not (and the reason for the wait state)
- what the process priorities are
- whether a process balance set quantum has expired

The balance set quantum effectively enforces a swapping rotation for compute-bound normal processes. Every normal process is assigned a time quantum that provides a guaranteed minimum amount of time in which the process can perform useful work before it is eligible to be swapped out of the balance set. A process can be pre-empted many times before it has received its full quantum. It remains in the balance set until it completes its first quantum unless a real-time process that is swapped out becomes executable and no other processes can be swapped out to make room for the real-time process.

VAX/VMS Process Control Services

In addition to the VAX/VMS system services that enable processes to create, delete, suspend, resume, and wake other processes, or to hibernate and wake themselves, a process can control the manner in which it is scheduled by:

- setting process swap mode
- setting resource wait mode

A suitably privileged process can request that it not be swapped out of the balance set, even when it becomes inactive. This is useful for high priority real-time processes that need to be activated rapidly when they become executable.

Normally, when a process requires dynamic resources of the system and they are not available, the process enters a wait state until the resources become available. Dynamic resources primarily include the buffer space needed for mailboxes, I/O requests, etc. A process can request to be notified when resources are not available and take alternative action instead of entering a wait state.

I/O PROCESSING

The I/O processing system consists of several modular, interdependent components that enable programmers to choose the programming interface and processing method appropriate for their needs, without incurring run time space or performance overhead for features not used. In addition, the I/O request processing software takes advantage of the hardware's ability to overlap I/O transfers with computation, switch contexts rapidly, and

generate interrupts on multiple priority levels to ensure the maximum possible data throughput and interrupt response. Figure 6-6 presents an overview of the major I/O processing system components and their relationships.

Programming Interfaces

The I/O programming tools are the record management system, VAX-11 RMS, for general purpose file and record processing, and the Queue I/O system services, for direct I/O processing. Table 6-2 summarizes the programming interfaces.

RMS (refer to the Data Management Facilities Section) provides device-independent access to file-structured I/O devices. The most general purpose type of access enables programs to process logical records; RMS automatically provides record blocking and unblocking.

RMS users can also choose to perform their own record blocking on file-structured volumes such as disk and magnetic tape, either to control buffer allocation or optimize special record processing. Users performing their own record blocking address blocks using a virtual block number (which is the number of the block relative to the file being processed) for volume-independent processing.

The I/O system services provide both device-independent and device-dependent programming. Users perform their own record blocking on file-structured and non-file-structured devices. Virtual block addressing is used on Files-11 disk or ANSI magnetic tape volumes. In addition, users with sufficient privilege can perform I/O operations using either logical or physical block addressing for defining their own file structures and accessing methods on disk and magnetic tape volumes.

The I/O system services also provide device-dependent programming of devices not supported by RMS, such as real-time interfaces.

Ancillary Control Processes

Both RMS and the I/O system services use the same I/O control processes, called ancillary control processes (ACPs), for processing file-structured I/O requests. An ACP provides file structuring and volume access control for a particular type of device. Typical ACP functions would include creating a directory entry or file, accessing or deaccessing a file, modifying file attributes, and deleting a directory entry or file header. There are three kinds of ACPs provided in the system: Files-11 disk, ANSI magnetic tape, and network communications link.

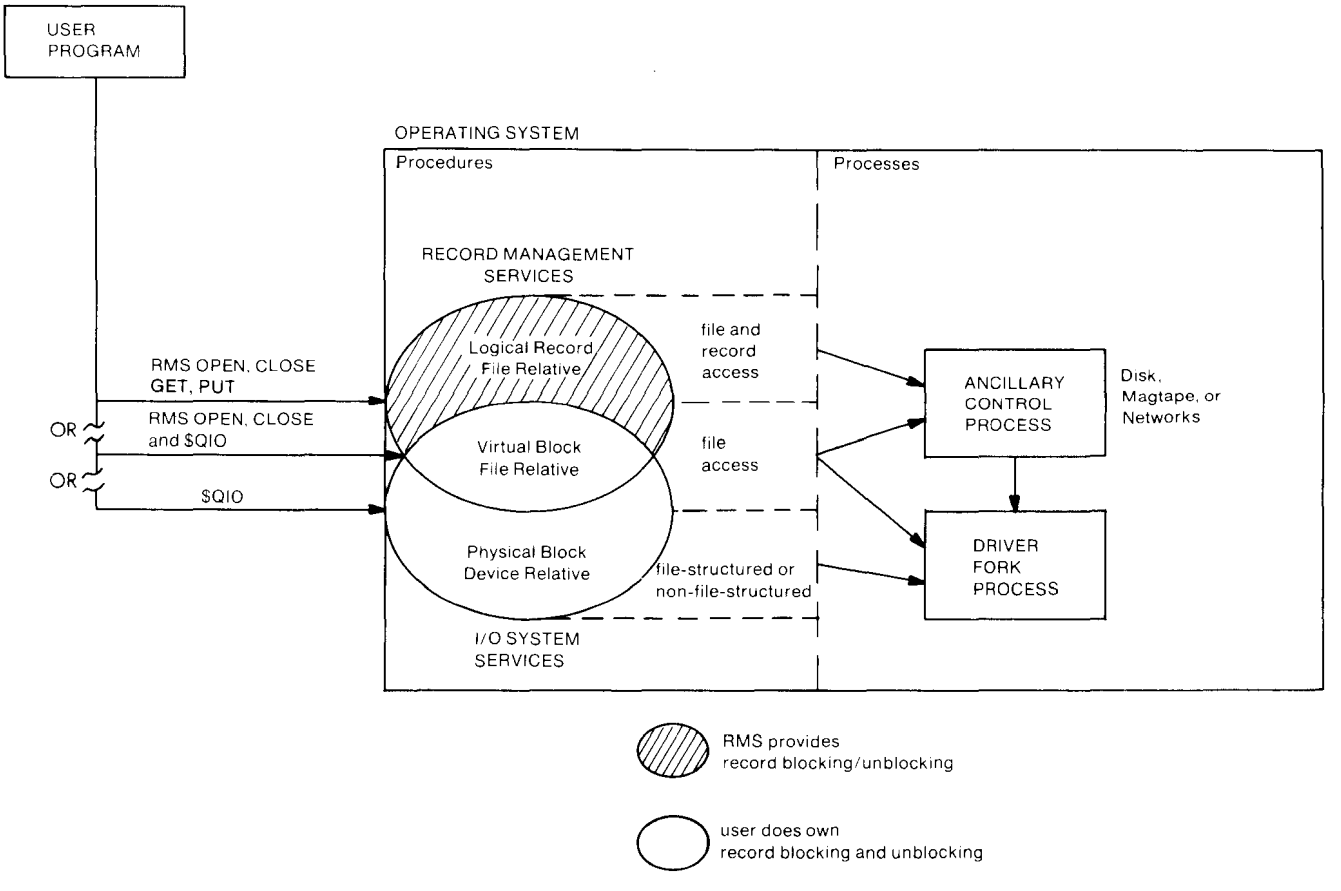


Figure 6-6

User Interfaces to I/O Services

Table 6-2

I/O Processing Interfaces

METHOD	PROGRAM INTERFACE	I/O COMPONENTS	PURPOSE
Record I/O	RMS requests	RMS, ACP and Driver	Use Files-11 disk or ANS magtape file structure, use RMS record access methods
File I/O	RMS OPEN and $QIO requests	RMS for OPEN, ACP and Driver	Use Files-11 disk or ANS magtape file structure, implement own record access methods
Device I/O	$QIO requests	Driver	Fast dumps to disk or magnetic tape, foreign file structure

The RMS and I/O system services programming interfaces are the same regardless of the ACP involved, but since ACPs are particular to a device type, they do not have to be present in the system if the device is not present. There is one network ACP process for all DECnet network communications links in the system, and none if the system is not in a network.

Device Drivers

Once the ACP sets up the information for file-structured I/O requests, a request can be passed on to a device driver. All non-file-structured I/O requests are passed directly to a device driver.

A VAX/VMS driver:

- defines the peripheral device for the rest of the VAX/VMS operating system
- defines the driver for the operating system procedure that maps and loads the driver and its device data base into system virtual memory
- initializes the device (and/or its controller) at system startup time and after a power failure
- translates software requests for I/O operations into device-specific commands
- activates the device
- responds to hardware interrupts generated by the device
- reports device errors
- returns data and status from the device to software

Device drivers work in conjunction with the VAX/VMS operating system. The operating system performs all I/O processing that is unaffected by the particular specifications of the target device (i.e., device-independent) processing. When details of an I/O operation need to be translated into terms recognizable by a specific type of device, the operating system transfers control to a device driver (i.e., device-dependent processing). Since different peripheral devices expect different commands and setups, each type of device on VAX/VMS requires its own supporting driver.

The VAX/VMS operating system contains device drivers for a number of standard DIGITAL-supported devices. These include both MASSBUS and UNIBUS devices. In addition, the user can write additional drivers for non-standard UNIBUS devices.

I/O Request Processing

All I/O requests are generated by a Queue I/O (QIO) Request system service. If a program requests RMS procedures, RMS issues the Queue I/O Request system service on the program's behalf. Queue I/O Request processing is extremely rapid because the system can:

- keep each device unit as busy as possible by minimizing the code that must be executed to initiate requests and post request completion
- keep each disk controller as busy as possible by overlapping seeks with I/O transfers

The processor's many interrupt priority levels improve interrupt response because they enable the software to have the minimum amount of code executing at high priority levels by using low priority levels for code handling request verification and completion notification. In addition, device drivers take advantage of the processor's ability to overlap execution with I/O by enabling processes to execute between the initiation of a request and its completion. User processes can queue requests to a driver at any time, and the driver immediately initiates the next request in its queue upon receiving an I/O completion interrupt.

All access validation and checking takes place before an I/O request is actually queued. For file-structured I/O requests, the Queue I/O Request system service obtains all the virtual block mapping and volume access checking information from the ACP or directly from tables created by the ACP. For example, on virtual block I/O requests for multivolume files, the system service obtains from the ACP's tables the mapping information that enables it to queue requests to different drivers when the user's I/O request involves a transfer that spans volumes. The Queue I/O Request system service also checks the validity of the function requested (read, write, rewind, etc.) for the particular device. Because all access validation and function checking is performed before the request is queued, the driver has little to do to initiate a request.

Once the system service has verified the I/O request, it raises the interrupt priority level to that of the driver. The only activity it has to perform at this level is a test to see if the driver is busy. If the driver is not busy, it calls the driver. Otherwise, it queues the request according to the priority of the requesting process and immediately returns to the user process.

When the driver is called, it initiates the request and returns to the user process. Because disk seeks do not require the controller once they are initiated, if a disk driver receives a seek request and the controller is currently busy with an I/O transfer request on some other disk unit, the driver queues the request so that the controller will initiate the seek request before any pending I/O transfers when it has finished the current transfer.

When the device subsequently generates its interrupt at the hardware interrupt priority level, the interrupt dispatcher calls the appropriate interrupt service routine. An interrupt service routine simply saves the device control/status registers, requests a software interrupt at the driver's interrupt priority level, and returns to the interrupt dispatcher which is then free to scan for unit attentions. Because a disk controller cannot generate interrupts on any unit performing a seek until the current transfer completes, the interrupt dispatcher will also dispatch seek completion when dispatching a disk I/O transfer completion interrupt.

When the driver receives the completion interrupt, it prepares the I/O completion status for the requester, and requests a software interrupt. The driver is then free to process another request in its queue and, if the queue is not empty, the driver begins again. All I/O completion notification takes place outside the driver, minimizing the interrequest idle time. The I/O post routine notifies the process of I/O completion and releases or unlocks buffers.

COMPATIBILITY MODE OPERATING ENVIRONMENT

The processor can execute user mode PDP-11 instruction streams in the context of a process. The operating system supplements this feature by substituting its functionally equivalent system services for many of the RSX-11M operating system executive directives that user mode tasks may call. This enables the system to execute such non-privileged RSX-11M task images as:

- the PDP-11 MACRO assembler
- the PDP-11 FORTRAN IV/VAX to RSX compiler
- the PDP-11 BASIC-PLUS-2/VAX compiler
- the RSX-11M program development and file management utilities, including the task builder, text editor, etc.

In addition, the operating system supports the RMS-11 and RMS-11K record management services procedures for compatibility mode programs. Program and data files can therefore be transported between VAX and RSX systems.

The operating system also supports the RSX-11M Monitor Console Routine (MCR) commands, either typed directly on a terminal, or submitted as indirect command files.

User Programming Considerations

Any PDP-11 BASIC-PLUS-2/VAX, PDP-11 FORTRAN IV/VAX to RSX, or PDP-11 MACRO program can be executed in compatibility mode, provided that it is first linked by the RSX-11M Version 3.2 task builder and that the resulting task image meets the following requirements:

- it must not execute PDP-11 privileged instructions
- it must have been built for a mapped system
- it must not depend on 32-word memory granularity
- it must not use the privileges that enable it to map into the executive or I/O page
- it must not use the PLAS (program logical address space) executive directives
- it must not rely on environmental features of RSX-11M that VAX/VMS does not support, e.g., partitioning or significant events
- it must not use DECnet

The task can be privileged to issue directives other than memory management directives—direct volume access using the QIO request executive directive, for example. IAS or RSX-11D tasks that meet these requirements can also be executed. They must first be built with the RSX-11M Version 3.2 task builder. For programs that do not meet these requirements, VAX/VMS provides the program development utilities (for example, the MACRO assembler and the task builder) for modifying programs to execute in compatibility mode.

For most RSX-11M executive directives, the native mode operating system executes a functionally equivalent system service. In most cases, the system service duplicates the function. For example:

- A checkpoint enable/disable directive is interpreted as the set swap mode system service.
- The send/receive directives are translated into mailbox write/read system services. Native mode and compatibility mode images can communicate using mailboxes.
- The event flag directives are for the most part identical. Native mode and compatibility mode images can communicate using common event flags, provided they are in the same group.
- A Logical Unit Number (LUN) assignment directive is interpreted as a channel assignment for the appropriate device.

In some cases the operating system cannot duplicate the function, but it does what it can to let a program continue. For example:

- A task image is allowed to declare a significant event, but the directive is ignored.
- A set priority directive is ignored, since the scheduling priority ranges are different. To run at a given priority, the image must be run in the context of a process given that priority.

For the most part, however, many RSX-11M and VAX/VMS program environment characteristics correspond. For example, tasks can hibernate, receive asynchronous system traps, and schedule wake requests. Synchronous system trap routines can be declared as condition handlers for trace traps, breakpoint traps, illegal instruction traps, memory protection violations, and odd address errors.

File System and Data Management

Both RSX-11M and VAX/VMS recognize User Identification Codes as a protection mechanism. UICs provide the default user file directory in RSX-11M systems, while, in VAX/VMS, a UIC is not necessarily associated with an account name or default directory name. UIC-based file protection, however, is much the same in both systems. That is, it is used in determining read, write, and delete privileges for system, owner, group, and world.

Tasks may use any of the RSX data management services including File Control Services (FCS), RMS-11, and RMS-11K. Special versions of FCS and RMS-11/RMS-11K are supplied with VAX/VMS. A compatibility mode task built on VAX/VMS is thus provided with the full file naming capabilities of VAX/VMS, including logical names and multilevel directories. However, update of a file by multiple tasks is not supported.

Both magnetic tape and Files-11 disk volumes can be transported between systems. VAX/VMS can read and write both Files-11 Level 1 disk structures (ODS-1) and the Level 2 disk structures (ODS-2). The Extend access protection field in ODS-1 is used for Execute access protection in ODS-2. While reading files stored on ODS-1 volumes, therefore, this protection field is ignored.

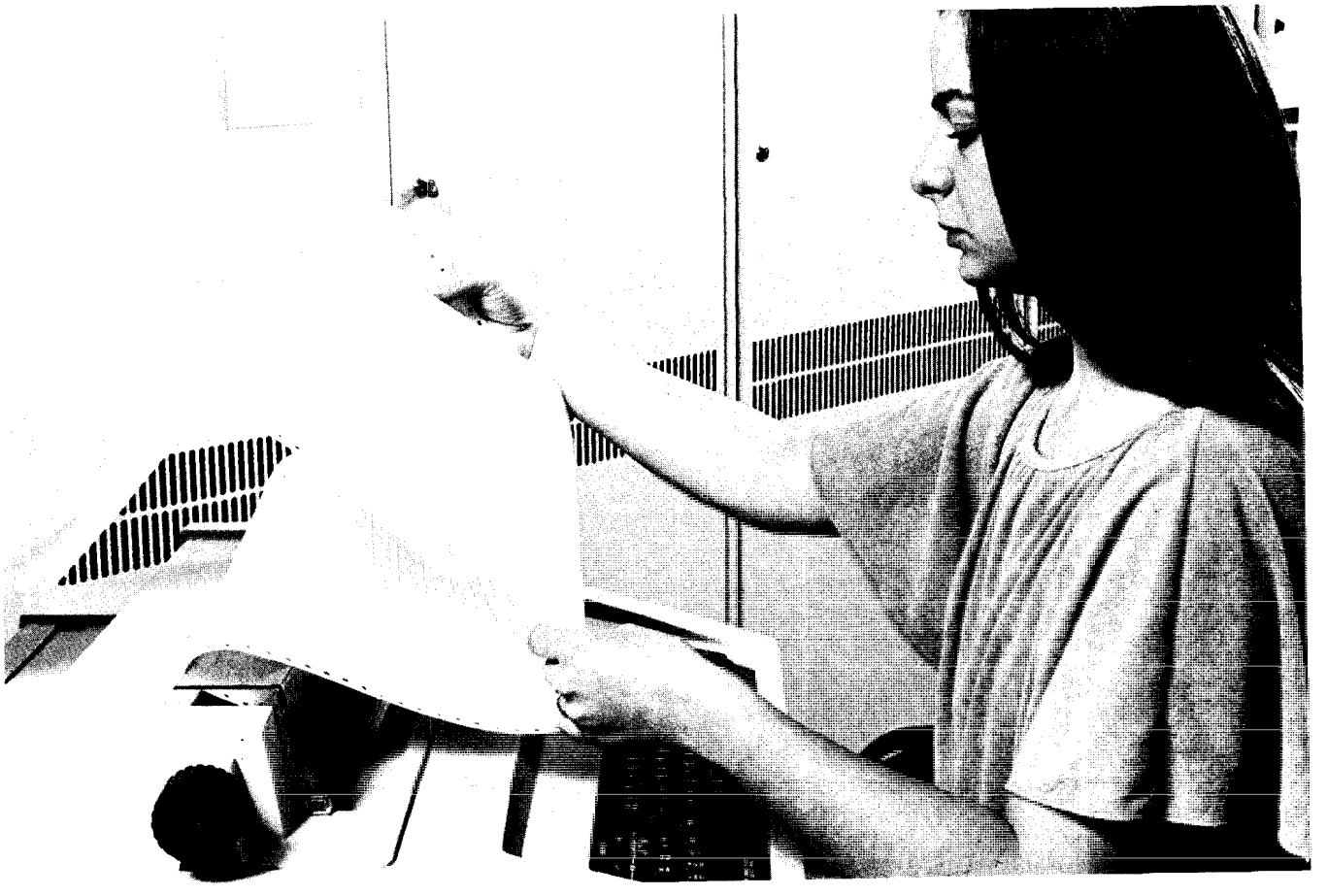
Command Languages

VAX/VMS users can select the MCR command interpreter, which allows them to execute a language (VAX/VMS MCR) that is similar to the RSX-11M MCR command language. Selecting the MCR command interpreter allows the VAX/VMS user to perform the following:

- Run RSX-11M images and VAX-11 images.
- Use RSX-11M components for RSX-11M program development, for example, MACRO-11 or the task builder.
- Use VAX/VMS components for native program development, for example, VAX-11 MACRO or the linker.
- Execute RSX-11M indirect command files. (The VAX/VMS user can use this facility to execute those files required for RSX-11M or RSX-11S system generation.)

VAX/VMS will associate the MCR command interpreter with a process, if "MCR" is the default command interpreter named in the user's authorization file entry or if the user specifies /CLI=MCR following "username" in the LOGIN statement (overriding the default).

7
The Languages



VAX/VMS includes a complete program development environment for a wide range of languages. In addition to the native assembly language, VAX/VMS offers many optional high-level programming languages commonly used in developing both scientific and commercial applications: FORTRAN, COBOL, BASIC, PL/I, PASCAL, CORAL 66, and BLISS-32. It provides the tools necessary to write, assemble or compile, and link programs, as well as build libraries of source, object, and image modules.

Programmers can use the system for development while production is in progress. They can interact with the system on-line, execute command procedures, or submit command procedures as batch jobs. Novice programmers can learn the system quickly because the command language accepts standard defaults for invoking the editors, compilers, and linker. Experienced programmers will appreciate the flexibility and control each tool offers.

INTRODUCTION

VAX/VMS provides a complete program development environment. In addition to the assembly language, MACRO, it offers the optional higher level languages commonly needed in engineering and scientific, commercial, instructional, and implementation applications—FORTRAN, COBOL, BASIC, PL/I, PASCAL, CORAL 66, and BLISS-32. VAX/VMS provides the tools to write, assemble or compile, and link programs, as well as to build libraries of source, object, and image modules. User applications may employ more than one language, and the ability of languages to call one another allows concatenation of application segments written in a variety of languages, provided they satisfy certain criteria.

These native mode language processors produce native object code, and take advantage of the native instruction set and 32-bit architecture of the VAX hardware.

In addition, there is the host development mode programming environment which provides support for PDP-11 BASIC-PLUS-2/VAX, PDP-11 FORTRAN IV/VAX to RSX, and MACRO-11. These produce compatibility mode object code.

VAX COMMON LANGUAGE ENVIRONMENT

An important feature provided by VAX is a "common language" environment, i.e., the VAX languages adhere to a specific set of standards, including:

- symbolic debugger interface
- use of the symbolic traceback facility
- use of the Common Run Time library
- conformance to the VAX calling standard which allows calls among any set of VAX languages, to VAX/VMS system services and to SORT and FMS subroutines
- common handling of exceptions
- use of VAX-11 RMS for record handling

Symbolic Debugger Interface

VAX/VMS provides facilities to aid the debugging of programs written in native mode. It accomplishes this via a program known as the interactive symbolic debugger. The debugger can be linked with a native program image to control image execution during development. It can be used interactively or can be controlled from a command procedure file. The debugging language is similar to the VAX/VMS command language. Expressions and data references are similar to those of the source language used to create the image being debugged. Debugging statements can be conditionally compiled.

Debugging commands include the ability to start and interrupt program execution, to step through instruction sequences, to call routines, to set break or trace points, to set default modes, to define symbols, and to deposit, examine, or evaluate virtual memory locations.

Symbolic Traceback Facility

VAX/VMS supports the Symbolic Traceback Facility. This is a run time facility that aids programmers in finding errors by describing the call sequences that occurred prior to the error. The traceback facility is automatic and does not require that any special qualifiers be included with the FORTRAN or LINK commands (but it can be suppressed by specifying NOTRACE with the LINK command).

When an error condition is detected, the error message is displayed by the run time library indicating the nature of the error and the address at which the error occurred (user PC). This is followed by the traceback information, which is presented in inverse order to the calls. For each call frame, traceback lists module name, routine name, source program line, and absolute and relative PC. Using this information, the programmer can usually locate the source of the error in a relatively short period of time.

Common Run Time Library

The VAX-11 Common Run Time Procedure Library contains sets of general purpose and language-specific procedures. User programs call these procedures to perform specific tasks required for program execution. Both VAX-11 MACRO and native mode high-level language programmers can use any of the Run Time Library procedures in any combination. Because all procedures follow the same programming standards and make no conflicting execution assumptions, a language-independent common run time environment is provided for user programs. Such an environment encourages a user program to be composed of procedures written in different languages, and thus increases programming flexibility.

VAX Calling Standard

The VAX-11 procedure calling standard defines and supports the mechanisms for passing arguments between modules of major VAX-11 software subsystems such as languages, VAX-11 RMS, and the VAX/VMS operating system. The standard facilitates the calling of a procedure written in one language from a program written in another language.

Exception Handling

The mechanisms defined by the VAX-11 calling standard are also used by the condition handling facility to signal the occurence of exceptions detected by hardware or software.

VAX-11 RMS

VAX-11 Record Management Services (RMS) is the technique programmers use to handle record I/O within programs. VAX-11 RMS routines are system routines that provide an efficient and flexible means of handling files and their data. Typically, VAX-11 RMS routines allow the programmer to create a file and:

- accept new input
- read or modify data
- produce output in a meaningful form

High-level language programmers normally use the I/O facilities of their particular language to perform record and file operations. These operations are implemented using the VAX-11 RMS facilities. VAX-11 MACRO programmers can use the VAX-11 RMS routines directly within their programs.

VAX-11 RMS routines are an integral part of the operating system. The programmer need not perform any special linking or declaring of global entry points for the routines.

Furthermore, calls to VAX-11 RMS routines are consistent with the VAX calling standard.

The elements of the common language environment are discussed more fully as they apply to each individual VAX language. Introduced below are each of the VAX-supported languages, their attributes, characteristics, and sample coding.

VAX-11 FORTRAN

Introduction

VAX-11 FORTRAN is an optional language processing system whose language specifications are based on the American National Standard FORTRAN X3.9-1978 (commonly called FORTRAN-77). The VAX-11 FORTRAN compiler supports this standard at the full-language level. At the same time, it provides optional support for certain FORTRAN features based on the previous ANSI standard, X3.9-1966. The VAX-11 FORTRAN compiler performs the following functions:

- produces highly optimized VAX native object code
- makes use of the VAX floating point and character string instructions
- produces shareable code

The VAX-11 FORTRAN language is upwardly compatible with the PDP-11 FORTRAN language. Table 7-1 lists extensions to the ANSI FORTRAN-77 language. Table 7-2 lists the features of FORTRAN-77.

Some characteristics of VAX-11 FORTRAN are described below.

File Manipulation

OPEN and CLOSE statements extend the file management characteristics of the FORTRAN language. An OPEN statement can contain specifications for file attributes that direct file creation or subsequent processing. Attributes include: file organization (sequential, relative, indexed); access method (sequential, direct, keyed); protection (read-only, read/write); record type (formatted, unformatted); record size; and file allocation or extension. The program can also specify whether the file can be shared, and whether the file is to be saved or deleted when closed. The OPEN statement can contain an ERR keyword which specifies the statement to which control is transferred if an error is detected during OPEN.

Of particular interest is the VAX-11 FORTRAN support for the Indexed Sequential Access Method (ISAM), a powerful keyed input/output file access capability. VAX-11 FORTRAN is able to create, read, and write indexed (and relative) files. In addition, FORTRAN is able to reference a relative or indexed file already created by another language (for instance, COBOL), provided the file and data formats and the key information are compatible.

Simplified I/O Formats

List-directed input and output statements provide a method for obtaining simple sequential formatted input or output without the need for FORMAT statements. On input, values are read, converted to internal format, and assigned to the elements of the I/O list. On output, values in the I/O list are converted to characters and written in a fixed format according to the data type of the value.

Character Data Type

A program can create fixed-length CHARACTER variables and arrays to store ASCII character strings. The VAX-11 FORTRAN language provides a concatenation operator, substring notation, CHARACTER relational expressions, and CHARACTER-valued functions. CHARACTER constants, consisting of a string of printable ASCII characters enclosed in string quotes, can be assigned symbolic names using the PARAMETER statement. Operations which use CHARACTER strings are more efficient and easier to use than their analogs using arithmetic data types. VAX/VMS provides a set of character manipulation instructions that are FORTRAN-callable (e.g., LIB$LOCC, locate a character in a string).

Figure 7-1 illustrates two VAX-11 FORTRAN subroutines. This figure illustrates the use of the FORTRAN CHARACTER data type and some of the VAX-11 FORTRAN extensions to FORTRAN-77. The first subroutine, which reverses a character string, illustrates CHARACTER declarations (both fixed and passed length), the intrinsic function LEN, substring manipulation, and the ENDDO statement. The second subroutine locates a substring of a character string and marks the starting position of the substring. This subroutine illustrates CHARACTER declarations (both fixed and passed length), assignments of character values to variables, the intrinsic function INDEX, substring manipulation, and the FORTRAN-77 block IF statement.

```
      SUBROUTINE REVERSE(S)
      CHARACTER T, S*(*)
      J = LEN(S)
      DO I=1, J/2
          T = S(I:I)
          S(I:I) = S(J:J)
          S(J:J) = T
          J = J-1
      ENDDO
      END
      SUBROUTINE FIND_SUBSTRINGS(SUB, S)
      CHARACTER*(*) SUB, S
      CHARACTER*132 MARKS
      I = 1
      MARKS = ' '
10    J = INDEX(S(I:), SUB)
      IF (J .NE. 0) THEN
          I = I + (J-1)
          MARKS(I:I) = '#'
          I = I+1
          IF (I .LE. LEN(S)) GO TO 10
          ENDIF
      WRITE(6,91) S, MARKS
91    FORMAT( 2(/1X, A))
      END
```

Figure 7-1
FORTRAN CHARACTER Data Type Program

Table 7-1
Language Extensions to FORTRAN-77, X3.9-1978

VAX-11 FORTRAN

31-character symbolic names	Symbolic names used to identify programs, subprograms, external functions and subroutines, COMMON blocks, variables, arrays, symbolic constants, and statement functions can be longer than the standard six characters. Symbolic names can include letters, digits, dollar sign, and underscore; however, the first character in name must be a letter.
CALL extensions	Permit interfacing to VAX/VMS system service procedures using the VAX-11 calling standards.
Hexadecimal and octal constants and field descriptors	Both octal and hexadecimal constants can be expressed in DATA statements. No conversion of the defined value (such as sign-extension) is performed. The Z field descriptor in FORMAT statements enables a program to read and write hexadecimal digits which are stored in an internal format in an I/O list element.
DO WHILE/END DO	Structured looping control constructs.
Data initialization in type-declaration statements	Variables can be assigned initial values in type declaration statements.
INTEGER data type defaults	A compiler command specification allows all INTEGER and LOGICAL declarations without explicit length specifications to be considered as INTEGER*2 and LOGICAL*2 or INTEGER*4 or LOGICAL*4, respectively.

VAX-11 FORTRAN and PDP-11 FORTRAN IV-PLUS

Additional data types and type declaration statements (DOUBLE COMPLEX, COMPLEX*16, and **CHARACTER*n** are VAX-11 FORTRAN only) NOTE Names appearing on the same line above are synonyms. Those in boldface are the ANSI standard ones.	BYTE, LOGICAL*1, LOGICAL*2, **LOGICAL**, LOGICAL*4, INTEGER*2, **INTEGER**, INTEGER*4, **REAL**, REAL*4, **DOUBLE PRECISION**, REAL*8, **COMPLEX**, COMPLEX*8, DOUBLE COMPLEX, COMPLEX*16, **CHARACTER*n**
Indexed File I/O	Extensions are provided to allow FORTRAN language access to RMS ISAM files.
Keyed READ	Key types: INTEGER*2, INTEGER*4, CHARACTER with generic, and approximate key match.
Indexed File WRITE	New records can be written to ISAM files with the write statements.
REWRITE statement	Existing records in ISAM files can be modified with the REWRITE statement.
DELETE statement	Existing records can be deleted from ISAM or relative files with the DELETE statement.
UNLOCK statement	Single-record locking (in the VAX environment) and bucket-level locking (in the PDP-11 environment) for shared file applications involving relative and indexed organization files.
Logical operations on integers	The logical operators .AND., .OR., .NOT., .XOR., and .EQV. may be applied to integer data to perform bit masking and manipulation.
INCLUDE statement	The INCLUDE statement incorporates FORTRAN source text from a separate file into a FORTRAN program.

VAX-11 FORTRAN, PDP-11 FORTRAN IV-PLUS, and PDP-11 FORTRAN IV

Array subscripts using general expressions of any numeric data type	Any arithmetic expression can be used as an array subscript. If the value of the expression is not an integer, it is converted to integer format.
End-of-Line comments	Any FORTRAN statement can be followed, in the same line, by a comment that begins with an exclamation point.
Conditional compilation of debugging statements	Statements that are included in a program for debugging purposes can be so designated by the letter D in column 1. Those statements are compiled only when the associated compiler command option is set. They are treated as comments otherwise.
Default FORMAT width	The programmer can specify input or output formatting by type and default width and precision values will be supplied.

Table 7-2
FORTRAN-77 Features

VAX-11 FORTRAN	
Additional data types	The data type INTEGER*4 provides a sign plus 31 bits of precision. INTEGER*4 allows a greater range of values to be represented than INTEGER*2. Both data types can be used in the same program.
Additional I/O statements	READ (u'r,fmt) and WRITE (u'r,fmt) provide input and output to direct access files.
DO control variable data types	The control variable of a DO statement can be a REAL or DOUBLE PRECISION variable, as well as an INTEGER*2 or INTEGER*4 variable. The initial, terminal, and increment parameters can be of any data type and are converted before use to the type of the control variable if necessary.
Additional data type	The data type CHARACTER permits manipulation of strings of ASCII characters expressed as constants, variables, arrays, substrings, symbolic names, or functions.
IF THEN ELSE statements	The FORTRAN-77 block-IF statements are provided: IF, ELSE IF, ELSE, and ENDIF. These structured programming statements provide more readable and reliable methods for expressing conditional statement execution.
Standard CALL facility	Provides standard argument definitions for called procedures.
VAX-11 FORTRAN and PDP-11 FORTRAN IV-PLUS	
ENTRY statement	ENTRY statements can be used in SUBROUTINE and FUNCTION subprograms to define multiple entry points in a single program unit.
PARAMETER statement	PARAMETER statements can be used to give symbolic names to constants.
Generic function selection	Function selection by argument data type is provided for many FORTRAN library functions.
Array dimension bounds	Lower bounds as well as upper bounds of the array dimension can be specified in array declarators. The value of the lower bound dimension declarator can be negative, zero or positive.
List-Directed I/O statements	The READ (u,*), WRITE (u,*), TYPE*, ACCEPT*, and PRINT* statements provide list-directed, or "free format," I/O without requiring a FORMAT specification.
Additional I/O statements	OPEN and CLOSE statements provide file control and attribute definition. ACCEPT, TYPE, and PRINT statements provide device-oriented I/O. ENCODE and DECODE statements provide memory-to-memory formatting. DEFINE FILE, READ (u'r), WRITE (u'r), and FIND (u'r) provide unformatted direct access I/O, which allows the FORTRAN programmer to read and write files written in any format.
End-of-file or Error Condition transfer	The specifications END=n and ERR=n (where n is a statement label) can be included in any READ or WRITE statement to transfer control to the specified statement upon detection of an end-of-file or error condition. The ERR=n option is also permitted in the ENCODE and DECODE statements, allowing program control of data format errors.
Additional data type	The byte data type (keyword LOGICAL*1 or BYTE) is useful for storing small integer values as well as for storing and manipulating character information.
IMPLICIT declaration	The IMPLICIT statement has been added to redefine the implied data type of symbolic names.

DO loop iteration count	The terminal and increment parameters can be modified within a DO loop without affecting the iteration count. The number of times a DO loop is executed is determined at the initialization of the DO statement and is not re-evaluated during successive executions of the loop. Consequently, the number of times the loop is executed will not be affected by changing the variables used in the DO statement.

VAX-11 FORTRAN, PDP-11 FORTRAN IV-PLUS, and PDP-11 FORTRAN IV

Array dimensions	Arrays can have up to seven dimensions.
Character literals	Character strings bounded by apostrophes can be used in place of Hollerith constants.
Mixed-mode expressions	Mixed-mode expressions can contain any data type, including complex and byte.
General expression DO and GO TO parameters	General expressions are permitted for the initial value, increment, and limit parameters in the DO statement, and as the control parameter in the computed GO TO statement.
DO increment parameter	The value of the DO statement increment parameter can be negative.
Optional statement label list	The statement label list is an assigned GO TO is optional.
General expressions in I/O lists	General expressions are permitted in I/O lists of WRITE, TYPE, and PRINT statements.

Source Program Libraries

The INCLUDE statement provides a mechanism for writing modular, reliable, and maintainable programs by eliminating duplication of source code. A section of program text that is used by several program units, such as a COMMON block specification, can be created and maintained as a separate source file. All program units that reference the COMMON block then merely INCLUDE this common file. Any changes to the COMMON block will be reflected automatically in all program units after compilation.

Calling External Functions and Procedures

FORTRAN programs can call subroutines written in any other VAX language, and also system services, using the VAX-11 procedure calling standard. Special operators exist for passing arguments by immediate value, by reference, or by descriptor. A special operator also exists for obtaining the location of argument values used by the system services procedures.

Shareable Programs

The FORTRAN language can be used to create shareable programs. FORTRAN subprograms can also be used to create shareable image libraries, which can be available to any program written in a native programming language.

Diagnostic Messages

Diagnostic messages are generated when an error or potential error is detected. Errors detected during compilation are reported by the compiler, and include source program errors, such as misspelled variable names, missing punctuation marks, etc.

Source program diagnostic messages are classified according to severity: F (Fatal), E (Error), or W (Warning). F-class messages indicate errors that must be corrected before compilation can be completed. Object code is not produced. E-class messages indicate that an error was detected that is likely to produce incorrect results; however, an object file is generated. W-class messages are produced when the compiler detects acceptable but nonstandard syntax; or when it corrects a syntactically incorrect statement. The message indicates the existence of possible trouble in executing the program.

Compiler Operations and Optimizations

The VAX-11 FORTRAN compiler accepts sources written in the FORTRAN language and produces an object file which must be linked prior to execution. The compiler generates VAX-11 native machine language code. Figure 7-2 is an illustration of VAX-11 FORTRAN code and its equivalent VAX-11 MACRO code.

During compilation, the compiler performs many code optimizations. The optimizations are designed to produce an object program that executes in less time than an equivalent nonoptimized program. Also, the optimizations are designed to reduce the size of the object program.

The VAX-11 FORTRAN compiler performs the following optimizations:

- Constant folding—constant expressions are evaluated at compile-time.
- Compile-time constant conversion.
- Compile-time evaluation of constant subscript expressions in array calculations.
- Constant pooling—only a single copy of a constant is allocated storage in the compiled program. Constants that can be used as immediate mode operands are not

```
0001            SUBROUTINE RELAX2(EPS)
0002            PARAMETER M=40, N=60
0003            DIMENSION X(0:M,0:N)
0004            COMMON X
0005            LOGICAL DONE
0006    1       DONE = .TRUE.
0007            DO 10 J = 1,N-1
0008            DO 10 I = 1,M-1
0009                    XNEW = ( X(I-1,J)+X(I+1,J)+X(I,J-1)+X(I,J+1) )/4
0010                    IF ( ABS(XNEW-X(I,J)) .GT. EPS ) DONE = .FALSE.
0011    10              X(I,J) = XNEW
0012            IF (.NOT. DONE) GO TO 1
0013            RETURN
0014            END
```



```
                        .TITLE  RELAX2
                        .IDENT  01
0000                    .PSECT  $BLANK
0000    X:
0000                    .PSECT  $CODE
0000    RELAX2::
0000                    .WORD   ^M<IV,R5,R6,R7,R8,R9,R10,R11>
0002                    MOVAL   $LOCAL, R11
                                                        ; 0006
0009    .1:
0009                    MNEGL   #1, DONE (R11)
                                                        ; 0007
000C                    MOVL    #1, R6
000F                    MOVAL   $BLANK, R5
0016    L$1:
                                                        ; 0008
0016                    MOVL    #1, R9
0019                    MULL3   #41, R6, R7
001D    L$2:
                                                        ; 0009
001D                    ADDL3   R9, R7, R10
0021                    ADDF3   X+4(R5)(R10), X-4(R5)[R10], R0
0029                    ADDF2   X-164(R5)[R10], R0
002F                    ADDF2   X+164(R5)[R10], R0
0035                    MULF3   #^X3F80, R0, R8
                                                        ; 0010
003D                    SUBF3   X(R5)[R10], R8, R0
0042                    BICW2   #^X8000, R0
0047                    CMPF    R0, ^EPS(AP)
004B                    BLEQ    L$3
004D                    CLRL    DONE(R11)
004F    L$3:
                                                        ; 0011
004F                    MOVL    R8, X(R5)[R10]
0053                    AOBLEQ  #39, R9, L$2
0057                    AOBLEQ  #59, R6, L$1
005B                    MOVL    R6, J(R11)
005F                    MOVL    R8, XNEW(R11)
0063                    MOVL    R9, I(R11)
                                                        ; 0012
0067                    BLBC    DONE(R11), .1
                                                        ; 0013
006A                    RET
                        .END
```

Page 1 above illustrates, as a VAX-11 FORTRAN subroutine, a relaxation function often found in engineering applications. This particular example is a planar (2-dimensional) function that can be used to obtain the values of a variable at coordinates on a surface, for instance, temperatures distributed across a metal plate. The algorithm illustrated here locates the array element values relative to a given point in the plane.

Page 2 contains the equivalent VAX-11 MACRO assembly code for this VAX-11 FORTRAN subroutine. The line numbers in the comment just to the left of this paragraph refer to the lines in the VAX-11 FORTRAN subroutine listing above. Several VAX-11 FORTRAN compiler optimizations are illustrated, including global and local register assignment, removal of invariant computations from the DO loop, recognition of common subexpressions, branch instruction optimizations, in-line ABS function, and peephole optimization.

*The code for lines 7 and 8 contains the global register assignments for the function. The multiply statement just preceding the code for line 9 is an invariant computation (J*41) removed from the DO loop. DO loop control is provided by the Add One and Branch Less Than or Equal (AOBLEQ) instructions in the code for line 11.*

The code for line 9 evaluates the common subexpression for the computation. The code contains a local register assignment (R10), and uses 2- and 3-operand instructions and context switching ([R10]) to calculate an array element value. The last instruction for line 9 is a peephole optimization that increases execution speed by using a "multiply by .25" in place of the FORTRAN statement's "divide by 4."

Figure 7-2

VAX-11 FORTRAN Program

allocated storage. For example, logical, integer, and small floating point constants are generated as immediate mode or short literal operands wherever possible.

- Argument list merging—if two function or subroutine references have the same arguments, a single copy of the argument list is generated.
- Branch instruction optimizations for arithmetic or logical IF statements.
- Elimination of unreachable code—an optional warning message is issued to mark unreachable statements in the source program listing.
- Recognition and replacement of common subexpressions.
- Removal of invariant computations from DO loops.
- Local register assignment—frequently referenced variables are retained (if possible) in registers to reduce the number of load and store instructions.
- Assignment of frequently used variables and expressions to registers across DO loops.
- Reordering expression evaluation to minimize the number of temporary registers required.
- Delaying negation/not to eliminate unary complement operations.
- Flow-Boolean optimizations.
- Jump/Branch instruction resolution—the Branch instruction is used wherever possible to eliminate unnecessary Jump instructions. This reduces code size.
- Peephole optimizations—the code is examined on an operation-by-operation basis to replace sequences of operations with shorter and faster equivalent operations.

Debugging Facilities

VAX-11 FORTRAN debugging facilities include diagnostic messages, conditional compilation flags, and access to the VAX/VMS DEBUG program. The DEBUG program lets the programmer set breakpoints and trace points, and examine and modify the contents of locations dynamically when executing the program.

DEBUG understands FORTRAN data type representations and syntax. It can examine and deposit locations using floating point representation, and it can reference FORTRAN symbols, statement labels, and line numbers symbolically. It can also reference arrays symbolically, for example:

```
EXAMINE A(I,J+3)
```

When debugging VAX-11 FORTRAN programs, the programmer can disable optimizations that would remove unreferenced statement labels, FORMAT statement labels, and immediately referenced labels. This ensures that all statement labels are available to the debugger.

Conditional Compilation of Statements

During the development stages of a program, it is often useful to establish points in the program at which specified values can be examined to insure that the program is functioning correctly. For example, if the value of a variable is known after the execution of a specified statement, the variable can be printed to verify its contents. Therefore, by including a number of such source lines at strategic points throughout the program, debugging the program is greatly simplified. FORTRAN provides a facility for conditionally compiling such source lines so that they can be compiled during the development stage but treated as comments once the program has been debugged.

Symbolic Traceback

Figure 7-3 illustrates a source VAX-11 FORTRAN program and the symbolic traceback facility supported by VAX/VMS. (Note that some of the entries in the list show relative and absolute PC but no corresponding values for module name and routine name; this indicates that the values refer to procedure calls internal to the run time library.)

```
0001          I=1
0002          CONTINUE
0003          J=2
0004          CONTINUE
0005          K=3
0006          CALL SUB1
0007          CONTINUE
0008          END

0001          SUBROUTINE SUB1
0002          I=1
0003          J=2
0004          CALL SUB2
0005          END

0001          SUBROUTINE SUB2
0002          COMPLEX W
0003          COMPLEX Z
0004          DATAW/(0.,0.)/
0005          Z = LOG(W)
0006          END
%MTH-F-INVARGMAT, invalid argument to math library
     user PC 00000449
%TRACE-F-TRACEBACK, symbolic stack dump follows
```

module name	routine name	line	relative PC	absolute PC
			0000074C	0000074C
			0000081C	0000081C
	SUB2	5	00000011	00000449
	SUB1	4	00000017	00000437
	T1$MAIN	6	0000001B	0000041B

Figure 7-3
FORTRAN Symbolic Traceback

VAX-11 COBOL

Introduction

VAX-11 COBOL is a new, high-performance implementation of COBOL. It is based on American National Standard Programming Language COBOL, X3.23-1974, the industry-wide accepted standard for COBOL. Some features planned for the next COBOL (anticipated in 1981), are also included. VAX-11 COBOL expands and enhances its predecessor, VAX-11 COBOL-74, and includes features that

appeal to a wider range of COBOL users because it allows more complex coding procedures to be accomplished more simply.

It is anticipated that the new ANSI standard will call for greater structured programming. This allows explicit delimiting of statements in the Procedure Division, a feature which can simplify COBOL coding that previously required additional GO TO statements and procedure names. In meeting the requirement for structured programming, the new VAX-11 COBOL includes—among other features—the in-line PERFORM statement, allowing a reduction of program complexity by putting all the logic of the PERFORM in line.

Many features of VAX-11 COBOL make the programmer's job easier, either by simplifying coding procedures or by giving direct access to more VAX/VMS facilities. The COBOL SORT and MERGE verbs are now available in VAX-11 COBOL so that sorting and merging can be performed at the source language level rather than though direct calls to the VAX/VMS utilities. VAX-11 COBOL supports symbolic characters so that the programmer can define non-printable characters simply and can generate video display forms. Further, the REFORMAT utility allows bidirectional conversion of COBOL source programs from easy-to-enter DIGITAL terminal format to ANSI standard format and vice versa.

VAX-11 COBOL is properly defined as an implementation of ANSI COBOL with full support of the following:

- full Level 2 Nucleus Module without the RERUN option in the I-O-CONTROL paragraph
- full Level 2 Table Handling Module
- full Level 2 Sequential I/O Module
- full Level 2 Relative I/O Module
- full Level 2 Indexed I/O Module
- full Level 2 Segmentation Module
- full Level 2 SORT/MERGE Module
- full Level 2 Library Module
- full Level 2 Interprogram Communication Module

Besides the VAX-11 object module, the compiler is capable of producing a machine language listing, a cross reference listing in either alphabetic sequence or order of declaration, and maps of file names, data names, procedure names, and external program names.

General Characteristics

Most of the code in an object module is implemented with in-line VAX-11 instructions. The object code produced by the compiler takes advantage of such native mode features as:

- direct calls to the operating system
- transparent access to DECnet
- direct calls to VAX-11 SORT
- many of the VAX-11 string manipulation instructions
- direct calls to the Common Run Time Library
- direct calls to an external routine (written in a DIGITAL-supported language) that conforms to the VAX-11 Procedure Calling Standard

The object code produced by VAX-11 COBOL uses the VAX/VMS traceback facility for determining the source of run time errors. If a fatal error occurs at run time, an English error message is printed to identify the cause of the error. Additionally, the traceback pinpoints the source of the error to a specific line number in the COBOL source module producing the error. The English error message coupled with the traceback facility gives the user a powerful debugging tool for identifying fatal execution errors.

Object modules produced by the compiler can be linked with native mode object modules produced by other VAX-11 language processors including BASIC, FORTRAN, and MACRO.

Structured Programming

Structured programming adds some of the features of a block-structured language (such as ALGOL) to the new VAX-11 COBOL compiler. Thus, more complex programs can be written in-line without recourse to subroutines. This makes programs easier to write and to read.

The example below shows the READ and IF statements using structured programming. The statements after END-READ are executed regardless of whether the AT END condition occurs. Similarly, the MOVE after END-IF is executed regardless of the value of FILE-END.

```
IF ITEMA = ITEMB
   READ FILE-A AT END
      MOVE 1 TO FILE-END
      CLOSE FILE-A
      END-READ
MOVE ITEMB TO ITEMC
IF FILE-END = 1
      DISPLAY ITEMC
      END-IF
MOVE ITEMD TO ITEME.
```

Several COBOL verbs have structured programming delimiters. Among them are:

```
ADD
CALL
COMPUTE
DELETE
DIVIDE
IF
MULTIPLY
PERFORM
READ
RETURN
REWRITE
SEARCH
START
STRING
SUBTRACT
UNSTRING
WRITE
```

Particularly, the PERFORM verb has been enhanced. The resultant in-line PERFORM capability is similar to DO WHILE and DO UNTIL in other high-level languages.

In this example, if the first occurrence of ITEMB is not equal to "X": (1) the in-line PERFORM statements are executed, moving an "X" to the first 10 occurrences of ITEMB;

then, (2) the message is displayed.

```
IF ITEMB (1) NOT = "X"
   PERFORM
      VARYING ITEMA FROM 1 BY 1
         UNTIL ITEMA > 10
      MOVE "X" TO ITEMB (ITEMA)
   END-PERFORM
   DISPLAY "ARRAY INITIALIZED"
```

Data Types

VAX-11 COBOL increases the number of data types available to the COBOL programmer, including floating point and double floating point. The standard data types are:

- Numeric DISPLAY Data
  - Trailing overpunch sign
  - Leading overpunch sign
  - Trailing separate sign
  - Leading separate sign
  - Unsigned
  - Numeric-edited
- Numeric COMPUTATIONAL Data
  - Word fixed binary
  - Longword fixed binary
  - Quadword fixed binary
- Packed-Decimal Data (COMPUTATIONAL-3)
  - Unsigned packed decimal
  - Signed packed decimal
- Floating Point Data
  - F_floating (COMPUTATIONAL-1)
  - D_floating (COMPUTATIONAL-2)
- Alphanumeric DISPLAY Data
  - Alphanumeric
  - Alphabetic
  - Alphanumeric-edited

As indicated previously, VAX-11 COBOL supports the COMP-3 (packed decimal) data type (two decimal digits per byte). This data type offers the following advantages:

- disk storage savings
- faster arithmetic operations than standard numeric display data type
- compatibility with and migration from other COBOL vendors

Figure 7-4 illustrates a record definition of a typical payroll master file application in which the COMP-3 data type is frequently used. In this record definition, all numeric fields on which arithmetic operations are performed are defined to be the COMP-3 data type.

Figure 7-5 illustrates a sample calculation of one such COMP-3 data item in the record. Here, the year-to-date net pay is calculated as a function of the gross pay, and all voluntary and involuntary deductions to date.

Infrequently, commercial applications arise in which the utilization of floating point data (COMP-1 and COMP-2) is useful. For example, a large corporation may want to survey its customers regarding its product quality. The corporation wishes to select a statistically valid sample of its customer base without going to the expense of contacting each and every customer. Hence, it is necessary to randomly sample its customer base; a random number generator is used to select those customers to be sampled.

Figure 7-6 illustrates a COBOL program fragment in which a CALL to the VAX-11 run-time procedure library routine MTH$RANDOM is made to generate random numbers. This routine returns a random number in COMP-1 (F_floating) data type representation in the range from 0.0 to 1.0. Such numbers are then integerized and subsequently used to select those customers to be sampled in the product quality survey.

The COMP-2 data type may be used in similar commercial applications.

Files and Records

VAX-11 COBOL's Sequential I/O, Relative I/O, and Indexed I/O modules meet the full ANSI Level 2 standard. The language's Level 2 Indexed I/O module statements enable VAX-11 COBOL programs to use the VAX-11 RMS multikey indexed record management services to process files. These files can be accessed sequentially, randomly,

```
FD      PAYROLL-MASTER
          LABEL RECORDS ARE STANDARD.
01      PAYROLL-REC.
          02  EMPLOYEE-NAME                PIC X(30).
          02  EMPLOYEE-ID                  PIC 9(9)        USAGE IS DISPLAY.
          02  YTD-GROSS-PAY                PIC 9(5)V99     USAGE IS COMP-3.
          02  YTD-FED-WITHHOLD-TAX         PIC 9(5) V99    USAGE IS COMP-3.
          02  YTD-FICA                     PIC 9(4)V99     USAGE IS COMP-3.
          02  YTD-STATE-WITHHOLD-TAX       PIC 9(5)V99     USAGE IS COMP-3.
          02  YTD-LOCAL-WITHHOLD-TAX       PIC 9(5)V99     USAGE IS COMP-3.
          02  YTD-VOLUNTARY-DEDUCTIONS     PIC 9(4)V99     USAGE IS COMP-3.
          02  YTD-NET-PAY                  PIC 9(5)V99     USAGE IS COMP-3.
```

Figure 7-4
COMP-3 Record Definition

```
•
•
•
SUBTRACT YTD-FED-WITHHOLD-TAX,
         YTD-FICA
         YTD-STATE-WITHHOLD-TAX,
         YTD-LOCAL-WITHHOLD-TAX,
         YTD-VOLUNTARY-DEDUCTIONS
         FROM YTD-GROSS-PAY
         GIVING YTD-NET-PAY.
•
•
•
```

Figure 7-5
Arithmetic on COMP-3 Data Type

or dynamically using one or more indexed keys to select records. The RESERVE AREAS clause enables the user to specify the number of I/O buffers for fast multikey processing. The APPLY clause allows the user to specify file processing optimization attributes for fast record access.

VAX-11 COBOL has full variable-length record capability. This is an improvement over VAX-11 COBOL-74, in which variable-length records were only partially supported.

Reference modification—the ability to refer to parts of defined fields without redefining them—has also been included in VAX-11 COBOL.

The language includes a facility to manipulate data strings. The INSPECT verb allows the user to search for embedded character strings, tallying and/or replacing the occurrences of such strings. Additionally, the STRING and UNSTRING verbs permit the user to join together and break out separate strings with various delimiters.

SORT/MERGE Facility

The VAX-11 COBOL SORT/MERGE module meets the full ANSI standard and permits performing sort and merge operations at the COBOL source language level without requiring the programmer to understand the VAX-11 SORT interface. The COBOL SORT/MERGE capability includes sorting and/or merging one or more files in the same source module, specifying one or more sort/merge key(s) (in ascending or descending order) for each file, and the option to use either standard or user-specified input/output procedures.

Figure 7-7 illustrates how to sort a file with the USING and GIVING phrases of the SORT statement. The fields to be sorted are S-KEY-1 and S-KEY-2; they contain account numbers and amounts. The sort sequence is amount within account number. Notice that OUTPUT-FILE is a relative file.

In Appendix B, the sample program is merging three identically sequenced regional sales files into one total sales file. The program adds sales amounts and writes one record for each product-code.

Symbolic Characters Facility

It is often useful for the programmer to be able to construct on a video terminal, the image of a form similar to a printed form. This process involves imbedded or non-printing characters (i.e., line feed, carriage return, escape key, etc.). VAX-11 COBOL provides the user with the ability to include, within the COBOL code, non-printing control characters. Essentially, these characters control the position of the cursor during an interactive session utilizing a video terminal (i.e., VT52, VT100, etc.).

Figure 7-8 illustrates a sample data entry form used as a prompt for data input.

The VAX-11 COBOL code used to generate this particular form is listed in Appendix C. The sample code is used in conjunction with the VT100 terminal. Code for the VT52 is similar.

```
01      RAND-NUM                                    USAGE IS COMP-1.
01      RAND-CUST-NUM               PIC 9(7).
01      CUST-NUM REDEFINES RAND-CUST-NUM.
        02 DISTRICT                 PIC 9(2).
        02 WITHIN-DIST              PIC 9(5).
01      SEED                        PIC 9(8)        USAGE IS COMP.
•
•
•
            CALL "MTH$RANDOM"           USING SEED
                                        GIVING RAND-NUM.
            COMPUTE RAND-CUST-NUM ROUNDED = RAND-NUM * 10000000.
•
•
•
```

Figure 7-6
Example of COMP-1 Data Type

```
IDENTIFICATION DIVISION.
PROGRAM-ID.        SORT EXAMPLE.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.                    VAX-11.
OBJECT-COMPUTER.                    VAX-11.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
        SELECT INPUT-FILE ASSIGN TO "INPFIL".
        SELECT OUTPUT-FILE ASSIGN TO "OUTFIL"
                        ORGANIZATION IS RELATIVE.
        SELECT SORT-FILE ASSIGN TO "SRIFIL".
DATA DIVISION.
FILE SECTION.
SD      SORT-FILE.
01      SORT-REC.
        03  S-KEY-1.
                        05  S-ACCOUNT-NUM    PIC X(8).

        03  FILLER    PIC X(32).
        03  S-KEY-2.
                        05  S-AMOUNT         PIC S9(5)V99.
        03  FILLER    PIC X(53).
FD      INPUT-FILE
        LABEL RECORDS ARE STANDARD.
01      IN-REC         PIC X(100).
FD      OUTPUT-FILE
        LABEL RECORDS ARE STANDARD. PIC X(100).
01      OUT-REC        PIC X(100).
PROCEDURE DIVISION.
000-00-THE-SORT.
                SORT SORT-FILE ON ASCENDING KEY
                                        S-KEY-1
                                        S-KEY-2
                WITH DUPLICATES IN ORDER
                USING INPUT-FILE GIVING OUTPUT-FILE.
        DISPLAY "END OF PROGRAM SORT EXAMPLE".
        STOP RUN.
```

Figure 7-7
Sample SORT Code

```
CUSTOMER NUMBER:12345678
CUSTOMER NAME:ROLAND J. JONES-----
CUSTOMER ADDRESS:747 FIRST AVE.-----
CITY:ANYTOWN------STATE:NH ZIP:03061
```

Figure 7-8
Video Form

CALL Facility

The CALL statement enables a COBOL programmer to execute routines that are external to the source module in which the CALL statement appears. The VAX-11 COBOL compiler produces an object module from a single source module. The object module file can be linked with other VAX-11 object modules, so as to produce an executable image. Thus, COBOL programs can call external routines written in other VAX-11 supported languages including BASIC, FORTRAN, and MACRO.

The CALL statement facility has been extended by allowing the user to pass arguments BY REFERENCE (the default in COBOL), BY DESCRIPTOR, and BY VALUE. These argument-passing mechanisms conform to the VAX-11 Procedure Calling Standard and allow COBOL programs to call VAX/VMS operating system service routines. Also, a COBOL program can receive a returned status value from the routine it calls via the GIVING clause associated with the extended CALL facility. Such an extended CALL facility gives the user access to operating system specific facilities and Common Run Time facilities. Figure 7-9 illustrates a sample program utilizing all three types of argument passing mechanisms.

In this program the system service routine $ASCTIM is called, which converts binary time to an ASCII string representation. In this example, the buffer length as specified by "timbuf" plus the value of the item "dummy" determine the type of information which the service routine will return to the COBOL program (e.g., specifying a length of 24 plus values of 0 in the following two arguments will cause both current date and time to be returned; if a length of 11 had been specified, then only the date would be returned).

Source Library Facility

VAX-11 COBOL supports the full ANSI COBOL Library facility. All frequently used data descriptions and program text sections can be stored in library files available to all programs. These files can then be copied into source programs performing textual substitution (i.e., replacement) in the process. This capability reduces program preparation time and eliminates a common source of error during program development.

Shareable Programs

The COBOL language can be used to create shareable programs. VAX-11 COBOL subprograms can be placed in shareable image libraries created by the linker, which then can be made available to any program written in a native programming language.

Debugging COBOL Programs

The VAX-11 COBOL compiler produces source language listings with embedded diagnostics indicating line and position of error. Fully descriptive diagnostic messages are listed at the point of error. Many error conditions are checked at compile time, varying from simple informational indications to severe error detections. At the user's option, the compiler can also produce a machine language listing, a file name map, a data name map, a procedure name map, an external program name map, and a cross reference listing.

When a fatal error occurs at run time, an error message identifying the cause of the error is displayed to the user. Additionally, the traceback system facility prints the sequence of routine invocations active at the time of the fatal error. For each routine invocation, traceback displays the

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CALLTST2.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.   VAX-11.
OBJECT-COMPUTER.   VAX-11.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 TIMLEN                          PIC 9(4) USAGE IS COMP VALUE IS 0.
01 D-TIMLEN                        PIC 9(4) VALUE IS 9999.
01 TIMBUF                          PIC X(24) VALUE IS SPACES.
01 RETURN-VALUE                    PIC 9(9)USAGE IS COMP
                                   VALUE IS 999999999.

01 D-RETURN-VALUE                  PIC 9(9) VALUE IS 999999999.
PROCEDURE DIVISION.
PO.

DISPLAY "CALL SYS$ASCTIM".
CALL "SYS$ASCTIM"
        USING
                BY REFERENCE TIMLEN
                BY DESCRIPTOR TIMBUF
                BY VALUE ZERO
                BY VALUE ZERO
        GIVING
                RETURN-VALUE.
DISPLAY "DATE/TIME= " TIMBUF.
MOVE TIMLEN TO D-TIMLEN.
DISPLAY "LENGTH OF RETURNED = " D-TIMLEN.
MOVE RETURN-VALUE TO D-RETURN-VALUE.
DISPLAY "RETURN-VALUE = " D-RETURN-VALUE.
STOP RUN.
```

Figure 7-9
System Services Call

module name, routine name, and source line number in which either an invocation to another user routine occurs or the fatal error itself occurs.

As an example of the traceback facility, Figure 7-10 illustrates the printing of error messages and the subsequent traceback for a COBOL module in which an I/O error occurs at run time. Specifically, a COBOL OPEN statement failed because the file "DB2:[COBOL]MASTERFIL.DAT" was not found on the OPEN operation. The "module name" and "routine name" fields (of the traceback) identify the entry point, IOERRTEST, into the COBOL module. The OPEN failure occurs on line number 22 of the source module. The "relative PC" field specifies that the OPEN failure corrrspondingly occurs at "67" hexadecimal bytes into the object code relative to the entry point IOERRTEST. The "absolute PC" field also specifies that the OPEN failure occurs at absolute location "667" in the executable image containing IOERRTEST.

Additionally, the user can request a complete explanation of the OPEN error by interrogating the system interactively with the command "HELP ERRORS COB FILNOTFOU". This VAX/VMS command displays the information shown in Figure 7-11.

Thus, the issuance of specific, English-like error messages coupled with the traceback facility and interactive interrogation of the system to explain completely the run-time error offers the user a powerful debugging tool in identifying programming errors.

Also, the VAX-11 COBOL debugging facilities provide access to the VAX/VMS SYMBOLIC DEBUGGER. The SYMBOLIC DEBUGGER lets the programmer set breakpoints, and examine and modify the contents of locations dynamically while the COBOL program is executing.

Source Translator Utility

The source translator utility is helpful to those users migrating from PDP-11 COBOL and VAX-11 COBOL-74 to the VAX-11 COBOL compiler. This utility produces a translated source program and a listing with flags indicating those language elements which could not be mechanically translated and which therefore require further investigation by the programmer.

Some of the differences between VAX-11 COBOL and PDP-11 COBOL or VAX-11 COBOL-74 that require such a translator are:

- some changes in file status codes
- different specification for the storage of intermediate results

```
%COB-F-FILNOTFOU, file _D32:[COBOL]MASTERFIL.DAT; not found on OPEN
-RMS-E-FNF, file not found
%TRACE-F-TRACEBACK, symbol stack dump follows
```

module name	routine name	line	relative PC	absolute PC
IOERRTEST	IOERRTEST	22	00000067	00000667

Figure 7-10
Example of Traceback Facility

- different methods of specifying file optimization attributes

Fortunately, most differences are transparent to the programmer, and moving programs from PDP-11 COBOL or VAX-11 COBOL-74 requires little (in some cases, no) programmer work.

Source Program Formats

The VAX-11 COBOL compiler accepts source programs that are coded using either the ANSI standard (conventional) format or a shorter, easy-to-enter DIGITAL terminal format. Terminal format is designed for use with the interactive text editors. It eliminates the line number and identification fields and allows the user to enter horizontal tab characters and short text lines.

The REFORMAT utility reads COBOL source programs that are coded using DIGITAL terminal format and converts the source statements to the ANSI standard format accepted by other COBOL compilers throughout the industry. It also has the inverse option to accept programs written in ANSI standard format and to convert the source statements to DIGITAL terminal format. This offers the advantage of saving disk space and compile-time processing when a user is initially migrating from a non-DIGITAL COBOL system to VAX-11 COBOL.

```
ERRORS
    COB
        FILNOTFOU
            file not found on OPEN
            Explanation: The named file was not
            found during the execution of the open
            statement. The file status variable, if pre-
            sent, has been set to 97. No applicable
            USE procedure has been found.
            User Action: The user should examine
            the referenced directory to check for the
            existence of the named file. Another
            common source of this error is a mistake
            in spelling the file specification for the
            file.
```

Figure 7-11
Interactive Explanation of Error

Additional Features

Some additional features of the VAX-11 COBOL compiler are:

- Subscripts can be arithmetic expressions.
- Subscripting and indexing are interchangeable.
- The CONTINUE statement is included. It transfers control to the next executable statement and can replace conditional or imperative statements.
- The AUTHOR, INSTALLATION, DATE-WRITTEN, DATE-COMPILED, and SECURITY paragraphs are included.
- INITIAL clause on the Program-ID is included.
- User-defined alphabets are included.
- PADDING CHARACTER is supported in the FILE-CONTROL paragraph.
- VALUE OF clause is included.
- Delimited scope statements are included (e.g., END-ADD, END-IF).
- All arithmetic statements with overlapping operands function as if the operands did not overlap except for operands specified in LINKAGE SECTION or as EXTERNAL.
- ALTER statement is included.
- CALL data-name is included. Both ON OVERFLOW and EXCEPTION are supported.
- CANCEL statement is fully implemented.
- INITIALIZE statement is fully implemented.
- INSPECT statement is fully implemented including combined TALLYING and REPLACING format.
- SET statement supporting mnemonic-names and condition-names is included.
- Independent segments (segments 50 and above) of the SEGMENTATION module are included.
- WRITE advancing mnemonic-name and associated SPECIAL NAMES C01 is included.
- Use of source file libraries by the COPY statement is included.

This powerful, flexible, and easy-to-use compiler is layered with the VAX/VMS operating system and is available to those customers who require COBOL with VAX/VMS, V2.0.

Sample VAX-11 COBOL Code

This sample VAX-11 COBOL code demonstrates some of

the powerful language elements of VAX-11 COBOL. It illustrates an interactive COBOL program which will generate various types of reports depending upon user specified options. The program operates on an indexed information file via the dynamic access mode. Illustrated are three major COBOL verbs: ACCEPT, DISPLAY and INSPECT.

In Figure 7-12, the program describes the file organization and the access mode. Also described are the primary and alternate keys used for accessing the file randomly.

```
INPUT-OUTPUT SECTION.

FILE CONTROL.
        SELECT CUSTOMER-FILE
           ASSIGN TO "CUSTOM.DAT"
           ORGANIZATION IS INDEXED
           ACCESS MODE IS DYNAMIC
           RECORD KEY IS CUST-CUST-NUMBER
           ALTERNATE RECORD KEY IS
               CUST-CUSTOMER-NAME
           FILE STATUS IS CUSTOMER-FILE-STATUS.
        SELECT STATEMENT-REPORT
           ASSIGN TO "STATEM.REP"
           FILE STATUS IS
               STATEMENT-REPORT-STATUS.
```

Figure 7-12
File Description

In Figure 7-13, using the DISPLAY verb, the interactive COBOL program requests the user to specify an options selection. The user response is then transmitted to the program via the ACCEPT verb. The program uses the INSPECT verb to check that a valid response has been received.

```
DISPLAY "ENTER OPTIONS:".
DISPLAY "S = Print statements".
DISPLAY " I = Print invoices".
DISPLAY "CA = Mail all catalogs".
DISPLAY "CO = Mail selective catalogs".
DISPLAY "CL = Credit limit letters".
ACCEPT OPTIONS-AREA.
MOVE ALL ZERO TO OPTION-STORAGE.
IF OPTIONS-AREA = SPACES
            DISPLAY "Discrepancy Report Only"
            GO TO CONFIRM-OPTIONS.
MOVE 0 TO A-COUNT.
INSPECT   OPTIONS-AREA TALLYING
            OPTION-ENTRY (1) FOR ALL "S"
            OPTION-ENTRY (2) FOR ALL "I"
            OPTION-ENTRY (3) FOR ALL "CA"
            OPTION-ENTRY (4) FOR ALL "CO"
            OPTION-ENTRY (5) FOR ALL "CL".
```

Figure 7-13
Procedure Division Using Interactive COBOL Verbs

```
IF OPTION-STORAGE = ALL ZERO
            DISPLAY "No options recognized"
            STOP RUN.
DISPLAY "Selected options:".
IF WANT-STATEMENTS
            DISPLAY "Statements".
IF WANT-INVOICES
DISPLAY "Invoices"
```

Figure 7-13 (Con't)
Procedure Division Using Interactive COBOL Verbs

Figure 7-14 illustrates the dynamic access method, i.e., shift from random to sequential access. The user moves zero to the primary record key, searches the file randomly, and commences sequential processing at the first non-zero number.

```
OPEN INPUT CUSTOMER-FILE.
MOVE "000000" TO CUST-CUST-NUMBER.
START CUSTOMER-FILE
                    KEY IS > CUST-CUST-NUMBER.
OPEN OUTPUT STATEMENT-REPORT.

*****************************

MAINLINE SECTION
SBEGIN.
        READ CUSTOMER-FILE NEXT
                AT END
                  GO TO END-PROCESS.
        ADD 1 TO RECORD-COUNT
*
*                 Print statement if required
*
```

Figure 7-14
Random to Sequential Access

VAX-11 BASIC

Introduction

The new BASIC product gives the VAX user all the benefits of a highly interactive programming environment and high-performance development language. It combines the best features of PDP-11 BASIC-PLUS-2 and RSTS/E BASIC-PLUS with the significant performance and addressing benefits provided by a native mode VAX language that is fully integrated with the VMS environment.

VAX-11 BASIC is a highly extended implementation language. It provides powerful mathematic and string handling facilities, support for symbolic characters, and full RMS indexed, sequential, and relative I/O operations. There does not yet exist an ANSI standard comparable to this level of BASIC.

VAX-11 BASIC can be used as though it were either an interpreter or a compiler. A fast RUN command and support

for direct execution of unnumbered statements (immediate mode) gives the VAX-11 BASIC user the "feel" of an interpreter. However, this product can also be used in a compilation mode, where it generates native-mode object modules like the other VAX compilers. In either case, VAX-11 BASIC generates optimized VAX native-mode instructions which have extremely fast execution times. Typical compilation speeds are up to 3,000 lines per minute and computations will generally execute up to five times faster than the same programs on a PDP-11.

Following is a brief overview of the general characteristics of VAX-11 BASIC.

General Characteristics

VAX-11 BASIC generates in-line native VAX-11 instructions in both its RUN and its compilation modes. The code produced takes advantage of VAX/VMS native mode capabilities, including:

- direct calls on operating system service routines, even in immediate mode
- transparent access to DECnet
- direct calls to the Common Run Time Library and standard system utilities, including VAX-11 SORT/MERGE
- direct calls to separately compiled native mode procedures written in any language that utilizes the VAX procedure calling standard
- program sizes up to 2 billion bytes are allowed
- all modules are position-independent (PIC) and can be run as fully re-entrant code
- the VAX-11 DEBUG facility has full support for VAX-11 BASIC

The code generated by VAX-11 BASIC uses the standard VAX/VMS traceback facility for determining the source of run-time errors. If a fatal program error should occur, an English message is printed identifying the module and line number where the error occurred. The English text, the traceback, and the integrated BASIC HELP utility provide a powerful program debugging environment.

Object modules produced by VAX-11 BASIC can be linked with native mode modules produced by other language processors including BLISS, COBOL, FORTRAN, PASCAL, and MACRO.

Structured Programming

Structured programming adds some of the features of a block structured language (such as PASCAL) to BASIC to allow complex programs to be written without recourse to subroutines or obscure programming techniques. This makes programs easier to write and maintain.

Figure 7-15 below illustrates a record defined by a MAP statement, successive retrievals by the use of a GET statement, and iteration controlled by a WHILE...NEXT statement block.

The SUBPROGRAM and FUNCTION constructs in VAX-11 BASIC have structured END and EXIT statements. In addition, BASIC allows the use of statement modifiers which allow conditional or repetetive execution of the statement without requiring the user to construct artificial loops or block constructs. Any non-declarative statement in VAX-11 BASIC can have one or more statement modifiers. The BASIC statement modifiers include FOR, IF, UNLESS, UNTIL, and WHILE constructs. Each of the statements in Figure 7-16 illustrates the use of a statement modifier.

Data Types

VAX-11 BASIC increases the number of data types available to the BASIC programmer by allowing the use of 32-bit integer and 64-bit floating point data values. Tabel 7-3 below describes the data type supported by VAX-11 BASIC.

Table 7-3 Data Types

Data Type	Meaning
REAL	Specifies that the variable or constant contains floating-point data. The precision depends on the COMPILE command qualifier you use. COMPILE/SINGLE specifies 32-bit floating point numbers; COMPILE/DOUBLE specifies 64-bit floating point numbers.
WORD	Specifies that the variable or constant contains word-length integer data, regardless of the COMPILE command qualifier you use.
LONG	Specifies that the variable or constant contains longword integer data, regardless of the COMPILE command qualifier you use.
INTEGER	Specifies that the variable or constant contains integer data. This data type defaults to the qualifier used at compile-time. If you compile the program with the /WORD qualifier, integers are 16 bits long; with the /LONG qualifier, 32 bits long.
STRING	Specifies that the variable or array contains string data.

Declarations

VAX-11 BASIC allows inplicit declaration of variables. Unless specifically named in a declaration statement, a variable will be declared by its first reference. The PDP-11 BASIC-PLUS-2 convention is to implicitly type a variable or value by the trailing character in its representation, e.g. ABC$ is a STRING variable; XYZ% and 123% are INTEGER; T12, 314159, and 3.14 are implicitly REAL.

Variables can be declared in COMMON, MAP, or DECLARE statements. Both COMMON and MAP statements are used to declare static storage areas—typically I/O records or shared data blocks. If a program has several named common statements with the same name, the common program sections (PSECTs) are stored one after the other. If several MAP statements have the same name, they overlay the same PSECT.

The DECLARE statement is used to explicitly type variables, functions, and constants. Note that the appearance of a variable name in a DECLARE statement implies that

```
99      !   ------------------------------------------------                    &
        !                           EMPLOYEE RECORD DEFINITION(S)               &
        !                                                                       &
        !                           LINE 100: THE "GENERAL DEFINITION"          &
        !                           LINE 200: THE "EXPANDED DEFINITION"         &
        !
100         MAP (REC1)              STRING EMPLOYEE.RECORD = 36,                &
                                    REAL RATE,                                  &
                                    INTEGER ENDFLAG
110     !
200         MAP (REC1)              STRING LAST.NAME = 20,                      &
                                    STRING FIRST.NAME = 12,                     &
                                    STRING MID.INITS = 4,                       &
                                    REAL FILL,                                  &
                                    INTEGER FILL                                &
210     !   ------------------------------------------------!
298     !
299     !
300         FILE.NAME.1$ = "EMPLOYEE.DAT"
310     !
320         OPEN FILE.NAME.1$ AS FILE #1,SEQUENTIAL, ACCESS READ, MAP REC1
330     !
400         TOTAL.RATES = 000000.00
410     !
411     !
490     !   ---------------  COMPUTE SUM OF RATES IN FILE ---------------
498     !
499     !
500         WHILE NOT ENDFLAG
        !
                                    GET #1
                                    TOTAL.RATE = TOTAL.RATE + RATE
        !
            NEXT
510     !
511     !
590     !   ---------------  REPORT CUMULATIVE SUM BELOW  ---------------
591     !
599     !
600         PRINT "TOTAL.RATE: $";TOTAL.RATE
610     !
611     !
690     !   ---------------  REPORT COMPLETED: CLOSE FILE(S)  ---------------
699     !
700         CLOSE #1
900     !
999         END
```

Figure 7-15

Sample Structured Basic Program

```
100         A(I) = A(I) + 1                 FOR     I = 1 TO 100
110     !
200         PRINT SUMMARY.DATA              IF      OPTION.1 AND REPORT="MONTHLY"
210     !
300         PRINT FNHOUSE.PAYMENT           UNTIL   RATE < 123.45
310     !
400         GET #1                          WHILE   EMPLOYEE.NUMBER < 76000
410     !
500         GOSUB 12300                     UNLESS  ERROR.FLAG
510     !
600         PRINT "NORMAL EXIT"             IF      TOTAL > 1000  UNLESS ERRORS > 0
```

Figure 7-16

Statement Modifiers

that variable will not be in static storage (see MAP, COMMON above).

Finally, the EXTERNAL statement is provided to let the BASIC programmer explicitly declare data types for symbols external to the current program unit, e.g. the name of a VMS system service module, an external BASIC function, or an external constant which is to be global in an application.

Figure 7-17 illustrates the use of COMMON, MAP, DECLARE, and EXTERNAL statements.

Files and Records

VAX-11 BASIC supports RMS sequential, indexed, and relative file organization. In addition, BASIC applications can access virtual arrays (stored on files), terminal-format files, and block I/O files via RMS.

The OPEN statement in VAX-11 BASIC allows specification of file organization, access modes, file sharing, record formats, record size, and file allocation. At the record level, a BASIC program can FIND, GET, PUT, UPDATE, DELETE, or RESTORE any record in a file either sequentially or randomly.

VAX-11 BASIC can access files created by other native mode languages, assuming appropriate data representations are maintained with the records.

Symbolic Characters

BASIC now supports references to symbolic characters—those characters in the 96-character ASCII set which do not print, e.g. NUL, SOH, FF, CR, etc. Figure 7-18 illustrates the use of symbolic characters in a BASIC program.

```
100  !  ---------------------  COMMON STATEMENTS  ---------------------
101  !
102      COMMON (DATASET1) REAL A,B,C,D,E,F,G,H,O,P,Q,R,S,T,U,V,W,X,Y,Z,                    &
                           INTEGER      I,J,K,L,M,N                                        &
                           STRING       S1,S2,S3,S4
103  !
104      COMMON (DATASET1) LAST.NAME$=10, FIRST.NAME$=5
105  !
200  !  ----------------------  MAP STATEMENTS  ----------------------
201  !
202      MAP (DATASET2)               REAL         PART.NUMBER, COST,                     &
                                      INTEGER      VENDOR.CODE, QA.INDEX,                 &
                                      STRING       VENDOR.ID=40
203  !
204      MAP (DATASET2)               REAL         FILL,     FILL,                        &
                                      INTEGER      FILL,     FILL,                        &
                                      STRING       VENDOR.NAME = 10, FILL,                &
                                                   VENDOR.TWX = 30
205  !
300  !  -------------------------  DECLARE STATEMENTS  -------------------------
301  !
302      DECLARE                      INTEGER      COUNTER.1, COUNTER.2,                  &
                                      REAL         STANDARD.DEVIATION,                    &
                                      LONG         A.32.BIT.VARIABLE,                     &
                                      WORD         A.16.BIT.VARIABLE,                     &
                                      STRING       LAB.NAME = 20                          &
303  !
304      DECLARE                      INTEGER      CONSTANT     DEBUG.MODE     = 0,       &
                                                                 MY.P = 3,                 &
                                      REAL         CONSTANT     MY.PI          = 3.1416,  &
                                      STRING       FUNCTION     CONCAT
305  !
306      DEF CONCAT( STRING Y, STRING Z)
307              CONCAT = Y + Z
308      FNEND
309  !
310      PRINT CONCAT("THIS IS"," THE RESULT")
311  !
312  !----------------------------  EXTERNAL STATEMENTS  --------------------------
401  !
402  !                                                          CAN BE USED FOR VMS SERVICES
404      EXTERNAL                     INTEGER FUNCTION SYS$ASSIGN
405  !
406      EXTERNAL                     INTEGER FUNCTION SYS$TRNLOG    ! LOGICAL TRANSLATIONS
407  !
408      EXTERNAL                     INTEGER FUNCTION SYS$QIOW      ! SYNCHRONOUS QIO CALL
410  !
500  !-------------------------------------------------------------
```

Figure 7-17

Declaration Statements

```
10          PRINT "PROGRAM STARTS...";LF;LF;"AT "+TIME$(0)
11  !
15          TITLE$ = "SUMMARY REPORT"
19  !
20          PRINT TITLE$;CR; FOR I = 1 TO 5        !     Bold copy
                                                         by overprinting
21  !
30          PRINT
31  !
40          PRINT A(I) FOR I = 1 TO 10             !     Output report data
41  !
50          PRINT
51  !
99          END
Ready
RUN
TEST5                       28-MAY-1980          17:20
PROGRAM STARTS...
                        AT 05:20 PM
SUMMARY REPORT
     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
Ready
```

Figure 7-18

Symbolic Characters

CALL Facility

The CALL statement allows the BASIC programmer to invoke procedures and functions that are external to the current source module. By using the VMS native mode LINK utility, procedures written in any of the VAX native mode languages can be invoked, i.e., BASIC routines can call or be called by procedures written in COBOL, CORAL, FORTRAN, PASCAL, etc.

The CALL statement in VAX-11 BASIC has been extended to allow a procedure to pass parameters BY REFerence, BY VALUE, or BY DESCriptor. These mechanisms conform to the VAX-11 procedure calling standard and allow BASIC programs to call VMS service routines and accept returning status values.

Shareable Programs

Applications written in VAX-11 BASIC can be made shareable images by the VMS LINKER. BASIC now generates fully re-entrant PIC code.

Developing BASIC Programs

VAX-11 BASIC delivers a high-productivity development environment. The key features of this environment include:

- Automatic line number generation via SEQUENCE command.
- Integral line editing with EDIT.
- A RUN command which allows a program to be placed directly into execution without requiring a separate LINK operation.
- Direct execution of unnumbered BASIC statements, allowing quick verification of algorithms, inspection/change of data values, and invocation of subroutines or functions in a halted BASIC program.
- An integral HELP facility helps program debug/development by providing online reference text from the BASIC manual set.
- The VAX-11 BASIC system can produce source language listings with embedded diagnostics indicating the line and position of any errors. Fully descriptive diagnostic messages are provided at the point of an error. Many error conditions are caught at compile time. At the user's option, VAX-11 BASIC can also output a machine language listing and/or a cross-reference listing.
- The VAX/VMS SYMBOLIC DEBUGGER lets the programmer set breakpoints, and inspect or change the value of variables during execution of a program.

Figure 7-19 illustrates the use of several of these features. The text appearing in blue type in Figure 7-19 corresponds to user input, the remaining text is supplied by the BASIC system.

The LOAD Command

A major goal of VAX-11 BASIC is to support a program development *environment*. The LOAD command allows a user to stay in BASIC, even when a program under development involves several separately compiled BASIC subroutines. When a RUN command is issued, any BASIC modules moved into memory by the previous LOAD command are automatically bound together with the module under development and the resulting in-memory image begins execution, i.e., the user is not required to leave BASIC, invoke the LINKER, and use the DCL $RUN command. This speeds program development considerably.

Once an application has been checked out, a final call on the LINKER can be used to create a shareable native mode executable image for production use.

Error Handling

VAX-11 BASIC allows user-directed error and event handling. Occurence of an error can activate one or more routines which handle the error (or event), and then return control to the point where the error occurred (RESUME), or to the calling program (ON ERROR GOBACK), or to the BASIC system itself for standard cleanup and return of control at the BASIC command level.

In determining the cause of an error, the BASIC program can use the value of: ERR—the error message number assigned by BASIC, ERL—the line number where the error occured, ERN$—the name of the BASIC module where the error occurred, and ERT$(ERR)—the error message text which the BASIC system would print if the error were not trapped by the program.

Migration to VAX/VMS

During the VAX-11 BASIC Field Test, numerous sites moved programs from BASIC-PLUS-2 and BASIC-PLUS (on PDP-11 systems) to the VAX native BASIC. A typical site converted literally hundreds of programs and generally had few difficulties. Minor changes were made to BASIC-PLUS-2 programs: the error checking in VAX-11 BA-

```
100     !--------------------INPUT A FILE NAME, COUNT NUMBER OF LINES IN IT----- ----------
110     LINPUT "What file to be opened ";FILE.NAME$
140     F.NAME$ = EDIT$(FILE.NAME$,32%)
160     OPEN F.NAME$ FOR INPUT AS FILE #1
180     ON ERROR GOTO 900
200     LINPUT #1%,TEXT$        FOR I = 1 to 1000000
210     STOP
900     LINE. = ERL
        NUMBER. = ERR
        MESSAGE$ = ERT$(NUMBER.)
        RESUME LINE 910
910     PRINT "*END, FROM LINE";LINE.;"WITH TEXT: ";MESSAGE$;
        PRINT " - AFTER ";I;"RECORDS"
991     STOP
995     PRINT "*** THE END ***"
999     END
Ready
RUNNH
%BASIC-E-SYNERR, syntax error
        at line 900 statement 4
             RESUME LINE 910
                        ↑
Ready
HELP RESUME
RESUME

The RESUME statement marks the end of an error handling routine,
and returns program control to a specified line number.
Format
        RESUME [<lin-num>]
Examples
        990 RESUME 300
        or
        990 RESUME


Ready
LIST 900
TEST6 28-MAY-1980 17:15
900     LINE. = ERL
        NUMBER. = ERR
        MESSGE$ = ERT$(NUMBER.)
        RESUME LINE 910
Ready
EDIT 900 / LINE / /
900     LINE. = ERL
        NUMBER. = ERR
        MESSAGE$ = ERT$(NUMBER)
        RESUME 910
Ready
RUN
TEST6 28-MAY-1980 17:16
What file to be opened          ? TEST6.BAS
*END, FROM LINE 200 WITH TEXT: ?End of file on device - AFTER 17 RECORDS
%BAS-I-STO, Stop
-BAS-I-FROLINMOD, from line 991 in module TEST 6
Ready

PRINT MESSAGE$;" FROM FILE";F.NAME$
?End of file on device FROM FILETEST6.BAS
Ready
PRINT F.NAME$;CR; FOR I = 1 TO 5
TEST6.BAS
Ready
```

Figure 7-19

BASIC Program Development Features

SIC caught actual bugs in many "working" programs. BASIC-PLUS programs were converted to EXTEND mode (or run through the BASIC-PLUS to VAX-11 BASIC translator) and then modified as though they were in BASIC-PLUS-2. Typically, these changes were made:

- the MODE expression on an OPEN statement was changed to the corresponding set of keywords, e.g.,

  OPEN F$ AS FILE #1 MODE2%

  becomes

  OPEN F$ AS FILE #1, ACCESS APPEND
- MAP and DIM statements were moved to occur before OPEN statements
- RSTS/E SYS-CALLS were examined and removed if not supported by VMS

Files were then copied over on tape or by using DECnet, and the programs were RUN under VAX-11 BASIC. In the event errors were detected by BASIC, the online HELP facility was used to determine any additional changes needed for correct compilation.

Certain features were carried forward from PDP-11 BASIC-PLUS and PDP-11 BASIC-PLUS-2 to VAX-11 BASIC in order to make the move to VAX easier. These include:

- BASIC-PLUS to VAX-11 BASIC Translator utility
- Program RESEQUENCE utility from BASIC-PLUS-2 V1.6
- FIELD statement
- CVT, SWAP, and MAGTAPE functions
- Foreign buffer support
- String arithmetic
- Numerous non-privileged RSTS/E SYS calls
- Virtual arrays

Performance

The programs in Figure 7-20 illustrate the level of compute-bound performance possible under VAX-11 BASIC. Program A was taken from page 84 of the March, 1980 issue of BYTE magazine. Program B is very similar and is from page 130 of the June, 1980 issue of Interface Age.

Finally, initial performance tests on VAX-11 BASIC were samples of the "Towers of Hanoi" program and the Whetstone benchmark. These tests show VAX BASIC execution speeds comparable to non-optimized VAX-11 FORTRAN.

Additional Functions

The features listed below complete the promise of a BASIC that leads the competition in virtually every area. This is not an exhaustive list, but does serve to indicate key capabilities of this new product.

- powerful string manipulation functions for creating, converting, searching, editing, and extracting character values
- variable names up to 30 characters long
- maxiumum length of a single string is 65,535 characters
- multiple statements on a line
- multiline IF...THEN...ELSE statements
- optional use of line continuator "&" and statement separator "\", e.g.,

```
100 PRINT          vs.    100   PRINT        &
    PRINT                     \ PRINT        &
    PRINT                     \ PRINT
```

- DCL pass-through in the BASIC command mode by simply prefixing the DCL command line with a dollar-sign, e.g.,

```
Ready
$DIR      *.BAS, *.OBJ
```

- Provision for up to ten individual BASIC object library files for automatic use at RUN time when developing an application using separately-compiled BASIC subroutines.

```
PRIMES          29-MAY-1980 21:08
90      OPEN "T.1" FOR OUTPUT AS FILE #1, RECORDSIZE
132
100     rem     Interface Age's benchmark program to
110     rem     discover the first 1000 prime numbers
120     rem
125     PRINT CHR$(7)
130     t1 = time(1)
140     FOR n = 1 to 1000
150             FOR k = 2 TO 500
160                     m=n/k
170                     I=int(m)
180                     IF I = 0 THEN 230
190                     IF I = I THEN 220
200                     IF m > I THEN 220
210                     IF m = I THEN 240
220             next k
230     PRINT #1, N;
240     NEXT n
250     t2 = time(1)
255     PRINT CHR$(7)
260     PRINT "Elapsed time: ";0.1*(t2-t1);" seconds"
270     END
```

System	CPU	Run-time
TRS-80	Z80	1982 sec
Technico	TI9900	585 sec
DEC-10	PDP-10	65 sec
BASIC-PLUS-2	PDP-11/70	11 sec
VAX-11 BASIC	11/780	2.7 sec

Figure 7-20A
Program A

```
PRIME3      29-MAY-1980 21:09
10    !     PRIME NUMBER PROGRAM #3 OF 3          &
      !                                           &
      !     FROM MARCH 1980 BYTE MAGAZINE         &
      !     PAGE 84                               &
      !     "TRS-80 PERFORMANCE...                &
      !     EVALUATION BY PROGRAM TIMING          &
      !     (INCLUDES 370/148 PL/I AND BAL TIMES  &
      !
15    DECLARE INTEGER M,K
20    OPEN "PRIMES3.TMP" FOR OUTPUT AS FILE #1
30    PRINT "PRIME3 "+TIME$(0)
40    PRINT
45    T1=TIME(1)
50    PRINT #1, 1;2;3;
55    C=0
70    M=3
80    M=M+2
90    FOR K = 3 TO M/2 STEP K-1
100         IF INT(M/K)*K-M = 0 THEN 190
110   NEXT K
121   PRINT #1%, M;
122   C=C+1
190   IF M < 10000 then 80
195   PRINT #1%, "C=";C
196   PRINT "C=";C
199   T2 = TIME(1%)
      P$ = "DONE: "+NUM1$(0.1*(T2-T1))+" CPU SEC"
200   PRINT P$
      PRINT #1, P$
201   END
```

TRS-80 BASIC	Z80	23470 sec
Assembler	Z80	1370 sec
Optimizing PL/I	370/148	79 sec
BAL	370/148	56 sec
VAX-11 BASIC	11/780	58.2 sec

Figure 7-20B
Program B

VAX-11 PL/I

Introduction

VAX-11 PL/I is an extended implementation of the General Purpose Subset (X3.74—"Subset G") of ANSI PL/I, X3.53-1976. VAX-11 PL/I extensions to the subset language are either full language PL/I features included because they were highly desirable, or system-specific extensions intended to provide more complete access to VAX/VMS features. VAX-11 PL/I is a shareable compiler which runs under the VMS operating system and generates highly optimized positon-independent machine code.

All compiler-generated code, with the exception of some built-in functions calls and I/O operations, is inline. Out-of-line operations are performed by the VAX-11 Common Run Time Library. Most high-level language operations are supported directly by VAX hardware instructions.

VAX-11 PL/I supports the VAX Symbolic Debugger.

All VMS system services are available to programs written in PL/I via the CALL statement. Furthermore, VAX-11 PL/I fully supports RMS, the VAX/VMS record management services. A set of ENVIRONMENT options provides access to a large subset of RMS features. All RMS file organizations are supported: sequential, relative, and indexed.

VAX-11 PL/I fully supports the VAX interlanguage calling standard. Routines written in any other native mode language can call PL/I and vice versa. In addition, all VAX/VMS system services and system utilities (Run Time Library, SORT, etc.) are available via the PL/I CALL statement. For system services, a library of predefined ENTRY declarations is provided to minimize the coding required to use these services.

Subset G is a rich language that combines the scientific computing abilities of FORTRAN, the commercial data handling of COBOL, the string manipulation of BASIC, and the block structuring of ALGOL. Selected extensions further enhance these basic capabilities.

The remainder of this section:

- provides an overview of the G Subset
- lists the extension made to the language to provide enhancements for PL/I programs executing in the VAX/VMS operating system environment
- lists features of full PL/I that were excluded from the G Subset but that have been incorporated in the implementation of VAX-11 PL/I
- lists the implementation-defined values that are used in VAX-11 PL/I

THE G (GENERAL-PURPOSE) SUBSET

The G subset of PL/I was designed to be useful in scientific, commercial, and system programming, especially on small and medium-size computer systems. Among the primary goals of the design of the subset were:

- to include features that were easy to learn and to use and to exclude features that were difficult to learn or prone to error
- to provide a subset that would be easily portable from one computer system to another

- to exclude features that were not often used and whose implementation greatly increased the complexity of the run time support required by the compiler

The essential elements of the subset are described below.

Program Structure

The G subset includes a complete character set, with comments, identifiers, decimal arithmetic constants, and simple string constants.

Begin blocks and DO-groups are included in the subset. Each block or group in the program must be terminated with an END statement. For example:

```
ON ENDFILE(INFILE) BEGIN;
   PUT SKIP LIST('End of input file');
   CLOSE FILE(INFILE);
   END;
```

Program Control

The following program control statements are included in the subset: CALL, RETURN, IF, DO, GOTO, null, STOP, ON, REVERT, and SIGNAL.

The DO statement options supported are TO, BY, WHILE, and REPEAT.

An IF statement may contain unlabeled THEN and ELSE clauses. A null statement may be used to specify no action for a given condition. For example:

```
IF A<B THEN; /* no action */
   ELSE PUT LIST('valid');
```

An ON statement may specify a single condition. The condition names supported are ERROR, ENDFILE, ENDPAGE, FIXEDOVERFLOW, KEY, OVERFLOW, UNDEFINEDFILE, UNDERFLOW, and ZERODIVIDE. For example, an attempt to divide by zero can be detected and handled by an ON-unit that specifies ZERODIVIDE:

```
ON ZERODIVIDE BEGIN;
   /* action to be taken
      .
      .
   */
   END;
```

Storage Control

The subset includes the assignment statement and the assignment of array and structure variables whose dimensions and data types match. The subset also permits aggregate promotion, that is, the assignment of a scalar expression to every element or member of an aggregate variable. For example:

```
ARRAY = A1;
```

evaluates the expression A_1 and assigns the result to every element of ARRAY.

The subset also provides the INITIAL attribute, which specifies initial values for variables when they are declared. For example:

```
DECLARE EOF BIT(1) STATIC INITIAL('0'B);
```

declares an "end-of-line" flag named EOF and sets its initial value to '0'B (false). In the subset, only static variables may be initialized.

The ALLOCATE statement with the SET option and the FREE statement are included in the subset. ALLOCATE dynamically allocates storage for a based variable and sets a pointer to the location of the allocated storage. For example:

```
ALLOCATE LIST_ELEMENT SET(LIST_POINTER);
```

allocates storage for the based variable LIST_ELEMENT and sets LIST_POINTER to the location of the allocated storage. The allocated storage can be subsequently released by:

```
FREE LIST_POINTER→LIST_ELEMENT
```

Input/Output

The I/O statements are:

- OPEN and CLOSE.
- READ, WRITE, DELETE and REWRITE for record I/O. Record I/O statements operate on an entire record in a file.
- GET, and PUT, with FILE, STRING, EDIT, LIST, PAGE, SKIP, and LINE options for stream I/O. Stream I/O statements operate on a stream of ASCII input or output data; the stream may be a file of such data or a character-string variable or expression.

The file attributes, specified in DECLARE or OPEN, are DIRECT, ENVIRONMENT, INPUT, KEYED, OUTPUT, PRINT, RECORD, SEQUENTIAL, STREAM, and UPDATE.

The FORMAT statement is included. The format items are E, F, P, A, X, R, PAGE, SKIP, COLUMN, TAB, and LINE. Format items, and the GET EDIT and PUT EDIT statements, provide a formatted I/O capability comparable to that of FORTRAN. For example:

```
PUT EDIT(A) (F(5,2));
```

writes out the value of A as a fixed-point decimal number of up to five digits, two of which are fractional.

Attributes and Pictures

The DECLARE statement is included in the subset. All names must be declared, either by means of a DECLARE statement or by means of a label prefix.

The attributes supported are: ALIGNED, AUTOMATIC, BASED, BINARY, BIT, BUILTIN, CHARACTER, DECIMAL, DEFINED, DIRECT, ENTRY, ENVIRONMENT, EXTERNAL, FILE, FIXED, FLOAT, INITIAL, INPUT, INTERNAL, KEYED, LABEL, OPTIONS, OUTPUT, PICTURE, POINTER, PRINT, RECORD, RETURNS, SEQUENTIAL, STATIC, STREAM, UPDATE, VARIABLE, and VARYING.

The picture characters included are CR, DB, S, V, Z, 9, –, +, $, and *. The picture insertion characters (. , / B) are also included. Pictures allow special characters to be inserted in a fixed-point decimal number. The picture facility (on output) is comparable to the PRINT USING statement of BASIC. For example:

```
PUT EDIT(A) (P '$99999V.99');
```

Writes out the value of A in fixed-point format with five integral digits and two fractional digits, and precedes the number with a dollar sign.

Built-in Functions and Pseudovariables

PL/I provides a set of built-in functions that perform common calculations and data manipulation. A built-in function may be used without declaration, wherever an expression of the same data type is permitted. The built-in

functions in the subset are: ABS, ACOS, ADDR, ASIN, ATAN, ATAND, ATANH, BINARY, BIT, BOOL, CEIL, CHARACTER, COLLATE, COPY, COS, COSD, COSH, DATE, DECIMAL, DIMENSION, DIVIDE, EXP, FIXED, FLOAT, FLOOR, HBOUND, INDEX, LBOUND, LENGTH, LINENO, LOG, LOG2, LOG10, MAX, MIN, MOD, NULL, ONCODE, ONFILE, ONKEY, PAGENO, ROUND, SIGN, SIN, SIND, SINH, SQRT, STRING, SUBSTR, TAN, TAND, TANH, TIME, TRANSLATE, TRUNC, UNSPEC, VALID, and VERIFY.

Subset G also provides pseudovariables that can be used as the targets of assignment statements. The pseudovariables are PAGENO, STRING, SUBSTR, and UNSPEC. For example, whereas the SUBSTR built-in function returns a substring of a specified bit or character string, the SUBSTR pseudovariable allows the assignment of an expression to such a substring.

Expressions

The subset supports all infix and prefix operators, the locator qualifier, parenthesized expressions, subscripts, and function references. Implicit conversion from one data type to another is restricted to those contexts in which the conversion is likely to produce the desired results. For example:

```
DECLARE I DECIMAL;
I = '1.2';
```

converts the character string '1.2' to the decimal equivalent and assigns it to the decimal variable I. However, the following assignment:

```
DECLARE B BIT(8);
B = 'A';
```

is not valid because, to be convertible to a bit string, a character string must consist entirely of 0 and 1 characters.

VAX-11 EXTENSIONS TO THE G SUBSET STANDARD

Procedure-Calling and Condition-Handling Extensions

The following extensions to PL/I were made to allow VAX-11 PL/I procedures to call procedures written in any other programming language that also supports the VAX-11 calling standard.

1. The attributes ANY and VALUE describe how data are to be passed to a called procedure.
2. The VARIABLE option for the ENTRY attribute permits a PL/I procedure to call a non-PL/I procedure with an argument list of variable length. It also permits a procedure to omit arguments in an argument list.
3. The DESCRIPTOR built-in function may be used to pass an argument by descriptor to a non-PL/I procedure.

The following new attributes provide storage classes for PL/I variables. These attributes permit PL/I programs to take advantage of features of the VAX-11 linker and to combine PL/I procedures with other procedures that use these storage classes.

1. The GLOBALDEF and GLOBALREF attributes let you define and access external global variables and optionally to place all external global definitions in the same program section.
2. The READONLY attribute can be applied to a static computational variable whose value does not change.
3. The VALUE attribute defines a variable that is, in effect, a constant whose value is supplied by the linker.

The following extensions to ON condition handling provide support for condition handling in the VAX/VMS environment:

1. The ON statement supports the ANYCONDITION keyword. The ON-unit established by this keyword is executed when any condition occurs for which no explicit ON-unit exists.
2. The ON statement supports programmer-named conditions with the VAXCONDITION keyword.
3. The RESIGNAL built-in subroutine permits an ON-unit to keep a signal active.
4. The ONARGSLIST built-in function provides an ON-unit with access to the mechanism and signal arguments of an exception condition.

Support of VAX-11 Record Management Services

The options of the ENVIRONMENT attributes provide support for many of the features and control values of the VAX-11 Record Management Services (RMS). Additional extensions have been made to the PL/I language to augment this support, as described below.

1. The OPTIONS option is supported on the GET, PUT, READ, WRITE, REWRITE, and DELETE statements. For example:

   ```
   GET LIST(A) OPTIONS(PROMPT('Enter A>'));
   ```

   displays the prompt "Enter A" on the user's terminal.
2. These built-in subroutines provide file handling and control functions: DISPLAY, EXTEND, FLUSH, NXTVOL, REWIND, and SPACEBLOCK.

Miscellaneous Extensions and Deviations

The following list summarizes miscellaneous extensions and deviations.

1. The RANK and BYTE built-in functions are supported, which return the ASCII code for a given character and the ASCII character for a given code, respectively.
2. The expression in a WHILE clause or in an IF statement may be a bit string of any length. When evaluated, the expression results in a true value if any bit of the string expression is a one and in a false value if all bits in the string expression are zeroes.
3. The control variable and the expressions in the TO, BY, and REPEAT options of the DO statement are not restricted to integers and pointers.

FULL PL/I FEATURES SUPPORTED

The items listed below are features that are explicitly excluded from the subset standard but that have been implemented in VAX-11 PL/I. These features all exist in full PL/I.

1. The ENTRY statement is supported. The ENTRY statement provides a means of defining alternate entry points to a procedure. For example, such an alternate entry point can be invoked by a function reference even though the containing procedure is invoked as a subroutine.
2. The ENVIRONMENT option is supported on the CLOSE statement.
3. The picture characters Y, T, I, and R are supported,

and pictures may include iteration factors. These characters provide an additional means of representing zeros in a number (Y) and allow a digit and a sign to be represented by a single character (T, I, R).

4. RETURNS CHARACTER(*) is valid. That is, a function can return a character string whose length is determined by the program.
5. The FINISH condition is supported.
6. A REWRITE statement need not specify the FROM option if the most recent I/O operation on the file was a READ statement with the SET option. (A READ SET statement acquires a record from an input file and sets a pointer to the location of the I/O buffer containing the contents of the record.)
7. The AREA and OFFSET attributes are supported. The AREA attribute allows declaration and manipulation of an entire region of memory (up to 500 million bytes), and the OFFSET attribute declares variables that are offsets into such an area. Offset variables may be used in most contexts in which pointer variables are permitted. Allocation within an area must be controlled by a user-written procedure.
8. The OFFSET and POINTER built-in functions are supported. These functions convert pointer values to offset values, and vice versa.
9. The POSITION attribute is supported. POSITION allows the declaration of a DEFINED variable to specify the position within the base string at which the definition begins.
10. Automatic variables may be initialized. The INITIAL attribute may contain scalar expressions and asterisks with automatic variables. Asterisks indicate that the corresponding variable or element in the declaration is not assigned an initial value.
11. The SET option is optional on the ALLOCATE statement if the allocated variable was declared with BASED(pointer-reference). The ALLOCATE statement then allocates storage for the based variable and sets the referenced pointer variable to the storage location.
12. The character pair /* may be embedded in a comment. This feature permits a statement such as:

```
IF ↑VALID(P) THEN
   PUT LIST('Input error');/* check validity */
```

to be "commented out":

```
/* IF ↑VALID(P) THEN
   PUT LIST('Input error');/* check validity */
```

IMPLEMENTATION-DEFINED VALUES AND FEATURES

1. VAX-11 PL/I supports the full 256-character ASCII character set.
2. The default precisions for arithmetic data are:

   FIXED BINARY (31)
   FIXED DECIMAL (7)
   FLOAT BINARY (24)
   FLOAT DECIMAL (7)

   where each precision is the number of binary or decimal digits, as appropriate.
3. The maximum record size for SEQUENTIAL files is 32,767 bytes minus the length of any fixed-length control area.
4. The maximum key size is 255 bytes for character keys.
5. The default value for the LINESIZE option is as follows. The line size is used by stream I/O statements to determine when to go to a new line.
   - If the output is to a physical record-oriented device, such as a line printer or terminal, the default line size is the width of the device.
   - If the output is to a print file, the default line size is 132.
   - If the output is to a nonrecord device (magnetic tape), the default line size is 510.
6. The default value for the PAGESIZE option is as follows. The page size is used by stream output (PUT) statements to determine when to signal the ENDPAGE condition.
   - If the logical name SYS$LP_LINES is defined, the default page size is the numeric value of SYS$LP_LINES – 6.
   - If SYS$LP_LINES is not defined, or if its value is less than 30 or greater than 90, or if its value is not numeric, the default page is 60.
7. The values for TAB positions are columns beginning with column 1 and every eight columns thereafter: 1, 9, 17, 25, .. 8*i+1, where i is (line size)/8. List-directed output (by the PUT LIST statement) is positioned on tab stops if the output is to a file with the PRINT attribute.
8. The maximum length allowed for a file title is 128 characters. File titles are VAX/VMS file specifications that are associated with PL/I file constants and file variables.
9. The maximum number of digits in editing fixed-point data is 34.
10. The maximum numbers of digits for each combination of base and scale are:

    FIXED BINARY —31
    FIXED DECIMAL —31
    FLOAT BINARY —113
    FLOAT DECIMAL —34
11. The default precision for integer values is 31.
12. The maximum number of arguments that can be passed to an entry point is 253.

VAX-11 PL/I Programming Example

Figure 7-21 illustrates a VAX-11 PL/I program. This program calls a VAX/VMS system service (SYS$TRNLOG) to determine the equivalence string for a logical name.

The program TRNLN calls a VAX/VMS system service, SYS$TRNLOG, to determine the equivalence string for a logical name. SYS$TRNLOG is declared as an external entry constant, with three parameters:

1. A character string of any length, representing the logical name.
2. An integer representing the length of the translated name.
3. A character string of any length representing the translated name itself.

```
/* Translates logical names to equivalent strings */
TRNLN: PROCEDURE OPTIONS (MAIN);
DECLARE SYS$TRNLOG ENTRY(                    /* VAX/VMS system service */
                                             / Parameters: */
  CHARACTER(*),                              /* Logical name */
  FIXED BINARY(15),                          /* Length of translated name */
  CHARACTER(*)                               /* Translated name */
            )
  OPTIONS(VARIABLE)                          /* Some arguments optional */
  RETURNS(FIXED BINARY(31));                 /* SYS$TRNLOG returns integer */
DECLARE INPUT CHARACTER(63) VARYING,                    /* Input name */
  OUTPUT CHARACTERS(63),                     /* Translated name */
  OUTPUT_LEN FIXED BINARY(15),               /* Length of translated name */
  RETURN_STAT FIXED BINARY(31),              /* Return status of SYS$TRNLOG */
  SUCCESS BIT(1)                             /* Successful return */
                    ALIGNED BASED (ADD(RETURN_STAT));

DECLARE SS$_NOTRAN                                      /* Flag for undefined name */
  GLOBALREF FIXED BINARY(31) VALUE;
%REPLACE NOTEND BY '1'B;                    /* "Always true"--to control DO-group */
ON ENDFILE(SYSIN) STOP;                                 /* Stop on CTRL/Z from terminal */
DO WHILE (NOTEND);                          /* Terminated by ON-unit */
  PUT SKIP;
  GET LIST(INPUT)
                    OPTIONS(PROMPT('Enter logical name '));

    /*Invoke sytem service as PL/I function reference: */
    RETURN_STAT = SYS$TRNLOG((INPUT),OUTPUT_LEN,OUTPUT,,,);
    IF RETURN_STAT = SS$_NOTRAN THEN
      PUT SKIP LIST(INPUT::'not defined');
    ELSE IF SUCCESS THEN
      PUT SKIP LIST(INPUT::'is '::SUBSTR(OUTPUT,1,OUTPUT_LEN));
    END;
  END TRNLN;
```

Figure 7-21
Sample VAX-11 PL/I Code

The declaration also states that SYS$TRNLOG returns an integer and can have an argument list of variable length.

Note that the explicit declaration of SYS$TRNLOG is shown here for clarity. VAX-11 PL/I is supplied with a library of predefined entries for VAX/VMS system services, so the declaration of SYS$TRNLOG can be replaced by the single statement:

```
%INCLUDE SYS$TRNLOG;
```

The program also uses a global reference to SS$_NOTRAN; the return value of the system service equals SS$_NOTRAN when there is no defined equivalence for the given logical name.

SYS$TRNLOG actually requires that its first and third arguments be the addresses of character string descriptors. VAX-11 PL/I allows you to pass a character string by descriptor by declaring the corresponding parameter as CHARACTER(*). Such a parameter can have an argument that is a fixed-length character string of any length. If the written argument for such a parameter is a varying-length string, a dummy argument is created by the compiler. To avoid the accompanying warning message, the argument INPUT is enclosed in parentheses.

When a dummy argument is created, the invoked procedure cannot modify the associated parameter. Therefore, the translated name (OUTPUT) is not declared as a varying-length string. Instead, OUTPUT and OUTPUT_LEN both are supplied as arguments to SYS$TRNLOG, which then assigns the necessary values to them. The translated name is written out with a reference to the SUBSTR built-in function:

```
SUBSTR(OUTPUT,1,OUTPUT_LEN);
```

which writes out a substring beginning at character 1 of OUTPUT and continuing for OUTPUT_LEN characters.

A sample session with the program might be:

```
$ DEF LNK$LIBRARY DB1:[PROJECT]MYLIB,OLB<RET>
$ R TRNLN<RET>
```

```
Enter logical name>LNK$LIBRARY<RET>
LNK$LIBRARY is DB1:[PROJECT]MYLIB.OLB<RET>
Enter logical name>GRACE<RET>
GRACE not defined
Enter logical name>↑Z
$
```

VAX-11 PASCAL

Introduction

VAX-11 Pascal, a re-entrant native mode compiler, is an extended implementation of the Pascal language as defined by Jensen and Wirth in the *Pascal User Manual and Report* (1974).

Particularly suited to instructional use, Pascal is also an increasingly popular general purpose language. It implements a well-chosen, compact set of general purpose language features. In addition, portability is easily achievable in programs written in Pascal.

Block structuring and flexible data types make Pascal a good language for commercial users. It is also suitable for systems programming and research applications.

VAX-11 Pascal takes advantage of the VAX-11 hardware floating point, character instruction sets, and virtual memory capabilities of the VAX/VMS operating system. Many of the features common to other languages of VAX/VMS are available through VAX-11 Pascal, including:

- separate compilation of modules
- standard call interface to routines written in other languages
- access to VAX/VMS system services

At compile time, options available to the process include:

- run-time checks for illegal assignment to set and subrange variables, and illegal array subscripts
- cross-reference listing of identifiers
- source program listing
- machine code listing
- generation of some DEBUG and TRACEBACK records for the VAX-11 Symbolic Debugger

Though VAX-11 Pascal has access to the Common Run Time Library routines of VAX/VMS, it also has Pascal-specific Run Time Library routines installed in STARLET.OLB. Such routines primarily provide I/O interfaces to the Record Management Services (RMS).

Standard Pascal provides a modular, systematic approach to computerized problem solving. Major features of the language are:

- INTEGER, REAL, CHAR, BOOLEAN, user-defined, and subrange scalar data types
- ARRAY, RECORD, SET, and FILE structured data types
- Constant identifier definition
- FOR, REPEAT, and WHILE loop control statements
- CASE and IF-THEN-ELSE conditional statements
- BEGIN...END compound statement
- GOTO statement
- GET, PUT, READ, WRITE, READLN, and WRITELN I/O procedures
- Standard functions and procedures

In addition, VAX-11 Pascal incorporates the following extensions to standard Pascal, some of which are common in other Pascal implementations:

- Lexical
  - Upper- and lowercase letters treated identically except in character and string constants
  - New reserved words: MODULE, OTHERWISE, SEQUENTIAL, VALUE, %DESCR, %IMMED, %INCLUDE, and %STDESCR
  - The exponentiation operator, **
  - Hexadecimal and octal constants
  - DOUBLE constants
  - $ and _ characters in identifiers
- Predefined data types
  - DOUBLE
  - SINGLE
- Predefined procedures
  - CLOSE (f)
  - FIND (f,n)
  - OPEN (f,...)
  - DATE (a)
  - HALT
  - LINELIMIT (f,n)
  - TIME (a)
- Predefined functions
  - LOWER (a,n)
  - SNGL (d)
  - UPPER (a,n)
  - EXPO (r)
  - CARD (s)
  - CLOCK
  - UNDEFINED (r)
- Extensions to procedures READ and WRITE
  - READ (or READLN) of user-defined scalar type
  - READ (or READLN) of string
  - WRITE (or WRITELN) of user-defined scalar type
  - WRITE (or WRITELN) of any data using hexadecimal or octal format
- %INCLUDE directive
- VALUE initialization
- OTHERWISE clause in CASE statement
- External procedure and function declarations
- Dynamic array parameters
- Extended parameter specifications
  - %DESCR
  - %IMMED
  - %IMMED PROCEDURE and %IMMED FUNCTION
  - %STDESCR
- Separate compilation of procedures and functions. (A separate compilation unit is termed a MODULE and several routines may be part of a MODULE. Each MODULE is eventually embedded in a host or main program.)

The OPEN, CLOSE and FIND procedures extend the I/O capabilities of the VAX-11 Pascal language. The OPEN procedure can contain file attributes that define the creation or subsequent processing of the file. A FIND procedure is another extension to the language for direct access to sequential files of fixed length records. The stan-

dard I/O procedures of GET, PUT, READ, WRITE, READLN and WRITELN are also available in VAX-11 Pascal.

The extended parameter specifications %DESCR, %IMMED, and %STDESCR are added to the Pascal language to denote the method of argument passing when calling a system service, procedure, or function not written in VAX-11 Pascal (for example, in VAX-11 FORTRAN or MACRO.)

Sample VAX-11 Pascal Code

Figure 7-22 illustrates a VAX-11 Pascal program which a user may write to calculate the hypotenuse of a right triangle. The inputs to the program are the length of the two sides of the triangle and the output is the length of the hypotenuse. This version of the program has not yet been debugged to illustrate how error messages are reported.

Compiler Listing Format

Figure 7-22 contains the compiler listing of the hypotenuse program. The compiler listing contains the following three sections:

- Source code listing—When the LIST qualifier is specified, the source code is listed by default.
- Machine code listing—To generate the machine code listing, the MACHINE_CODE qualifier must be specified.
- Cross-reference listing—To generate cross references for all identifiers used in the program, the CROSS_REFERENCE qualifier must be specified.

The following sections describe the format of the user requested listings. Note that the numbers in these sections are keyed to the circled numbers in the listings of Figure 7-22.

Title Line — Each page of the listing contains a title line. The title line lists the module name (1), the date and time of the compilation (2), the Pascal compiler name and version number (3), and the listing page number (4).

Source Code Listing

Each page of the source code listing contains a line under the title line specifying the date and time of source file creation (5) and the VAX/VMS file specification of the source file (6).

```
EXAMPLE        23-MAY-1980    23:00:05        VAX-11 PASCAL VERSION V1.1-29        PAGE 1
01             11-OCT-1979    11:34:47        _DB1:[200,200]EXAMPLE.PAS;4 (1)

     LINE      LEVEL
   NUMBERS   PROC  STMT    STATEMENT

      100    1     1       program EXAMPLE(INPUT,OUTPUT);
      200    2     1
      300    3     1       label 10;
      400    4     1       var A,B,C:REAL;
      500    5     1
      600    6     1       begin
      700    7     1
      800    8           0  repeat
      900    9           2      WRITELN('Enter triangle sides');
     1000   10           2      if EOLN(INPUT) then goto 10;
     1100   11           2      READLN(A,B);
     1200   12           2      C := (SQR(A) + SQR(B)) ** 0.5;
%PAS-W-DIAGN                                  ↑450          ***     12 ==>     0
                *** WARNING 450: nonstandard Pascal: Exponentiation
     1300   13           2      WRITELN('Hypotenuse is: ',C);
     1400   14           2    until FALSE;
     1500   15           1
     1600   16           1    10:
     1700   17           1    WRITELN(' Done');
%PAS-F-DIAGN                                  ↑20,*,4       ***     17 ==>     12
                *** ERROR    4: ")" expected
                *** ERROR 20: "," expected
     1800   18           1
     1900   19           1    end.
            20           0

Compilation time =          1.35 seconds (      889 lines/minute).

  2 Errors      1 Nonstandard feature

Last error(warning) on line          17.
Active options at end of compilation:
NODEBUG,STANDARD,LIST,NOCHECK,WARNINGS,CROSS_REFERENCE,
MACHINE-CODE,OBJECT,ERROR_LIMIT = 30
```

Figure 7-22
Sample VAX-11 Pascal Code

MACHINE-CODE

GENERATED CODE (PRIOR TO BRANCH OPTIMIZATION)

LINE	ADDRESS	OPCODE	OPERANDS	BYTESTREAM (HEXADECIMAL; READ FROM RIGHT TO LEFT)
	0002	MOVAB	*VAR(0, 0),R11	58 00 00 00 00 00 9E
	0009	MOVL	R13,*VAR(0, 4)	00 00 00 00 00 5D D0
	0010	CLRL	−(R14)	7E D4
	0012	CLRD	−(R14)	7E 7C
	0014	PUSHAL	(R11)B↑8	08 AB DF
	0017	CALLS	#1,PAS$INPUT	00 00 00 00 00 01 FB
	001E	PUSHAL	(R11)B↑8	98 AB DF
	0021	PUSHAL	(R11)W↑236	00 EC CB DF
	0025	CALLS	#2,PAS$OUTPUT (22)	00 00 00 00 00 02 FB
(23) 8	002C	MOVL	R14,(R13)B↑−12	F4 AD 5E D0
9	0030	MOVAB	(R11)W↑236,R10	5A 00 EC CB 9E
	0035	SUBL2	#16,R14	(25) 5E 10 C2
	0038 (24)	PUSHL	R10	5A DD
	003A	MOVAB	*VAR(2, 0),R9	59 00 00 00 00 00 9E
	0041	MOVL	R9,(R14)B↑4	04 AE 59 D0
	0045	MOVL	#21,(R14)B↑12	0C AE 15 D0
	0049	MOVL	#21,(R14)B↑8	08 AE 15 D0
	004D	CALLS	#5,PAS$WRITESTR	00 00 00 00 00 05 FB
	0054	MOVAB	(R11)W↑236,R10	5A 00 EC CB 9E

CROSS_REFERENCE

CROSSREFERENCE LISTING

A	4	11	12	
B (28)	4	11	12 (26)	
C	(29) 4	12	13	
INPUT	1	10		
OUTPUT	1			
GLOBALLY DEFINED IDENTIFIERS:				
EOLN	0	10		
FALSE	0	14		
READLN	0	11		
REAL	0	4	(27)	
SQR	0	12	12	
WRITELN	0	9	13	17

Figure 7-22 (Con't)
Sample VAX-11 Pascal Code

Source Code Listing — The lines of the source code are printed in the source code listing. In addition, the listing contains the following information pertaining to the source code:

- SOS line numbers (7)—If the source lines were created or edited in a Pascal module with the SOS editor, SOS line numbers appear in the leftmost column of the source code listing. SOS line numbers are irrelevant to the Pascal compiler.
- Line numbers (8)—The compiler assigns unique line numbers to the source lines in a Pascal module. The symbolic traceback that is printed if the program encounters an exception at run time refers to these line numbers.
- Procedure level (9)—Each line that contains a declaration lists the procedure level of that declaration. Procedure level 1 indicates declarations in the outermost block. The procedure level number increases by one for each nesting level of functions or procedures.
- Statement level (10)—The listing specifies a statement level for each line of source code after the first BEGIN delimiter. The statement level starts at 0 and increases by 1 for each nesting level of Pascal structured statements. The statement level of a comment is the same level as that of the statement that follows it.

Errors and Warnings — The source code listing includes information on any errors or warnings detected by the compiler. A line beneath the source code line in which the error is detected specifies whether the diagnostic is a warning or an error. In addition, the error description can contain the following information:

- A circumflex (↑) that points to the character position in the line where the error was detected (11).
- A numeric code, following the circumflex, that specifies

the particular error (12). On the following lines of the source listing, the compiler prints the text that corresponds to each numeric code (13). Note that one source program error often causes the Pascal compiler to detect more than one error (14).

- An asterisk (*) that shows where the compiler resumed translation after the error (15).
- The line number in which the error was detected (16) and the line number of the last line containing an error diagnostic (17). These error line numbers can be used to trace the error diagnostics backwards through the source listing.

Summary — At the end of the source listing, the compiler lists the amount of time required for the compilation (18). If program generated warning or error messages resulted, the compiler prints a summary of all the errors (20) and the source line number of the last message (19). Finally, the compiler lists the status of all the compilation options (21).

Machine Code Listing

The machine code listing (if requested with the MACHINE_CODE qualifier) follows the source listing. The machine code listing contains:

- Symbolic representation (22)—The symbolic representation, similar to a VAX-11 MACRO instruction, appears for each object instruction generated. Because the Pascal compiler operates in one pass, it must generate these instructions before it performs branch optimization. Branch optimization can cause certain BRW instructions to be deleted. Therefore, these instructions will not be identical to those appearing in the executable image.
- Source line number (23)—A source line number marks the first object instruction that the compiler generated for the first Pascal statement on that source line.
- Hexadecimal address (24)—The hexadecimal address is an approximation of the address of the object instruction. Do not use these addresses for debugging purposes because they do not correctly correspond to the locations in the executable image. The branch optimization mentioned above can change the addresses of the object instructions.
- Hexadecimal instruction (25)—This is the hexadecimal representation of the object instruction. The hexadecimal instruction should be read from right to left because the rightmost byte has the smallest address. Again, because of its one-pass operation, the compiler must generate some object instructions before it can determine the address bytes of their operands. The addresses of these operands are printed as zeros. After generating the hexadecimal representation of an instruction, but before writing the object code file, the compiler places the correct values into the binary object code.

Cross-Reference Listing

The cross-reference listing (if requested with the CROSS_REFERENCE qualifier) appears after the machine code listing. It contains two sections:

- User-specified identifiers (26)—This section lists all the user declared identifiers.
- Globally-defined identifiers (27)—This section lists the Pascal predefined identifiers that the program uses.

Each line of the cross-reference listing contains an identifier (28) and a list of the source line numbers where the identifier is used (29). The first line number indicates where the identifier is declared. Predefined identifiers are listed as if they were declared on line 0. The cross-reference listing does not specify pointer type identifiers that are used before they are declared.

VAX-11 BLISS-32

Introduction

BLISS-32 is a high level systems implementation language for VAX-11. It is specifically designed for building software such as compilers, real-time processors, and operating systems modules. (For example, the VAX-11 FORTRAN compiler is coded in BLISS, as is most of the VAX-11 RTL.) It contains many of the features of high-level languages (e.g., DO loops, IF-THEN-ELSE statements, automatic stack and register allocations, and mechanisms for defining and calling routines) but also provides the flexibility, efficiency, and access to hardware which one would expect from an assembly language. The BLISS compiler produces highly-optimized object code which is typically within 5% to 10% of the efficiency of code produced by an experienced assembly language programmer. The VAX-11 BLISS-32 compiler is written entirely in BLISS and runs in native mode under the VAX/VMS operating system.

Features of BLISS-32

VAX-11 BLISS-32 has several characteristics which set it apart from other high-level languages:

- Data—BLISS-32 is "type-free": all data are manipulated as values from 1 to 32 bits in length. The interpretation of any data item is provided by the operator applied to it. A value can be fetched from or assigned to any contiguous field of from 1 to 32 bits located anywhere in the VAX-11 virtual address space. The expression Y<4,4>=0 deposits zero into bits 4 through 7 of location Y. This BLISS expression generates the single VAX-11 instruction: BICB2 #^XF0,Y.
- Value Assignments—all names in BLISS-32 represent addresses. Contents of storage locations are accessed by means of a fetch operator (.). Hence, the expression X=.Y+3 is interpreted as adding 3 to the contents of location Y, then assigning the result to the storage location beginning at X.
- Operators—BLISS-32 supplies operators which interpret their operands as addresses, signed integers, unsigned integers or character-sequences.
- Expressions—BLISS-32 permits construction of complex expressions in which several different kinds of operations can be performed in a single program statement. For example, the expression 2*(B=.C+1) calculates 2*(.C+1) and simultaneously assigns the value of .C+1 to B.
- Structures—BLISS-32 defines such data structures as VECTOR, BLOCK, BITVECTOR, and BLOCKVECTOR. In addition, the programmer can define arbitrary data structures specifically designed for a given application.

Other BLISS-32 features include:

- CASE, SELECT, SELECTONE, and IF-THEN-ELSE—providing for sequencing of instructions based on evaluation of expressions at run-time.
- DO, WHILE, and UNTIL—providing for looping until a particular condition is satisfied.
- INCR, DECR—providing for iterative looping, incrementing or decrementing a loop index.
- EXITLOOP and LEAVE—providing for early termination of loops and for exiting a BEGIN-END block. (There is no GO TO in BLISS.)
- Condition Handling—the BLISS-32 language fully supports a VAX/VMS condition-handling environment. The ENABLE declaration establishes a condition-handler for exceptions raised within the scope of a routine. The SIGNAL, SIGNAL_STOP and UNWIND predeclared functions allow a programmer to raise conditions and process them.
- GLOBAL and EXTERNAL declarations—allowing code and data to be shared among several modules.
- LOCAL, STACKLOCAL, and REGISTER declarations—allowing dynamic stack-like allocation using either the execution stack or the general registers.
- REQUIRE and LIBRARY declarations—allowing source material from subsidiary files to be included in the module at compile time.
- LEXICAL functions—allowing a variety of compile-time operations such as concatenation of strings, construction of names, testing properties of macro parameters, testing compiler switch options, generating compiler diagnostic messages, and controlling macro expansion.
- Conditional Compilation—The programmer may include or exclude source text from compilation through use of %IF-%THEN-%ELSE-%FI. In conjunction with the lexical functions, this provides a powerful capability for tailoring source code for distinct environments.

BLISS-32 also provides a number of machine-dependent features specifically designed to complement the VAX architecture and VAX/VMS. These include the following:

- LINKAGE declarations allow the programmer to make full use of the VAX-11 call facilities; in particular, you may specify either the CALLS/CALLG/RET or JSB/BSB/RSB call and return sequences, pass parameters in general registers or on the stack, and control the use of registers by a routine or across a set of routines.
- PSECT declarations provide information to the linker regarding storage requirements for various sections of a program. For example, a particular data segment may be designated as READ or NOREAD, SHARE or NOSHARE, LOCAL or GLOBAL, and so on.
- System Interfaces—STARLET.REQ (or STARLET.L32) provides keyword macros for all VAX-11 RMS and VAX/VMS system services, as well as symbolic definitions for system data structures and completion codes. LIB.REQ (or LIB.L32) provides the definitions in STARLET, as well as definitions needed for writing ACPs or privileged kernel-mode code.
- BUILTIN declarations allow use of VAX-11 machine-specific functions for access to VAX-11 features not otherwise provided in the BLISS-32 language. The compilation of a machine specific function results in the generation of inline code, often a single instruction, rather than a call to an external routine. Machine specific functions generally have the same name as their corresponding VAX-11 instructions (e.g., ADAWI, BISPSW, CRC, HALT, INDEX, MTPR, PROBER, REMQUE, MOVP, etc.). Over 50 such functions are provided. (The complete list is shown in Table 7-4).

Table 7-4

VAX-11 Machine Specific Functions

Processor Register Operations	
MTPR	Move to a Processor Register
MFPR	Move from a Processor Register
Parameter Validation Operations	
PROBER	Probe Read accessibility
PROBEW	Probe Write accessibility
Program Status Operations	
MOVPSL	Move from PSL
BISPSW	Bit set PSW
BICPSW	Bit clear PSW
Queue Operations	
INSQUE	Insert entry in Queue
REMQUE	Remove entry from Queue
Bit Operations	
TESTBITSS	Test for Bit Set, then Set bit
TESTBITSC	Test for Bit Set, then Clear bit
TESTBITCS	Test for Bit Clear, then Set bit
TESTBITCC	Test for Bit Clear, then Clear bit
TESTBITSSI	Test for Bit Set, then Set bit Interlocked
TESTBITCCI	Test for Bit Clear, then Clear bit Interlocked
FFS	Find First Set bit
FFC	Find First Clear bit
Extended Arithmetic Operations	
ASHQ	Arithmetic Shift Quad
EDIV	Extended Divide
EMUL	Extended Multiply
INDEX	Index (Subscript) Calculation
CRC	Cyclic Redundancy Calculation
Floating Point Conversion Operations	
CVTLF	Convert Long to Floating
CVTLD	Convert Long to Double
CVTFL	Convert Floating to Long
CVTDL	Convert Double to Long
CVTFD	Convert Floating to Double

Table 7-4 (con't)
VAX-11 Machine Specific Functions

CVTDF	Convert Double to Floating
CVTRDL	Convert Rounded Double to Long
CVTRFL	Convert Rounded Floating to Long
CMPF	Compare Floating
CMPD	Compare Double
String Operations	
MOVTUC	Move Translated Until Character
SCANC	Scan Characters
SPANC	Span Characters
Decimal String Operations	
MOVP	Move Packed
CMPP	Compare Packed
CVTLP	Convert Long to Packed
CVTPL	Convert Packed to Long
CVTPT	Convert Packed to Trailing Numeric
CVTTP	Convert Trailing Numeric to Packed
CVTPS	Convert Packed to Leading Separate Numeric
CVTSP	Convert Leading Separate Numeric to Packed
EDITPC	Edit Packed to Character
Miscellaneous Operations	
HALT	Halt Processor
ROT	Rotate
ADAWI	Add Aligned Word Interlocked
BPT	Breakpoint
CHMx	Change Mode
CALLG	Call with General Argument List
NOP	No Operation

The VAX-11 BLISS-32 Compiler

The VAX-11 BLISS-32 compiler performs a number of optimizations. These include common subexpression elimination, removal of loop invariants, constant folding, block register allocation, peephole replacement, test instruction elimination, jump vs. branch instruction resolution, branch chaining, and cross-jumping.

The VAX-11 BLISS-32 compiler optionally produces source text and generated code in a format closely resembling a VAX-11 assembly listing. Other options allow the programmer to control the degree of optimization, suppress production of object code, determine types and formats of output listings, generate traceback information, and specify the types of information to be listed at the terminal.

BLISS-32 generates shareable, re-entrant and position-independent code. These features make it easy to use BLISS-32 for writing software to be included in shareable libraries for use by any native language. it is also useful for writing connect-to-interrupt device handlers and user-written system services.

Library and Require Files

BLISS-32 provides two methods for including commonly used text into BLISS programs at compile time. These involve use of either Library files or Require files:

- Library Files—These are special files created by the compiler in a previous library compilation and are invoked by the LIBRARY declaration in the BLISS source program.
- Require Files—These are source (text) files which are invoked via the REQUIRE declaration in the BLISS source program.

Since Library files are "precompiled," lexical processing and declaration parsing and checking need not be repeated each time these files are included in a compilation; their use results in a considerable reduction in total compilation time.

The contents of Require files must be fully processed each time the file is used in a compilation. Hence, using Require files will, in general, be less efficient than using Library files. However, since these files operate under a less stringent set of syntactical rules, their use may be warranted in situations where a higher level of flexibility is desired.

Macros

VAX-11 BLISS-32 provides an extensive macro-building facility, allowing frequently used groups of declarations or expressions to be expressed in an abbreviated way. Macros are defined via MACRO declarations and are accessed by simple call statements. They are fully expanded at compile time. BLISS-32 allows parameters to be specified in the macro definition, thus allowing each block of text to be specialized by the actual parameters passed to it. Macros may be positional or keyword, and may be simple, iterative, or conditional.

Debugging

The VAX-11 BLISS-32 compiler produces a list of error messages showing the source program line on which the error occurred followed by a description of the error. If the error is recoverable, then the compiler will generate a "warning" diagnostic and continue with the compilation process. If the error is serious enough to invalidate the compiler's internal representation of the module, then an "error" diagnostic is generated, and processing ceases following the syntax checking—no object module is produced.

If an error occurs at execution time, the process image can access the VAX-11 Symbolic Debugger program. This program may be accessed when the object module is linked with the DEBUG option. The DEBUG program allows the programmer to examine and deposit values in storage, set breakpoints, call routines, trace through a program as it executes, and perform other operations useful in checking out a program. VAX-11 DEBUG understands BLISS syntax and permits the use of the user's symbolic names. (See the section on the Symbolic Debugger for a further description of VAX-11 debugging facilities.)

Transportability Features

The BLISS-32 language is designed to facilitate transportability, that is, the writing of programs that can be executed on architecturally different machines with little or no

modification. BLISS compilers also exist for the DECsystem-10 and DECSYSTEM-20 (BLISS-36), and for the PDP-11 (BLISS-16). Several language features enhance transportability:

- The high-level language constructs may be transferred from one machine to another with little or no alteration.
- Machine-specific functions can be separated from the common, mainline code via modularization, macros, and Library and Require files.
- Parameterization allows machine-specific characteristics to be passed to BLISS data structures.
- The compiler can be instructed to perform transportability checking via the "LANGUAGE (COMMON)" module-head switch.

BLISS's transportability makes it an ideal language for system programming applications—and a desirable alternative to assembly language coding in those applications in which extreme machine dependence is not involved.

BLISS-32 Sample Program

Figure 7-23 illustrates how VAX-11 BLISS-32 can call VAX/VMS system services and the VAX-11 Common Run Time Procedure Library to print the current time on SYS$OUTPUT.

VAX-11 CORAL 66

The VAX-11 CORAL 66 compiler compiles in compatibility mode and generates native mode object code under VAX/VMS. The CORAL 66 language, derived from JOVIAL and ALGOL-60 in 1966, is the standard language prescribed by the British government for military real-time applications and systems implementation. A government agency controls the language standard for CORAL 66, which was first widely used in military projects beginning in 1970. Her Majesty's Stationery Office publishes the "Official Definition of CORAL 66."

The CORAL 66 language replaces assembly-level programming in a number of commercial, process control, research, and military applications. It is particularly adapted to long-life products requiring flexibility and ease of maintenance.

VAX-11 CORAL 66 is a block-structured language. A block is a piece of a program that can be entered only at the be-

```
;  0001  MODULE showtime( IDENT='1-1' %TITLE 'SHOW TIME', MAIN=timeout)=
;  0002  BEGIN
;  0003
;  0004  LIBRARY 'SYS$LIBRARY:STARLET';                    ! Defines System Services, etc.
;  0005
;  0006  MACRO
; M0007    desc[] = %CHARCOUNT(%REMAINING),                ! A VAX-11 Style String descriptor
;  0008         UPLIT BYTE( %REMAINING) %;
;  0009
;  0010  OWN
;  0011         timebuf:        VECTOR[2],                                  ! 64 bit system time
;  0012         msgbuf:         VECTOR[80,BYTE],                            ! Output msg. buffer
;  0013         msgdesc:        BLOCK[8,BYTE]                               ! String descriptor
;  0014                           PRESET(     [dsc$w_length]= 80,           ! for output buffer
;  0015                                       [dsc$a_pointer] = msgbuf );
;  0016
;  0017  BIND
;  0018         fmtdesc= UPLIT(DESC('At the tone, the time is ', %CHAR(7), '115%T' ));
;  0019
;  0020  EXTERNAL ROUTINE
;  0021         lib$put_output : ADDRESSING_MODE(GENERAL);                  ! From VMS RTL
;  0022
;  0023  ROUTINE timeout=
;  0024         BEGIN
;  0025         LOCAL
;  0026               RSLT:           WORD;                  ! Resultant string length
;  0027
;  0028         $GETTIM( TIMADR=timebuf );                     ! Get time as 64 bit integer
;  0029
; P0030         $FAOL(                CTRSTR=fmtdesc,          ! Format control-string address
; P0031                               OUTLEN=nslt,             ! Resultant length (only a word!)
; P0032                               OUTBUF=msgdesc,          ! Output buffer descriptor address
;  0033                               PRMLST=%REF(timebuf));   ! Address of pointer to time block
;  0034
;  0035         MSGDESC[dsc$w_length] = .rslt;                 ! modify output descriptor
;  0036
;  0037         lib$put_output( msgdesc )                      ! print the formatted time
;  0038
;  0039         END;
```

Figure 7-23
Sample VAX-11 BLISS-32 Code

ginning. Though internal structures cannot be "seen" from the outside, statements inside a block can "see" out. Sorting is possible, so that programs may be written in which information is accessible for only the time it is required, and no longer. In this way, unwanted interactions among program parts are avoided, and out-of-date information is very quickly forgotten.

To satisfy real-time needs, CORAL 66 allows different modules of the same suite of programs to be executed at apparently the same time. A CORAL 66 compiler assumes that any subroutine global to the whole program is likely to be active at the same time as any other, so the compiler assures that such subroutines do not share any local storage. Interactions, however, can be explicitly arranged by the programmer. A program consists of communicators and separately compiled segments. Each segment has the form of an ALGOL-60 block, within which blocks and procedures may be nested to arbitrary depth. In the absence of communicators, block structure would prevent different segments from using common data, labels, switches, or procedures. The purpose of a communicator is to specify and name those objects which are to be commonly accessible to all segments. The presence of communicators imposes a modular and disciplined approach to programming larger systems where a team of programmers is employed.

In addition to the functionalities prescribed in the Official Definition, the VAX-11 CORAL 66 compiler provides the following features:

- BYTE, LONG (32-bit integer) and DOUBLE (64-bit floating point) numeric types
- generation of re-entrant code at the procedure level
- switch-selectable option to optimize generated code
- conditional compilation of defined parts of source code
- English text error messages at compile and (optionally) run-time
- switch-selectable option to control listing output
- INCLUDE keyword to incorporate CORAL 66 source code from user-defined files
- switch-selectable option to read card format

VAX-11 CORAL 66 is essentially a high-level block-structured language possessing certain facilities associated with low-level languages, and is designed for use on small or medium-size dedicated computers. One of the main intentions is that programs written in CORAL 66 should be fast to execute, taking up limited quantities of storage, while being easy to write.

The real-time applications of the language are implicit rather than explicit, permitting the utilization of any hardware or special features. Procedures, optionally with parameters, permit communication with and reaction to external events. This is aided further by a direct code facility which enables machine code to be included in the source program for extra efficiency in any critical tasks.

VAX-11 MACRO

The VAX-11 MACRO assembler accepts one or more source modules written in MACRO assembly language and produces a relocatable object module and optional assembly listing. VAX-11 MACRO is similar to PDP-11 MACRO, but its instruction mnemonics correspond to the VAX native instructions. VAX-11 MACRO is characterized by:

- relocatable object modules
- global symbols for linking separately assembled object programs
- global arithmetic, global assignment operator, global label operator and default global declarations
- user-defined macros with keyword arguments
- multiple macro libraries with fast access structure
- program sectioning directives
- conditional assembly directives
- assembly and listing control functions
- alphabetized, formatted symbol table listing
- default error listing on command output device
- a Cross Reference Table (CREF) symbol listing

Symbols and Symbol Definitions

Three types of symbols can be defined for use within MACRO source programs: permanent symbols, user-defined symbols and macro symbols. Permanent symbols consist of the VAX instruction mnemonics and MACRO directives and do not have to be defined by the user. User-defined symbols are those used as labels or defined by direct assignment. Macro symbols are those symbols used as macro names.

MACRO maintains a symbol table for each type of symbol. The value of a symbol depends on its use in the program. To determine the value of a symbol in the operator field, the assembler searches the macro symbol table, user symbol table, and permanent symbol table in that order. To determine the value of the symbol used in the operand field, the assembler searches the user symbol table and the permanent symbol table in that order. These search orders allow redefinition of permanent symbol table entries as user-defined or macro symbols.

User-defined symbols are either internal to a source program module or global (externally available). An internal symbol definition is limited to the module in which it appears. Internal symbols are local definitions which are resolved by the assembler.

A global symbol can be defined in one source program module and referenced within another. Global symbols are preserved in the object module and are not resolved until the object modules are linked into an executable program. With some exceptions, all user-defined symbols are internal unless explicitly defined as being global.

Directives

A program statement can contain one of three different operators: a macro call, a VAX instruction mnemonic, or an assembler directive. MACRO includes directives for:

- listing control
- function specification
- data storage
- radix and numeric usage declarations
- location counter control

- program termination
- program boundaries information
- program sectioning
- global symbol definition
- conditional assembly
- macro definition
- macro attributes
- macro message control
- repeat block definition
- macro libraries

Listing Control Directives
Several listing control directives are provided in MACRO to control the content, format, and pagination of all listing output generated during assembly. Facilities also exist for titling object modules and presenting other identification information in the listing output.

The listing control options can also be specified at assembly time through qualifiers in the command string issued to the MACRO assembler. The use of these qualifiers provides initial listing control options that may be overridden by the corresponding listing control directives in the source program.

Conditional Assembly Directives
Conditional assembly directives enable the programmer to include or exclude blocks of source code during the assembly process, based on the evaluation of stated condition tests within the body of the program. This capability allows several variations of a program to be generated from the same source module.

The user can define a conditional assembly block of code, and within that block, issue subconditional directives. Subconditional directives can indicate the conditional or unconditional assembly of an alternate or non-contiguous body of code within the conditional assembly block. Conditional assembly directives can be nested.

Macro Definitions and Repeat Blocks
In assembly language programming, it is often convenient and desirable to generate a recurring coding sequence by invoking a single statement within the program. In order to do this, the desired coding sequence is first established with dummy arguments as a macro definition. Once a macro has been defined, a single statement calling the macro by name with a list of real arguments (replacing the corresponding dummy arguments in the macro definition) generates the desired coding sequence or macro expansion. MACRO automatically creates unique symbols where a label is required in an expanded macro to avoid duplicate label specifications. Macros can be nested; that is, the definition of one macro can include a call to another.

An indefinite repeat block is a structure that is similar to a macro definition, except it has only one dummy argument. At each expansion of the indefinite repeat range, this dummy argument is replaced with successive elements of a specified real argument list. This type of macro definition does not require calling the macro by name, as required in the expansion of conventional macros. An indefinite repeat block can appear within or outside of another macro definition, indefinite repeat block, or repeat block.

Macro Calls and Structured Macro Libraries
A program can call macros that are not defined in that program. A user can create libraries of macro definitions, and MACRO will look up definitions in one or more given library files when the calls are encountered in the program. Each library file contains an index of the macro definitions it contains to enable MACRO to find definitions quickly.

Program Sectioning
The MACRO program sectioning directives are used to declare names for program sections and to establish certain program section attributes. These program section attributes are used when the program is linked into an image.

The program sectioning directive allows the user to exercise complete control over the virtual memory allocation of a program, since any program attributes established through this directive are passed to the linker. For example, if a programmer is writing multiuser programs, the program sections containing only instructions can be declared separately from the sections containing only data. Furthermore, these program sections can be declared as read-only code, qualifying them for use as protected, reentrant programs.

PDP-11 BASIC-PLUS-2/VAX

PDP-11 BASIC-PLUS-2/VAX is an optional language processing system that includes a compiler and an object time system. PDP-11 BASIC-PLUS-2is also available as an optional language processor for the RSTS/E, RSX-11M, RSX-11M PLUS, and IAS operating systems. The PDP-11 BASIC-PLUS-2/VAX compiler produces code that executes in PDP-11 compatibility mode.

BASIC-PLUS-2 is a PDP-11 applications implementation language which features many programming facilities found in VAX-11 BASIC. These include:

- program formatting and commenting facilities
- long variable names
- indexed, sequential, and relative file I/O
- virtual arrays
- variable-length strings
- PRINT USING statement
- COMMON statement
- a subprogram CALL statement
- extended debugging facilities
- integrated INQUIRE (HELP) facility

The compiler accepts source programs written in the BASIC-PLUS-2 language.

The programmer can edit the source if necessary, and compile it to produce an object module which can be linked with previously compiled object modules.

The object time system (OTS) is a collection of library modules used during program execution. The library routines include math and floating point functions, input/output operations, error handling, and dynamic string storage functions. Since the OTS is an object module library, the task builder can select only those functions needed at run time to be included in a program. Unnecessary routines are omitted from the program and memory usage is reduced.

Program Format

The basic source program unit is a line. In its simplest form, it consists of a line number, a keyword and statement, and a line terminator. In BASIC-PLUS-2, one or several spaces or tabs can be used to separate line numbers, keywords, and variable names. Line numbering determines the order in which the program is processed; the programmer can enter BASIC-PLUS-2 program lines in any order.

BASIC-PLUS-2 programs can be one or several lines long. The programmer can place one statement on each line, place several statements on any one line, or spread one statement over several lines. Program comments can be placed anywhere within a line using the REM (Remark) statement or using comment field delimiters. These facilities enable the programmer to freely format a program to make it more readable.

Long Variable and Function Names

Most BASIC languages limit the length of a variable or user-defined function name to one character. BASIC-PLUS-2 recognizes variable names and function names as long as 30 characters. The programmer can fully identify variables and functions.

Powerful File I/O

The BASIC-PLUS-2 language supports use of RMS-11 block, indexed, sequential, and relative file operations. Although RMS-11 operates in RSX-11 compatibility mode, it does not have support for file sharing under VMS. For applications requiring access to shared files, VAX-11 BASIC should be used.

Powerful String Handling

The BASIC-PLUS-2 language enables the programmer to manipulate strings of alphanumeric characters easily. As in BASIC-PLUS, the BASIC-PLUS-2 relational operators enable programmers to concatenate and compare strings, string operators enable the programmer to convert strings and numerics, and string functions add the ability to analyze the composition of strings. The BASIC-PLUS-2 language includes string functions that:

- create a string of a fixed length
- search for the position of a set of characters within a string
- insert spaces within a string
- trim trailing blanks from a string
- determine the length of a string

Unlike many BASIC languages, BASIC-PLUS-2 imposes no limit on the size of string values or string elements of arrays manipulated in memory, other than the amount of available memory.

Virtual Arrays

Virtual arrays are random access disk-resident files. A program can create and access virtual arrays in the same way memory-resident arrays are accessed: using element names. Explicit read/write programming is not required. The last element in the array can be accessed as quickly as the first. Because the arrays are stored on disk, however, the programmer can manipulate large amounts of data without affecting program size.

PRINT USING Output Formats

The PRINT USING statement allows the programmer to control the appearance and location of data on an output line to create complex lists, tables, reports, and forms. In addition to the numeric field definitions provided by BASIC-PLUS, which allow the programmer to generate floating dollar sign, aligned decimal point, trailing minus, asterisk fill, and exponential format fields, BASIC-PLUS-2 provides string field definitions which allow the programmer to generate left-justified, right-justified, centered, and extended string fields.

Subprograms and the CALL Statement

The BASIC-PLUS-2 CALL statement enables a program to access external BASIC-PLUS-2 or MACRO-11 subprograms. A programmer can therefore write a program in several modular segments, each of which can be compiled separately to speed program development. BASIC-PLUS-2 provides a complete traceback on errors occurring in subroutines.

COMMON Statement

The COMMON statement enables a program to pass data to another program or subprogram. The BASIC-PLUS-2 COMMON statement format is compatible with PDP-11 FORTRAN COMMON. Strings passed in COMMON are fixed length, which reduces string handling overhead.

Debugging Statements

BASIC-PLUS-2 provides an interactive debugging mode similar to the "immediate mode" facilities found in most BASIC interpreters. During program development, the programmer can use the compiler to create, save, edit, and test the source program. The compiler checks syntax immediately on input from a terminal so that many errors can be found prior to compilation. The debugging statements can be used when executing and testing the program. The BREAK, LET, PRINT, UNBREAK, CONTINUE, STEP, and STOP statements enable the programmer to control and observe program execution interactively.

To set breakpoints, the programmer uses the BREAK command just prior to running the program, or while it is stopped. As many as ten breakpoints can be set during the course of program execution. On reaching a breakpoint, the program halts to allow the programmer to examine or modify variables or set other breakpoints.

To examine variables while a program is stopped, the programmer uses the PRINT statement. The LET statement allows the programmer to modify the value stored in the variable.

Typing the CONTINUE command resumes execution until the next breakpoint is reached. Before typing CONTINUE, the programmer can issue an UNBREAK command to selectively disable one or all of the breakpoints set, and execution continues until a STOP statement is encountered in the program or until the program completes.

When a program halts because a STOP statement is included in the program, or because a BREAK command was issued interactively, the programmer can type the STEP command on the terminal to let program execution continue on a line-by-line basis. Typing a STOP command in interactive debugging mode terminates program execution, just as if an END statement was encountered in the program.

PDP-11 FORTRAN IV/VAX to RSX

FORTRAN IV is an extended FORTRAN implementation based on American National Standard (ANSI) FORTRAN, X3.9-1966. PDP-11 FORTRAN IV code is executed in compatibility mode under VAX/VMS. The FORTRAN IV language includes the following extensions to the ANSI standard;

- general expressions allowed in all meaningful contexts
- mixed-mode arithmetic
- BYTE data type for character manipulation
- ENCODE, DECODE statements
- PRINT, TYPE, and ACCEPT input/output statements
- direct-access unformatted input/output DEFINE FILE statement
- comments allowed at the end of each source line
- PROGRAM statement
- OPEN and CLOSE file access control statements
- list-directed input/output

Additionally, virtual arrays are supported on target systems with memory management directives. Virtual arrays are memory-resident and require enough main memory to contain all elements of all arrays.

The PDP-11 FORTRAN IV compiler is a fast, one-pass compiler. Compiler options allow program size versus execution speed (threaded code versus inline code) tradeoffs. FORTRAN IV compiler optimizations include:

- common subexpression elimination
- local code tailoring
- array vectoring
- optional inline code generation for integer and logical operations

MACRO-11 assembly language subroutines may be called from FORTRAN IV programs. FORTRAN IV also includes a set of object modules, called the Object Time System (OTS), that are selectively linked with compiler-produced object modules to produce an executable program.

MACRO-11

MACRO-11, the PDP-11 assembly language, is included in the compatibility mode environment. Programs written in MACRO-11 can be assembled to produce relocatable object modules and optional assembly listings. MACRO-11 provides the following features:

- relocateable object modules
- global symbols for linking separately assembled object programs
- device and file name specifications for input and output files
- user-defined macros
- comprehensive system macro library
- program sectioning directives
- conditional assembly directives
- assembly and listing control functions at program and command string levels
- alphabetized, formatted symbol table listing
- default error listing on command output device

Symbols and Symbol Definitions

Three types of symbols can be defined for use within MACRO-11 source programs: permanent symbols, user-defined symbols, and macro symbols. Accordingly, MACRO maintains three types of symbol tables: the Permanent Symbol Table (PST), the User Symbol Table (UST), and the Macro Symbol Table (MST).

Permanent symbols consist of the PDP-11 instruction mnemonics and MACRO directives. The PST contains all the permanent symbols automatically recognized by MACRO and is part of the assembler itself. Since these symbols are permanent, they do not have to be defined by the user in the source program.

User-defined symbols are those used as labels or defined by direct assignment. Macro symbols are those symbols used as macro names. The UST and MST are constructed during assembly by adding the symbols to the UST or MST as they are encountered.

Directives

A program statement can contain one of three different operators: a macro call, a PDP-11 instruction mnemonic, or an assembler directive. MACRO includes directives for:

- listing control
- function specification
- data storage
- radix and numeric usage declarations
- location counter control
- program termination
- program boundaries information
- program sectioning
- global symbol definition
- conditional assembly
- macro definition
- macro attributes
- macro message control
- repeat block definition
- macro libraries

8

Program Development and Support Facilities

VAX/VMS offers the user a powerful and sophisticated program development environment including several support facilities. Described in this section are the interactive and batch text editors, the linker, the Common Run Time Procedure Library, the VAX-11 interactive symbolic debugger, the librarian utility, command language procedures, the differences utility, and VAX-11 RUNOFF.

In addition to the interactive text editor SOS and the SLP batch editor, VAX/VMS now supports the powerful interactive text editor EDT. EDT supports many user oriented features including both line and character editing facilities, an extensive HELP facility, a journaling facility, and the window editing facility.

The VAX/VMS linker is the program development tool that takes the output of a translator or compiler and produces a file that can be executed on the VAX-11 hardware.

The Common Run Time Procedure Library offers the user a set of common language routines that can be called from any of the native mode languages via the VAX-11 calling standard.

The VAX-11 interactive symbolic debugger is extremely useful in isolating program errors by allowing the user to examine and modify the contents of memory locations while the program is executing.

The librarian is a utility that allows the user easy access to the data stored in any of the four libraries (object, macro, help, text). The librarian allows an executing program to initialize and open a library, and to retreive, insert, and delete modules.

The command language procedure section describes the power and flexibility of executing strings of frequently used sequences of DCL commands, or creating new commands from the existing DCL command set. Command procedures can accept up to eight parameters and can include extensive control flow.

By utilyzing the DIFFERENCES utility, the user can determine whether or not two files are identical.

VAX-11 RUNOFF is a document formatter, offering the user such features as page and title formatting, subject-matter formatting, and the ability to produce indexes and tables of contents easily.

INTRODUCTION

VAX/VMS provides a complete program development environment for the user. Development tools supporting this environment are:

- Interactive and batch text editors
- Linker
- Common Run Time Procedure Library
- VAX-11 interactive symbolic debugger
- Librarian Utility
- Command Language Procedures
- Differences Utility
- VAX-11 Runoff

These tools, as well as program language compilers, are available to the programmer via the command language facility. In addition, VAX/VMS supports a complete set of shareable routines collectively known as the common run time procedure library. And finally, VAX/VMS supports the user's ability to write programs utilizing the DCL command set (similar to coding programs in other high level languages). These programs are known as command language procedures.

The text editors can be used to create memos, documentation, and text and data files, as well as source program modules for any language processor. The linker, librarian, debugger, and run time procedure library are used only in conjunction with the language processors that produce native code.

TEXT EDITORS

Text editing refers to the process of creating, modifying, and maintaining files. VAX/VMS supports three text editors: two interactive text editors (SOS and EDT) and a batch-oriented text editor (SLP).

The user invokes the SOS and EDT text editors interactively, i.e., the user creates and processes files on-line. The SLP text editor, on the other hand, allows direct modification to a file via a command file prepared by the user. SOS is often used to create SLP command files. All editors are invoked by the command EDIT. The default editor is SOS. Therefore, to invoke SOS, enter the command EDIT or EDIT/SOS; to invoke EDT, enter EDIT/EDT; for SLP, use EDIT/SLP.

Before describing the text editors, a short summary of file naming conventions and default file types is presented.

File Names and File Types

By taking advantage of the default disk and directory, the user can identify a file uniquely by specifying its file name and file type, illustrated in the following format:

```
filename.typ
```

The file name can be from one to nine alphanumeric characters, and can assume any name that is meaningful to the user.

The file type is a 3-character identifier preceded by a period; it describes more specifically the kind of data in the file. Although file type can consist of any three alphanumeric characters meaningful to the user, several file types have standard meanings. Among these special file types are:

File Type	Default Use
.FOR	FORTRAN language source statements
.MAR	MACRO assembly source statements
.COB	COBOL language source statements
.BAS	BASIC language source statements
.PAS	PASCAL language source statements
.DAT	A data file
.LIS	An output listing from a compiler
.EXE	An executable image
.OBJ	An output file from a compiler

For example, a file containing FORTRAN source statements would possess the file type .FOR.

SOS EDITOR

SOS is a line-oriented, interactive text-editing program. SOS has features that allow examination and modification of text, character by character. SOS can be used to perform the following functions:

- examine, create, and modify ASCII files
- search for and/or change one or more arbitrary text strings, with the option to verify each change before it is made
- merge parts of one file into another
- create a file that is a subset of another file

SOS is line-oriented, so it usually operates with line-numbered text files. If a file is edited that does not contain line numbers, the editor adds line numbers to the text lines. For most SOS commands, a line number or range of line numbers specifies the text to be operated on. When commanded to insert, delete, move, or copy text, SOS maintains line numbers in ascending order within each page of text.

Advanced features of SOS allow considerable flexibility in searching for a string of text and allow specification of blocks of text by content, or relative position from a known location, instead of by line number. SOS has many operational features under user control.

Initiating and Terminating SOS

SOS is initiated by entering one of the following commands in response to the command language prompt ($):

```
$ EDIT file-spec <RET>
```

If the user were to omit file-spec, SOS would immediately prompt the user for the missing parameter.

To terminate SOS, enter the command E (EXIT) followed by a carriage return after SOS's prompt (*).

```
*E<RET>
[file-spec]
$
```

Upon terminating, SOS writes an output file containing all the modifications made in editing the file. The original file is not changed. The specifier SOS uses for the output file has a version number higher by 1 than the latest version of the original file unless otherwise specified by the user.

SOS Examples

Copy command

1) C300,9000:9500
   Make a copy of lines numbered 9000-9500 and insert the lines after line 300.

Find command

1) Fmore<ESC>
   Search for "more" from the current point in the file.
2) Fmore<ESC>,1:1000
   Search for the first occurrence of "more" in the range of lines from 1 through 1000.

Print command

1) P500:800
   Print lines 500 through 800.
2) P1800
   Print line numbered 1800.

Substitute command

1) Smore<ESC>less<ESC>,500:800
   Change all occurrences of "more" into "less" on lines numbered 500 through 800.

EDT EDITOR

EDT, an interactive text editor, is included with VAX/VMS Version 2.0. This editor lets users enter and manipulate text and programs. EDT, with its extensive HELP facility, is designed to be learned easily by novices. In addition, EDT provides many capabilities that will appeal to advanced users.

What EDT Does

EDT is a powerful text editor that provides:

- both line and character editing facilities
- screen editing and keypad editing on the VT52 and VT100 video terminals
- ability to work on multiple files simultaneously
- a journaling facility, which protects against loss of edits due to system crashes, or loss of carrier on a dial-up line
- an extensive HELP facility
- a start-up command file, which allows a choice of editing options to be set automatically
- a window into a file (on video terminals only) that lets users view changes in file contents immediately

EDT is also supported on hardcopy terminals and video terminals other than the VT52 and VT100.

EDT SPECIAL FEATURES

Editing with a Window

"Window editing" is a valuable feature that lets users edit one 22-line window (screenful) at a time. This feature allows a user to see immediately how the edits made affect his file. The user may position the window anywhere in the file. Window editing is illustrated in Figure 8-1.

Start-up File

When the editor is started, it executes commands from a start-up file. In this file, one can insert editing options such as SET NOKEYPAD and DEFINE KEY. These options take effect automatically when an editing session begins.

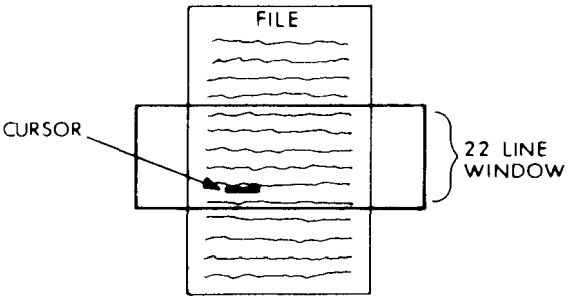


Figure 8-1
Window Editing

HELP Facilities

The HELP facilities on EDT are extensive. Users can get help on general EDT operations by typing HELP. While in keypad mode, users can get help by pressing the help key, which displays a picture of the keypad and provides additional information on each of the keypad keys.

The Keypad

The keypad is a special set of keys to the right of the main keyboard. Figure 8-2 illustrates the functions of the VT100 keypad; the VT52 keypad is similar.

SHIFT	HELP	FIND FNDNXT	UND L DEL EL
COMMAND PAGE	SECT	REPL APPEND	UND W DEL EW
BOTTOM ADVANCE	TOP BACKUP	PASTE CUT	UND C DEL C
WORD	DEL EOL EL	INSCOD CHAR	SUBS
OPEN LINE LINE		RESET SELECT	ENTER

Figure 8-2
VT100 Keypad Functions

Keypad functions allow the user to perform a variety of operations. Furthermore, the function of any keypad key can be changed to meet the needs of the user via the DEFINE command.

The commands in the keypad submode let users alter text or change the cursor position in the file. Keypad functions are available to advance or back up the cursor or move the cursor to the top or bottom of the text. One can also move the cursor any number of characters, words, lines, or pages at a time.

Keypad keys let a user select a string of text and move it elsewhere in any of his files. One can even find the next

occurrence of some text and delete or replace it. There is also a key to press for help messages.

Redefining Keypad Keys

One can redefine any of the keypad keys, and most of the control (CTRL) keys, on VT52 and VT100 terminals. This feature lets the user assign a series of commands to a key; EDT performs these commands when the keys are pressed. Therefore, one can adapt the functions of keypad and CTRL keys to meet special needs.

The SET and SHOW Commands

The SET command, with a variety of qualifiers, controls EDT's editing capabilities. SET controls such screen parameters as line width or lets a user determine the appearance of text, such as changing the window size to less than 22 lines. The SHOW command provides information on the current state of the editor, such as terminal parameters, definitions of keypad keys, and the names of buffers in use during the editing session.

Journal Processing

Journal processing protects the user's work against unlikely system crashes. During an editing session, EDT saves all the terminal input in a journal file. After a crash and recovery, the user may choose to retrieve and execute commands in this saved file with the /RECOVER EDIT command qualifier. In this way the user can recover edited files to the time of the crash.

The EDT CAI Program

Also available with VAX/VMS V2.0 EDT is a Computer-Assisted Instruction (CAI) program on EDT. This interactive program presents the "Introduction to the EDT Editor" minicourse, which demonstrates how to use EDT. The CAI program runs on VT100 terminals and takes about three hours.

EDT Modes of Operation

A "mode" in EDT is a state in which the editor lets a user perform a specific set of functions. EDT has two basic modes of operation: line mode and change mode.

Line mode allows users to establish editing parameters and to display and edit text by range specification. (One can specify a range with such entities as line numbers and character strings.)

One can modify the text with line editing commands such as COPY, SUBSTITUTE, and REPLACE. Or one can move about in the text by using the FIND and TYPE commands, for example, or by pressing the RETURN key.

Change mode lets users operate on such entities as characters, words, sentences, paragraphs, and lines. One can also work with strings of text or delete and move whole pages. EDT lets a user redefine these entities to tailor them to specific applications, which can be as diverse as documentation or programming.

Change mode consists of a set of NOKEYPAD commands. Typing any of these commands lets a user perform useful functions. By typing FNDNXT, for example, one can find the next occurrence of a string of characters.

With VT52 and VT100 terminals, one can also use KEYPAD commands. The set of keypad keys, as well as several CTRL keys, lets the user enter any of the NOKEYPAD commands simply by pressing a key. Users can also redefine the function of these keys.

SLP EDITOR

SLP is the batch-oriented editing program used for source file maintenance. SLP allows updating (deletion, replacement, addition) of lines in an existing file. The SLP command file provides a reliable method of duplicating the changes made to a file, at a later time or on another system.

Input to SLP consists of a correction input file that is to be updated, and command input containing text lines and edit command lines that specify the update operations to be performed. SLP locates lines to be changed by means of locators (line numbers or character strings). Command input normally enters through an indirect file that contains commands and text input lines to be inserted into the file. Alternatively, commands can be entered from the terminal.

SLP output is an optional listing file and an updated copy of the corrected input file. SLP provides an optional audit trail that helps keep track of the update status of each line in the file. The audit trail is provided in the listing and is included permanently in the output file. When a given file is updated with successive versions of an SLP command file, different audit trails may be used to differentiate between changes made at various times.

SLP output qualifiers permit the user to create or suppress an audit trail, eliminate an existing audit trail, specify the length and beginning position of the audit trail, or generate a double-spaced listing.

Initiating and Terminating SLP

SLP is initiated via the command language EDIT command. The normal way to use SLP is to specify an indirect command file that informs SLP what files to process, and indicates what editing changes are to be made to the correction input file. The indirect file can be specified on the same line with the EDIT command, or on a separate line. The indirect file must be created before running SLP. An interactive text editor is normally used to create SLP indirect command files. If both new and old versions of the file exist, the differences utility can be used to create a SLP correction file that will change the old file into the new one.

SLP Input and Output Files

SLP requires two types of input: a correction input file and command input. The correction input file is the source file to be updated using SLP. Command input consists of an initialization line, followed by SLP edit commands that indicate how the file is to be changed.

SLP output consists of a listing file and an output file. The listing file is a copy of the output file with sequence numbers added; it shows the changes SLP makes to the correction input file. The output file is the permanently updated copy of the input file.

Correction Input File

The correction input file is the file to be updated by SLP. It can contain any number of lines of text. When SLP processes the correction input file, it makes the changes specified by SLP edit commands in the output file.

SLP Output File

The SLP output file is the updated input file. All of the updates specified by the command input are inserted in this file. An audit trail, unless suppressed, is applied to lines changed by the update. The numbers generated by SLP for the listing file do not appear in the output file.

LINKER

The VAX/VMS linker is a program development tool that takes the output of language translators (object files or modules), such as the VAX-11 MACRO assembler or the VAX-11 FORTRAN compiler, and produces a file that can be executed on the VAX-11 hardware. This output file is known as an **image**. To write an application in modules, it is necessary to be able to link together the separately compiled modules. The linker is activated by the DCL LINK command, which can be entered interactively or from within a command procedure. Linking consists of three basic operations:

- allocation of virtual memory addresses
- resolution of intermodule symbolic references
- initialization of the contents of a memory image

At the end of a linking operation, the program has virtual memory addresses assigned, has intermodule references resolved, and exists as an executable initialized entity in a disk-resident image file.

The LINK Command

The DCL LINK command provides the interface between the user and the linker. When the user requests the linking of object modules, the command interpreter receives the command and activates the linker.

Virtual Memory Allocation

Language translators do not compute any addresses in the program. At the time of translation, the allocation of virtual address space is undecided. Each object module is relocatable in virtual memory. The reason that language translators cannot allocate virtual memory addresses is that a translator can see only one module at a time: it cannot know how modules interrelate. As a result, it is the linker's function to perform the memory allocation, reference resolution, and image initialization required to form one executable program from a number of object modules.

VAX/VMS language translators use the object language to describe a module to the linker. The output from a translator is an object module consisting of records describing the module to the linker. The language translators define each object module as a number of separate areas called program sections. Some program sections contain data, others contain instructions. Some can be modified during execution, others cannot. Some are accessible to procedures in other modules, others are local to a module. When determining the virtual memory allocation of a program, the linker must consider the attributes of each program section. The linker groups program sections with similar attributes together in virtual memory.

Resolution of Symbolic References

VAX language translators provide the ability to call external procedures by name. They permit the use of other external items such as literals and variables by name. External references have values that are available only to the linker when all the input (e.g., modules and library procedures) is gathered together. The VAX-11 object language provides the ability for a language translator to describe to the linker the external items required by a module. The linker maintains a description of the items of each module that are available to other separately translated modules. In the object language, all of these external items are either references to global symbols or definitions of global symbols.

Image Initialization

After the linker allocates virtual memory and resolves external references, the linker fills in the actual contents of the image. This image initialization consists mainly of copying the binary data and code that was written by the compiler or assembler. However, the linker must perform two additional functions to initialize the image contents:

- It must insert addresses into instructions that refer to externally defined fields. For example, if a module contains an instruction moving FIELDA to FIELDB, and if FIELDB is defined in another module, the linker must determine the virtual address of FIELDB and insert it into the instruction.
- It must compute values that depend on externally defined fields. For example, if a module defines X as being equal to Y plus Z, and if Y and Z are defined in an external module, the linker must compute the value of Y plus Z and insert it in X.

Overview of Linker Interface to Memory Management

The linker describes the virtual address space required for an image in such a way that the image activator function of VAX/VMS can initialize the VAX memory management hardware to place the image in a process virtual address space. When a user requests execution of an image, the image activator obtains a description of the image's virtual address requirements from the image file produced by the linker.

The mechanism used to describe images to VAX/VMS is an image section descriptor. The linker creates an image section description (ISD) for each image section of a shareable or executable image. The header of an image contains the ISDs for the image. With the ISD, memory management can determine the following information about an image section:

- the starting block number of the image section in the image file
- the starting virtual page number in the process's virtual address space to which to map the image section
- characteristics of the image section, e.g., read-only, read/write
- additional control information

Using the information in the ISD, memory management sets the page table and other data structures used to bring process pages into physical memory and to allow sharing in physical memory.

Linker Input

The linker accepts the following types of files as input to a binding operation:

1. Object module files
2. Libraries of object modules
3. Shareable images files
4. Symbol tables from shareable images

Object Module Files

The linker requires as a minimum one object file as input to a binding operation. An object module contains four types of information:

1. Compiled program code and data.
2. Descriptions of program code and data used by the linker in performing relocation and link-time computations.
3. Identification of the object module and its history for use by the librarian and patch utilities.
4. Description of the memory allocation requirements of the module.

Object Module Libraries

The librarian creates and updates object module library files. Each library file contains a catalog of the object modules and global symbols within it. The linker can access modules in such libraries either explicitly or implicitly.

Explicit extraction is performed on the basis of the name of a particular module in the file or by naming the library file and letting the linker extract any modules required to resolve undefined symbols.

Implicit access to object module libraries occurs after all explicitly named input modules have been extracted, and is done by loading modules which contain global symbols that resolve undefined global symbols in the link.

Shareable Image Files

A shareable image is an image that comprises part of a complete program. All references in the shareable image are resolved when the shareable image is created. Shareable images are used as input to a later link to create an executable image.

Shareable Image Symbol Tables

When the linker produces an image file, it appends the symbol table to the file. The symbol table produced by the linker has the same form as an object module. That is, it defines those symbols available to object modules that are outside the set of object modules that produced the shareable image. Such symbols are called universal.

Linker Output

The linker can produce three different types of images.

1. Executable images
2. Shareable images
3. System images

Executable images are the most common. As the name suggests, an executable image is the type run in response to a command given to the command interpreter. The second type, shareable images, is intended for use at link time and, potentially, at run time. At link time, a shareable image can be linked with object modules to produce an executable image. The same shareable image can be shared when executable images bound to it are run. A system image is a special type of image intended for stand-alone operation on the hardware i.e., it does not run under the control of the VAX/VMS operating system.

COMMON RUN TIME PROCEDURE LIBRARY

The VAX-11 Common Run Time Procedure Library (RTL) is composed of a set of general purpose and language-specific VAX procedures which establish a common run time environment for all user programs written in any native mode language. Because all of the language support procedures follow the same programming standards and make nonconflicting assumptions about the execution environment, a user program can be composed of modules written in different languages, including assembly language. Because of the VAX procedure calling standard, each native mode user module can call any other native mode user module or any of the procedures in the Run Time Library.

Most of the VAX-11 Run Time Library is constructed as a separate shareable image which is accessed by users via entry point vectors. This allows:

1. Installation of a new library without the need to relink a user's program.
2. Implementation of new internal algorithms without relinking all user programs.
3. A single copy of the library to be shared by all processes.

Each procedure entry point in the shareable image is at an address defined relative to the base of the shareable section, and will never change, once it has been assigned. New entry points are always added at the end of the list of entry point vectors. The entry point vector contains the procedure entry mask and a transfer of control to the procedure. Use of entry vectors permits a single position-independent copy of the library to be bound to different virtual addresses in processes which are sharing it. Use of entry vectors also permits a new release of the library to be installed without requiring that user images be relinked.

The VAX-11 Run Time Library is designed as a set of modular re-entrant procedures comprising several functional groups. They are:

- a resource allocation group (virtual memory, logical unit number, and event flags)
- a condition handling group (signaling exception conditions and declaring condition handlers)
- a general utility group (data type conversions)
- a mathematical group (single and double precision trigonometric, logarithmic, and exponential functions)
- a language-independent support group (error handling and Record Management Services support functions)
- language-specific support groups (file handling support functions)
- a string handling group (static and dynamic string functions)

Resource Allocation group (LIB$)

The resource allocation group includes all procedures which allow allocation of process-wide resources. Such resources include:

1. Virtual Memory—one procedure to allocate and one to deallocate arbitrary-sized blocks of process virtual memory.

2. Logical Unit Numbers—allow logical unit numbers to be allocated in a modular manner.
3. Event Flags—allow event flags to be allocated in a modular manner.

In most cases, the resource allocation procedures must be used to allocate process-wide resources in order for all library, DIGITAL, and customer-written procedures to work together properly within an image.

Signaling and Condition Handling

The VAX-11 condition handling facility is a collection of library procedures and system services which provides a unified and standardized mechanism for handling errors internally in the operating system, the Run Time Library, and user programs. In some cases, the mechanism is also used to communicate errors across these interfaces. In particular, all error messages are printed using this mechanism. When an error condition is signaled, the process stack is scanned in reverse order. Establishing a handler provides the programmer with some control over fix-up, reporting, and flow of control on errors. It can override the standard error messages in order to give a more suitable application-oriented user interface.

General Utility (LIB$)

General utility procedures are not mandatory in order to use the rest of the library successfully. They are provided for the convenience of the user only. General utility procedures include outputting a record to a logical device (SYS$OUTPUT).

Mathematical Functions (MTH$)

The mathematical library consists of standard procedures to perform common mathematical functions, such as taking the sine of an angle. The standard entry points have one or two call-by-reference input parameters and a single function value. Some frequently used procedures also have call-by-value entry points that are called by the JSB instruction.

Language-Independent Support (OTS$)

The language support libraries support the code generated inline by compilers. As such, most of the procedures are called implicitly as a consequence of a language construct specified by the user, rather than being called explicitly by the user with a CALL statement. Those language support procedures which are independent of higher level language use the facility prefix OTS$.

Language-Specific Support (FOR$. BAS$)

Each of the language support libraries is composed of five principal types of procedures:

- I/O processing procedures
- Language-independent initialization and termination
- System procedures
- Compiled-code support procedures
- error and exception-condition processing procedures

String Processing (STR$)

The string processing procedures allocate and deallocate dynamic strings and perform a number of useful string functions on any class of VAX strings.

System Procedures

VAX-11 programs written in the higher-level languages may call the operating system directly. However, since some languages cannot easily pass arguments in the form that system services require, and some languages use data types that system services cannot properly handle (i.e., dynamic strings), LIB$ routines provide easy access to the operating system directives.

Compiled-Code Support Procedures

These routines complement the compiled code by performing operations too complicated or too cumbersome to perform directly with inline code. Thus, the language support libraries support the code generated by the compiler. For example, division of complex numbers is performed by a library procedure.

Error Processing Procedures

Errors detected by the Run Time Library are indicated by returning an error completion status wherever possible. This is especially true for the general utility library (LIB$). However, the math library and the language support libraries indicate most errors by CALLing the VAX-11 LIB$SIGNAL or LIB$STOP procedures. The LIB$SIGNAL procedures use a condition value as an argument which has an associated error message stored in a system error message file. The condition is signaled to successive procedure activations in the process stack. These procedures may have established handlers to handle the conditions or change the error message. Thus an application can tailor its error messages to its own needs.

VAX-11 SYMBOLIC DEBUGGER

The VAX-11 SYMBOLIC DEBUGGER is a language-independent, interactive program that can be linked with user code written in all native mode languages supported by VAX/VMS. Current languages with which the debugger can be used are: VAX-11 FORTRAN, VAX-11 BASIC, VAX-11 COBOL, VAX-11 BLISS-32, VAX-11 CORAL 66, VAX-11 PASCAL, and the VAX-11 MACRO assembly language. After linking with the user program, the DEBUG facility is operative in the language of the first module of the image file. If it is necessary to alter the language for a later module, the SET LANGUAGE command may be used.

DEBUG enables dynamic examination and modification of the contents of memory locations, which is useful in isolating program errors. Since user program execution is controlled by DEBUG once it is invoked, modifications may be made to the program while it is executing.

The VAX-11 debugger includes many user oriented functions that facilitate the use of the VAX-11 SYMBOLIC DEBUGGER.

- The debugger is interactive—the user maintains control of the program while conversing with the debugger via the terminal.
- The debugger is symbolic—program locations may be referred to by the symbols the user has created in the program. The debugger is also capable of displaying locations as symbolic expressions.
- The debugger supports all native mode languages—the debugger lets the user converse with the program via the source program's language. Furthermore, the user may change languages during the course of a debugging session by means of a simple command.

- The debugger permits a variety of data forms—the user controls the way in which the debugger accepts and displays addresses and data. For example, an address can be represented symbolically, or as a virtual address in decimal, octal, or hexadecimal. Also, data can be represented by symbols, expressions (X+3), VAX-11 MACRO instructions, ASCII character strings, or numeric strings in decimal, octal, or hexadecimal.

DEBUG Commands

DEBUG commands direct the execution of the program and can be entered interactively from a terminal or from an indirect command file. Typically, the DEBUG commands can:

- Specify points at which execution will be suspended, when and if they are encountered, by using the SET BREAK command.
- Trace the sequence of program execution by means of the SET TRACE command. This command establishes tracepoints in the program.
- Display before-and-after values of a location whenever that location is stored into, by means of the SET WATCH command.
- Initiate or resume execution, by means of the GO command or the STEP command.
- Determine the location of breakpoints, tracepoints, and watchpoints by means of the commands SHOW BREAK, SHOW TRACE, and SHOW WATCH, respectively.
- Erase breakpoints, tracepoints, and watchpoints in the program, through use of the CANCEL command.
- Display the contents of memory locations, by using the EXAMINE command.
- Change the value of the contents of memory locations, by using the DEPOSIT command.
- Obtain the value of an expression or the current address of a symbol, or express a numeric value in a different radix, by using the EVALUATE command.
- Call a subroutine at DEBUG time, by means of the CALL command.
- Change values of parameters for LANGUAGE, SCOPE, MODE, and TYPE.
- Specify an arbitrary file name for the DEBUG log file by means of the SET LOG command.
- Control DEBUG I/O at debug time, via the SET OUTPUT command. This includes normal terminal output, log file output, and command file verification.
- Find all current output attributes (VERIFY, TERMINAL and LOG) by using the SHOW OUTPUT command. For more limited needs, a SHOW LOG command is available that displays only the LOG data.
- Instruct DEBUG to take commands from a specified file by means of @filespec.

THE LIBRARIAN UTILITY

Libraries are indexed files that contain frequently used modules of code or text. There are four types of libraries; object, macro, help, and text. The library type indicates the type of module that the library contains. Each library contains indexes that store information regarding the library's content, including type, and location. The librarian is a utility that allows the user easy access to the data stored in libraries.

The librarian may be invoked in one of two ways; via a set of librarian routines that can be called from the user program directly, or interactively via the DCL command LIBRARY issued from the terminal or from within an indirect command file. The DCL LIBRARY command enables the user to replace and maintain modules in an existing library, or to create a new library. The librarian routines enable an executing program to initialize and open a library, and to retrieve, insert, and delete modules.

The four library types are defined as follows:

- Object libraries (file type OLB) contain frequently called routines and are used as input to the linker. The linker searches the object module library whenever it encounters a reference it cannot resolve from the specified input files.
- Macro libraries (file type MLB) contain macro definitions used as input to the MACRO assembler. The assembler searches the macro library whenever it encounters a macro that is not defined in the input source file.
- Help libraries (file type HLB) contain help modules; that is, modules that provide user information concerning a program. The help message can be retreived by calling the appropriate librarian routines.
- Text libraries (file type TLB) contain any sequential record files required by the user program. A user program can call library routines directly to retrieve text modules.

Librarian Routines

The Librarian utility provides a set of 18 user-callable routines that:

- initialize a library
- open a library
- look up a key in a library
- insert a new key in a library
- return the names of the keys
- delete a key and its associated text
- read text records
- write text records

The user program can call the librarian routines using the VAX-11 standard calling sequence supported in all languages producing VAX-11 native mode code.

DCL LIBRARY Command

The LIBRARY command creates or modifies an object, help, text, or a macro library, or inserts, deletes, replaces, or lists modules, macros, or global symbol names in a library.

To invoke the LIBRARY command, enter the following format:

```
LIBRARY        library[file-spec,....]
```

For example, to create an object library named TESTLIB, and insert entries ERRMSG, and STARTUP, the user would proceed as follows:

```
$LIBRARY/CREATE TESTLIB        ERRMSG,STARTUP
```

COMMAND LANGUAGE PROCEDURES

A command procedure is a file containing DCL commands, command or program input data, or both. Command procedures may be used to catalog sequences of commands frequently used during an interactive session or to submit all jobs for batch processing.

In its simplest form, a command procedure consists of one or more command lines that the command interpreter executes. In its most complex form, a command procedure resembles a program written in a high level programming language: it can establish loops and error checking procedures, call other procedures, pass values to other procedures and test values set in other procedures, perform arithmetic calculations and input/output operations, and manipulate character string data.

Passing Parameters to Command Procedures

The user can write generalized command procedures that may perform differently each time they are executed. The command interpreter defines eight special symbols for use as parameters within command procedures. These symbols are named P1, P2, P3...P8; they are all initially equated to null strings. Either numeric or character string values for these parameters may be passed when executing the procedure with the @ command or the SUBMIT command when entering a batch job.

For example, the procedure named EXECUTE contains the following lines:

```
$ IF P2 .EQS. "" THEN $P2:="FORTRAN"
$ 'P2' 'P1'
$ LINK 'P1'
$ RUN 'P1'
```

The command procedure EXECUTE accepts both the language compiler and the user program name as input. If the user executes the procedure with the @ command, the values for the command parameters P1 and P2 would be entered as follows:

```
$ @EXECUTE PAYROLL COBOL
```

In this sample run, the user chose the program name PAYROLL, and the COBOL compiler.

It is also possible to define a symbol as a local symbol, using a single equals sign (=) in an assignment statement. For example, the user might have equated the symbol EXE to the execution command @EXECUTE as follows:

```
$ EXE*CUTE:=@EXECUTE
```

The asterisk (*) specifies that EXE, EXEC, EXECU, etc. are abbreviations of EXECUTE. The minimum abbreviation is three characters (in this case, EXE). A colon (:) in an assignment statement indicates a character string assignment. Now to execute the command procedure the use can enter the following:

```
$ EXE STRESS
```

In this run, STRESS is the user program name and the compiler is the default compiler, FORTRAN (i.e., the second parameter in the EXE command was left blank).

Logical Commands

Normally, the command interpreter executes each command in a command procedure in sequential order, and terminates processing when it reaches the end of the command procedure file. However, by using combinations of the logical commands, the user can alter the flow of execution of the command procedure. By using the IF, GOTO, ON, EXIT, and STOP commands, the user can control the execution sequence, conditionally execute lines, construct loops, and handle errors.

Lexical Functions

The command interpreter recognizes a set of functions, called lexical functions, that return information about character strings and attributes of the current process. Lexical functions may be used in any context in which symbols and expressions are used. Within command procedures, lexical functions are used to translate logical names, perform character string manipulations, and determine the current processing mode of the procedure.

Command Procedure Example

The command procedure described in Figure 8-3, when invoked, locks up a user terminal as being in use. If the current user must leave the terminal for some time and does not wish to have it disturbed, the user can invoke the command procedure INUSE.COM, rendering the terminal inaccessible to any other user not knowing the access password.

This command procedure illustrates several of the powerful features of DCL, including:

- Trapping of the Control-Y function.
- Calling a command procedure from within a command procedure (i.e., @COMMANDS:INUSE.TXT). ERASE, ERASELINE, and TEXT are user-defined symbols that also invoke command procedures.
- Referencing lexical functions (i.e., 'F$TIME, 'F$LOCATE, and 'F$EXTRACT).

Upon invoking the command procedure INUSE.COM:

- The current setting of VERIFY is retrieved via the lexical function 'F$VERIFY, and stored in local variable VER (line 100).
- The procedure will set NOVERIFY (line 200), i.e., the command lines are not echoed on the terminal during execution (if verify was on when the command procedure was invoked, it will be turned on when the procedure exits successfully).
- The address of the password routine (line 900) is stored for later use if Control-Y is typed.
- The address of ERR_HNDLR (line 950) is stored for processing any errors that might occur.

Execution then proceeds with the BEGIN code block (lines 1000-1300). The ERASE procedure (line 1100) is called, which clears the screen of all text. INUSE.COM then calls the command procedure INUSE.TXT (line 1300). This procedure prints in block letters, "IN USE," across the video screen. Execution then proceeds to the LOOP section of code (lines 1500-2300). This block of code retreives the current date and time of day from VMS, using the lexical function 'F$TIME. The date and time of day normally appear as follows:

```
dd-mmm-yyyy hh:mm:ss.cc
```

The 'F$LOCATE and 'F$EXTRACT lexical functions operate upon the date and time of day, reducing the time quantity to hours and seconds only. Therefore the final date and

```
100     $     VER='F$VERIFY()
200     $     SET NOVERIFY
300     $
400     $!     THIS COMMAND PROCEDURE LOCKS A TERMINAL AS BEING IN USE. IT DISABLES
500     $!     CONTROL Y, SETS A CONTROL Y ENTRY POINT AND LOOPS ON A SHOW TIME
600     $!     COMMAND. A CONTROL Y TYPED AT THE TERMINAL WILL TRANSFER CONTROL
700     $!     TO A CHECK FOR A PASSWORD TO EXIT FROM THIS PROCEDURE.
800     $
900     $     ON CONTROL Y THEN $GOTO PASSWORD
950     $     ON ERROR THEN GOTO ERR_HNDLR
1000    $BEGIN:
1100    $     ERASE
1200    $
1300    $     COMMANDS:INUSE.TXT
1400    $
1500    $LOOP:
1600    $     TIMSTR:='F$TIME()
1700    $     DOT='F$LOCATE(".",TIMSTR)
1800    $     DOT=DOT-3                              !SHOW TIME DOWN TO MINUTES
1900    $     TIMSTR:='F$EXTRACT(0,DOT,TIMSTR)
2000    $     TEXT 5 32 "'TIMSTR'"
2100    $     TEXT 1   1
2200    $     WAIT 00:01
2300    $     GOTO LOOP
2400    $
2500    $PASSWORD:
2600    $     TEXT 1   1
2700    $     INQUIRE MAGIC "ENTER THE PASSWORD TO CONTINUE"
2800    $     IF "'MAGIC'" .EQS. "'P1'" THEN $GOTO EXIT
2900    $     IF "'MAGIC'" .EQS. "REFRESH" THEN $GOTO BEGIN
3000    $     ERASELINE 1
3100    $     ERASELINE 2
3200    $     ERASELINE 3
3300    $     ERASELINE 4
3400    $     ERASELINE 5
3500    $     ERASELINE 6
3600    $     ERASELINE 7
3700    $     GOTO LOOP
3800    $
3801    $   ERR_HNDLR:
3802    $     ON ERROR THEN GOTO ERR_HNDLR
3803    $     GOTO PASSWORD
3900    $EXIT:
4000    $     SET CONTROL Y
4100    $     ERASE
4200    $     IF VER THEN $SET VERIFY
4300    $     EXIT
4400    $
4500    $!     END OF INUSE.COM
```

Figure 8-3

Example Command Procedure

time of day appear as:

dd-mmm-yyyy hh:mm

This date and time of day function are printed on the screen above the "IN USE" message. The date and time of day are refreshed once every minute. The PASSWORD code (lines 2500-3700) is entered only if a Control-Y is typed at the terminal while the terminal is locked up. The command procedure prompts for a password. If the entered password matches the initial password (declared by the current user of the terminal), the flow of execution drops to the EXIT code block (lines 3900-4300). If the passwords do not match, execution drops through to line 3000, which clears the top seven lines of the screen.

Another added feature is the REFRESH statement (line 2900). The REFRESH statement directly follows the PASSWORD check statement (line 2900). If the screen gets cluttered with garbage characters, any user may enter a CONTROL Y, and in response to the system prompt for a password (line 2700), type REFRESH. REFRESH is recognized in line 2900, clearing the entire screen of unwanted characters. This is followed by a new IN USE and date and time of day message.

DIFFERENCES UTILITY

By invoking the DIFFERENCES command, the user can determine if two files are identical and, if not, how they dif-

fer. The DIFFERENCES utility compares the contents of two disk files on a record-by-record basis and creates a listing of the records that do not match. By default, the DIFFERENCES utility compares every character in each file, and by default, the DIFFERENCES utility writes the output in ASCII.

VAX-11 RUNOFF

VAX-11 RUNOFF is a document formatter. A RUNOFF-processed document can be updated without extensive retyping because text changes, via the text editors, do not affect the basic design. The input to RUNOFF is a file containing the text of the document and the RUNOFF instructions. Executing in default mode, RUNOFF provides:

- a standard typewriter page size of 8½″ × 11″
- sequential page numbering for every page but the first
- page width of 60 characters
- single spacing
- automatic tab settings for every eighth print position, starting with the ninth column (9,17,25, etc.)
- automatic filling and justifying

The output file is the print-ready document. After RUNOFF has processed the file, the original file remains available for further editing.

VAX-11 RUNOFF contains commands to perform the following functions:

- filling and justifying text
- page formatting
- title formatting
- subject-matter formatting
- graphic, list and note formatting
- index and table of contents
- miscellaneous formatting

Filling and Justifying

RUNOFF commands set left and right margins, so that the user may enter text without concern for line width or variable spacing between words. The RUNOFF program will **fill** and **justify** the text when it is run. Filling is the successive addition of words to a line until one more word would exceed the right margin. RUNOFF justifies the line by expanding the spaces between words to produce an even right margin.

Page Formatting

The page formatting commands control the appearance of each page of output. For example, there are page formatting commands to establish the style and location of chapter headings and subheads. Other page formatting commands engage or disengage page numbering, produce and format titles and subtitles, or force the printer to advance to a new page.

Title Formatting

Title formatting commands provide page, title, and subtitle information for all pages. Such actions as placing only the chapter heading on the first page of a chapter; printing any subtitles of designated words; and determining the number of header levels (up to six) that the document will have are all provided by the title formatting commands.

Subject-Matter Formatting

Subject-matter formatting commands include managing the design and appearance of text, as with ragged right-hand margin, indenting a paragraph, skipping a number of lines, centering the text, underlining, hyphenation, and overstriking. Of course, different parts of the text may be formatted differently, and commands may be combined. To illustrate, a user has the option to have lists justified or to have them with ragged margins.

Index and Table of Contents

RUNOFF has powerful facilities for creating indexes and tables of contents easily. There is a command to generate a one-column index. In addition, the TCX program generates two-column indexes, while the TOC program generates tables of contents. Both TCX and TOC create files that can be edited or can be processed by RUNOFF; this adds great flexibility to the preparation of indexes and tables of contents.

Miscellaneous Formatting

A number of useful RUNOFF commands help the user to re-establish all default values, to add nonprinted comments to the source file, to gather externally located files into the input, and to set time and date.

9 Data Management Facilities

```
ENVIRONMENT DIVISION.
[INPUT-OUTPUT SECTION.]
FILE-CONTROL.
SELECT file-name
    ASSIGN TO device-name-1   [, device-name-2]  . . .
    ; ORGANIZATION IS INDEXED
[ ; ACCESS MODE IS { SEQUENTIAL | RANDOM | DYNAMIC } ]
    ; RECORD KEY IS data-name-1
[ ; ALTERNATE RECORD KEY IS data-name-2  [WITH DUPLICATES] ]

DATA DIVISION.
[FILE SECTION.
[FD file-name
  [ ; BLOCK CONTAINS  [integer-1 TO]  integer-2 { RECORDS | CHARACTERS } ]
  [ ; RECORD CONTAINS  [integer-3 TO]  integer-4 CHARACTERS ]
    ; LABEL { RECORD IS | RECORDS ARE } { STANDARD | OMITTED }
  [ ; DATA { RECORD IS | RECORDS ARE } data-name-3  [, data-name-4]  . . . ]

PROCEDURE DIVISION.
OPEN { INPUT file-name-1   [, file-name-2]  . . .
     | OUTPUT file-name-3  [, file-name-4]  . . .
     | I-O file-name-5     [, file-name-6]  . . . } . . .
```

VAX/VMS data management includes a file system that provides volume structuring and protection, and record management services that provide device-independent access to the VAX peripherals.

The VAX/VMS on-disk structure provides a multilevel hierarchy of named directories and subdirectories. Files can extend across multiple volumes and be as large as the volume set on which they reside. Volumes are mounted to identify them to the system. VAX/VMS also supports multivolume ANSI format magnetic tape files with transparent volume switching.

The VAX/VMS record management input/output system (RMS) provides device-independent access to disks, tapes, unit record equipment, terminals, and mailboxes. RMS allows user and application programs to create, access, and maintain data files with efficiency and economy. Under RMS, records are regarded by the user program as logical data units that are structured and accessed in accordance with application requirements.

RMS provides sequential record access to sequential file organizations, sequential, random, or combined record access to relative file organizations and sequential, random, or a combination using index key access to multikey indexed files. Multikey indexed file processing includes incremental reorganization.

VAX/VMS also supports several other data management facilities: DATATRIEVE, VAX-11 SORT, and the Forms Management System (FMS) utility package.

INTRODUCTION

The operating system's data management services are provided by the following facilities:

- utilities for data and file manipulation and inquiry
- file system
- record management services
- device drivers
- command interpreter

Utilities which VAX offers include the VAX-11 SORT/MERGE for reordering data, DATATRIEVE for data inquiry and report writing, and FMS for screen formatting and forms generation.

The file system provides volume structuring and directory access to disk and magnetic tape files. Programmers can use the file system as a base to build their own record processing system, or they can use the VAX/VMS record management services.

The record management services (RMS) provide device-independent access to all types of I/O peripherals. The RMS procedures enable a program to access records within files, and provide the same programming interface regardless of device characteristics. The system includes utilities for RMS file creation and maintenance.

The device drivers provide the basic I/O device handling for all of the other data management services. Device drivers and their features are described in the Peripherals and Operating System sections.

As described in the Users section, the command interpreter enables a user to reserve devices for exclusive use, set device and directory name defaults, and assign logical names to file specifications. The command interpreter also enables the user to execute file management utilities that provide file copy, transfer, and conversion operations.

The following paragraphs discuss some of the features and functions of the file system, including the file structures, file naming facilities, and the file management utility programs. The remainder of this section describes the record management services programming environment, and utilities for high-level data and file manipulation.

FILE MANAGEMENT

VAX/VMS provides two file structures: one for disk volumes and one for magnetic tape volumes. From the user's point of view, the only differences between the two file structures are those imposed by the capabilities of the media. Volumes are mounted for identification, and files can extend across multiple volumes. The practical limit to file size is that they can be only as large as the volume set on which they reside.

Volume and file protection are based on User Identification Codes (UICs) assigned to accessors and the file or volume. The UICs establish the accessor's relationship to the data structure as owner, the owner's group, the system, or the world (all others). Depending on the relationship, the accessor may or may not have read, write, execute, or delete access to any given file.

Disk volumes are multiuser volumes. They can contain a multilevel directory hierarchy that is defined dynamically by the users of the volume. The on-disk file structure appears to a program to be a virtually contiguous set of blocks. The blocks of the file, however, may be scattered anywhere on a volume. Mapping information is maintained to identify all the blocks constituting a file. Figure 9-1 illustrates the file structure.

Disk files can be extended easily. The blocks of the file are allocated in physically contiguous sets, called **extents**. Users are not required to preallocate space, although they can do so. Users can specify placement on an allocation request, and they can control automatic allocation. For example, when a file is automatically extended, it can be extended by any given number of contiguous blocks. If desired, a file can be created as a contiguous file, in which case it is both virtually and physically contiguous.

The disk structure includes duplicates of its critical volume information. The system detects bad disk blocks dynamically and prevents re-use once the files to which they are allocated are deleted.

Magnetic tape volumes are single-user volumes. Magnetic tape files consist of physically contiguous blocks. Record blocking is under program control. Files have ANSI format labels. VAX/VMS also supports unlabeled (non-file-structured) magnetic tapes.

File Directories and Directory Structures

A directory is a file containing a list of files on a given volume. A directory entry contains the name, type, version, and unique file ID for a particular file. A directory can list files having the same owner UIC or files having different owner UICs. The entries are listed alphabetically.

A disk volume contains at least one directory, called the master file directory. The system manager is responsible for creating a volume's master file directory. The master file directory can (and normally does) contain a list of directory files which form a second level of directories. The second level of directory files can list data files and/or other directory files, called **subdirectories**. Users can create subdirectories within the directories they own. The subdirectories can also list other directory files and/or data files. Figure 9-2 illustrates a multilevel directory structure.

Since directories of files on volumes are files themselves, they are assigned owner UICs and can be protected from certain kinds of access depending on the relationship established by an accessor's UIC. In the special case of directory files, the file protection fields control an accessor's ability to:

- look up files
- enter new files in the directory, including new versions of existing files
- remove files from the directory

File Specifications

A **file specification** identifies which file is to be used in a file processing operation. Programs use file specifications to identify the file they want to create, access, delete, or extend, and users supply the command interpreter with a file specification to identify the file they want to edit, compile, copy, delete, etc. A complete file specification is a well-defined character string composed of the following fields:

- *Node Name* — The node of the network in which the volume containing the file is stored. The node name is

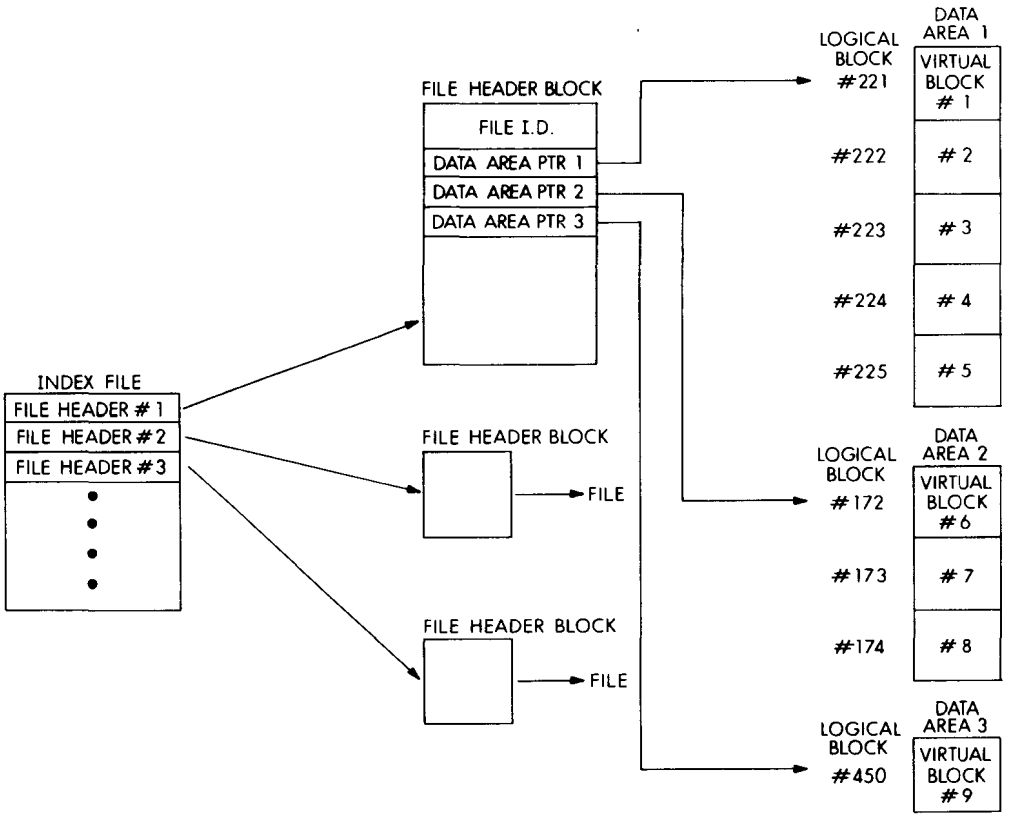


Figure 9-1

Disk File Structure

followed by two colons (::) to delimit it from the remainder of the file specification.

- *Device Name* — The device on which the volume containing the file is mounted. The device name is followed by a single colon (:) to delimit it from the remainder of the file specification.
- *Directory Name* — The directory in which the file is listed. A directory name begins with an opening bracket (< or [) and ends with a closing bracket (> or]). If the file is listed in a subdirectory, the directories to be searched are listed in the desired search order, with the names separated by periods, e.g.:

  [name1.name2.name3]

- *File Name* — The user-assigned name of the file.
- *File Type* — The type identification for the file. The type is preceded by a period (.) to delimit it from the remainder of the file specification.
- *File Version* — the generation number of the file. The file version is preceded by a semicolon (;) or period (.) to delimit it from the remainder of the file specification.

For example, a complete file specification might be:

NODE47::DBA1:[JONES]HANOI.FOR;2

In this case, NODE47 is the name of the network node, DBA1 is the name of the device (DB for disk pack device, A for disk controller, 1 for drive unit number), [JONES] is the directory name, HANOI is the file name, FOR is the file type (meaning that the file is a FORTRAN source file), and 2 is the version number.

Neither programs nor command language users need to provide a complete file specification to identify files. The system applies defaults to most fields of a file specification when they are not present. For example, if the node name is not present, the node is assumed to be the node on which the program is executing. If the version number is not present, the version is always assumed to be the latest version. Device name and directory name defaults for users and the programs they execute are supplied by the system manager in the user authorization file, and users can change the standard defaults at any time during their session on the system.

Some commands (such as COPY, PRINT, and DELETE) accept a wild card in one or more fields of a file specification. A wild card is an asterisk appearing in a file specification field and it means "all."

File specifications also apply to non-file-structured devices such as line printers, card readers, and terminals. In these cases, however, the user or program needs to supply only the node name and device name, as appropriate.

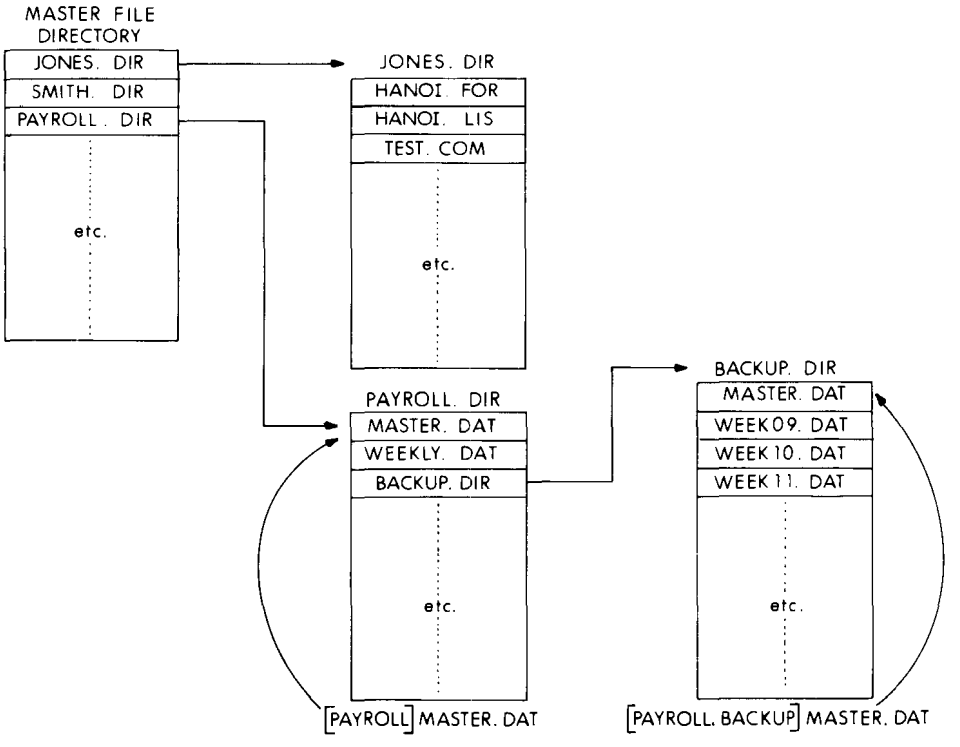


Figure 9-2

Multilevel Directory Structure

Logical File Naming

To provide both system and device independence, users and programs are not limited to identifying files by their file specifications. They can use **logical names** in place of a complete file specification, or in place of a portion of a file specification. For example, a user can assign a logical name to the left-most three fields of a file specification:

$ ASSIGN NODE47::DBA4:[JONES] *to* VOL

And then use the logical name VOL in a subsequent command:

$ TYPE VOL:HANOI.FOR

Defaults also apply when translating logical names, so that the user could have made the assignment:

$ ASSIGN NODE47::[JONES] *to* VOL

In this case, the user's default device name would be used to derive the complete file specification.

Logical name assignments can be made on a process, group, or system-wide basis. Logical names can also be recursive, that is, a logical name can be assigned to another logical name, or to a logical name and a portion of a file specification.

For example, suppose a company's weekly payroll production run includes an application program that uses the current week's payroll changes data file. That data file may be located in the directory named [PAYROLL] one week, or in the payroll backup subdirectory, [PAYROLL.BACKUP], another week. The volume on which the file is stored may be mounted on disk pack drive unit number 1 one week, or on unit 2 another week.

The application programmer can write the program without knowing which directory the data file is listed in, or which device the volume is mounted on. A series of logical name assignments provides the complete file specification. The assignments are the responsibility of the people who know what directory the file is listed in, and what drive the volume is mounted on.

In the example shown in Figure 9-3, the application program contains an OPEN statement for the payroll data file using the logical name WEEKLY_PAYROLL_CHANGES (note that underscore is a legal character). The application systems designer has created a command procedure file called PAYRUN that controls the production run. The command procedure file includes a logical name assignment that obtains the file name as a parameter supplied by the operator or production clerk who starts the production run. The logical name used by the application program is given a value that consists of another logical name (WEEKLY_PAYROLL) and the file name and type specifications.

To complete the series of logical name assignments, the payroll group operations manager makes a group-wide logical name assignment: the payroll data files this week are stored in the PAYROLL.BACKUP subdirectory. The logical name assignment provides the directory name, using another logical name (PAY_PACK) known to the oper-

Application Programmmer:

```
OPEN ("WEEKLY_PAYROLL_CHANGES")
```

Application System Programmer:

```
Command Procedure: PAY_RUN.COM
accepts one parameter (P1): Week Number
$ ASSIGN WEEKLY_PAYROLL:'P1'.WPY     WEEKLY_PAYROLL_CHANGES
$ RUN APPLICATION
```

Production Clerk:

```
$ @PAY_RUN WEEK09
```

Payroll Group Operations Manager:

```
$ ASSIGN/GROUP PAY_PACK:[PAYROLL.BACKUP]     WEEKLY_PAYROLL
```

Local Operator:

```
$ ASSIGN/SYSTEM DBA2:     PAY_PACK
```

Figure 9-3

Logical Naming

ator who mounts the payroll data files volume. The operator makes the system-wide logical name assignment when mounting the pack before the production run. Given the assignments shown in the example, the logical name used to open the file is translated to:

DBA2:[PAYROLL.BACKUP]WEEK09.WPY

(The local system node name and the latest version number are used as defaults to complete the file specification.) Should the directory name change, or the pack be mounted on another device that day, the only changes made are the logical name assignments. There is no need to modify either the application program or the command procedure controlling the production run.

File Management

The VAX/VMS system includes many services that aid in data management and maintenance. Some of these are described in the following paragraphs.

Sorting Files — The SORT/MERGE program allows the user to rearrange, delete, and reformat records in a file. The user can arrange the records in the ascending or descending sequence of one or more fields within the records for subsequent sequential processing. SORT can also create several different index files for accessing a file according to these indexes without reordering the data itself.

Comparing Files — A file differences command contrasts two files by automatically aligning matching text, and optionally ignoring comments, empty records, trailing blanks, or multiple blanks. The output can be a file-by-file list of differences, an interleaved list of differences, a list with change bars, or a batch editor command input file.

Backing Up Files and Volumes — The Disk Save and Compress (DSC) utility enables a user to back up entire disk volumes to magnetic tape or to other disks. When backing up disk volumes to other disk volumes, or restoring disk volumes from magnetic tape, DSC combines unused blocks on disks into contiguous areas.

Verifying File Structures — The file verification utility checks the consistency and accuracy of the file structure on a Files-11 disk volume. It can also display the number of available blocks in a volume, locate files that could not otherwise be accessed, and list the names of files on the volume.

Bad Block Locator — The bad block locator utility determines the number and location of bad blocks on Files-11 disk volumes and stores this information in the bad block file on the volume so that the blocks can not be allocated. Running this utility before initializing a Files-11 volume is useful in ensuring a disk's integrity.

RMS Utilities — The record management services procedures are complemented by a number of utilities designed especially for RMS file creation and maintenance. They allow the user to:

- create an RMS file and define the attributes of the file
- list the attributes of a single file or a group of files, or list the contents of a backup magnetic tape
- convert a file with any file organization or record format to a file with any other file organization or record format
- back up a single file or group of files in a compact format (optionally by creation or revision date)
- restore files previously backed up (optionally by creation or revision date)

RECORD MANAGEMENT SERVICES

The record management services (RMS) are a set of system procedures that provide efficient and flexible facilities for data storage, retrieval, and modification. When writing programs, the user can select processing methods from among the RMS file organizations and accessing techniques. The following sections discuss RMS:

- file organizations
- file attributes
- program operations
- run-time environment

The manner in which RMS builds a file is called its organization. RMS provides three file organizations:

- sequential
- relative
- indexed

All three file organizations are available in both compatibility mode (using RMS-11) and in native mode (using VAX-11 RMS).

The organization of a file establishes the techniques one can use to retrieve and store data in the file. These techniques are known as record access modes. The record access modes that RMS supports are:

- sequential
- random
- Record's File Address (RFA)

An application program or a RMS utility can be used when creating a RMS file to specify the organization and characteristics of the file. Among the attributes specified are:

- storage medium
- file name and protection specifications
- record format and size
- file allocation information

After RMS creates a file according to the specified attributes, application programs can store, retrieve and modify data. These program operations take place on the logical records in a file or the blocks comprising the file.

RMS FILE ORGANIZATIONS

A file is a collection of related information. For example, a file might contain a company's personnel information (employee names, addresses, job titles). Within this file, the information is divided into records. All the information on a single employee might constitute a single record. Each record in the personnel file would be subdivided into discrete pieces of information known as fields. The user defines the number, locations within the record, and logical interpretations of these fields.

The user can completely control the grouping of fields into records and records into files. The relationship among fields and records is embedded in the logic of the programs. RMS does not know the logical relationships that exist within the information in the files.

RMS ensures that every record written into a file can subsequently be retrieved and passed to a requesting program as a single logical unit of data. The structure, or organization, of a file establishes the manner in which RMS stores and retrieves records. The way a program requests the storage or retrieval of records is known as the record access mode. The organization of a file determines which record access modes can be used.

Sequential File Organization

In sequential file organization, records appear in consecutive sequence. The order in which records appear is the order in which the records were originally written to the file by an application program or RMS utility. Sequential organization is the only file organization permitted for magnetic tape and unit record devices. Most VAX/VMS system utilities that deal with files, deal with sequentially organized files. All system editors and language processors, for instance, operate on sequentially organized files. Figure 9-4 illustrates sequential file organization.

Relative File Organization

When relative organization is selected, RMS structures a file as a series of fixed-size record cells. Cell size is based on the maximum length permitted for a record in the file. These cells are numbered from 1 (the first) to n (the last). A cell's number represents its location relative to the beginning of the file.

Each cell in a relative file can contain a single record. There is no requirement, however, that every cell contain a record. Empty cells can be interspersed among cells containing records. Figure 9-5 illustrates a relative file organization.

Since cell numbers in a relative file are unique, they can be used to identify both a cell and the record (if any) occupying that cell. Thus, record number 1 occupies the first cell in the file, record number 17 occupies the seventeenth cell, and so forth. When a cell number is used to identify a record, it is also known as a relative record number.

Indexed File Organization

The location of records in indexed file organization is

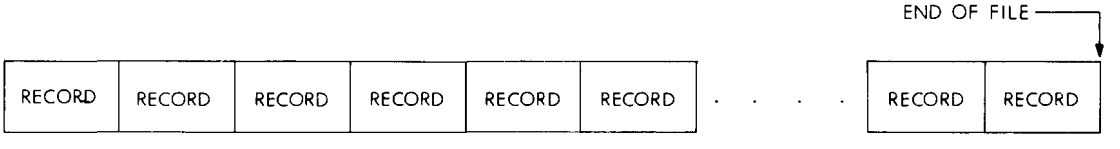


Figure 9-4

Sequential File Organization

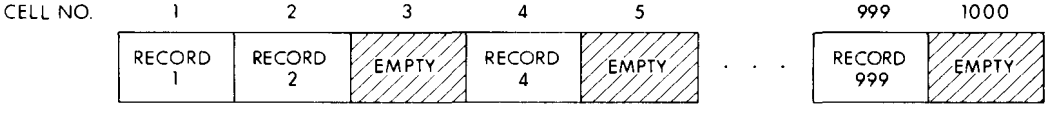


Figure 9-5

Relative File Organization

transparent to the program. RMS completely controls the placement of records in an indexed file. The presence of keys in the records of the file governs this placement.

A key is a field present in every record of an indexed file. The location and length of this field are identical in all records. When creating an indexed file, the user decides which field or fields in the file's records are to be a key. Selecting such fields indicates to RMS that the contents (i.e., key value) of those fields in any particular record written to the file can be used by a program to identify that record for subsequent retrieval.

At least one key must be defined for an indexed file: the primary key. Optionally, additional keys or alternate keys can be defined. An alternate key value can also be used as a means of identifying a record for retrieval.

As programs write records into an indexed file, RMS builds a tree-structured table known as an index. An index consists of a series of entries containing a key value copied from a record that a program wrote into the file. Stored with each key value is a pointer to the location in the file of the record from which the value was copied. RMS builds and maintains a separate index for each key defined for the file. Each index is stored in the file. Thus, every indexed file contains at least one index, the primary key index. Figure 9-6 illustrates an indexed file organization with a primary key. When alternate keys are defined, RMS builds and stores an additional index for each alternate key.

RMS RECORD ACCESS MODES

The methods of retrieving and storing records in a file are called record access modes. A different record access mode can be used to process records within the file each time it is opened. A program can also change record access mode during the processing of a file. RMS permits only certain combinations of file organization and record access mode. Table 9-1 lists these combinations.

Sequential Record Access Mode

Sequential record access mode can be used to access all RMS files and all record-oriented devices, including mailboxes. Sequential record access means that records are retrieved or written in the sequence established by the organization of the file.

Sequential Access to Sequential Files — When using sequential record access mode in a sequentially organized file, physical arrangement establishes the order in which records are retrieved. To read a particular record in a file, say the fifteenth record, a program must open the file and access the first fourteen records before accessing the desired record. Thus each record in a sequential file can be retrieved only by first accessing all records that physically precede it. Similarly, once a program has retrieved the fifteenth record, it can read all the remaining records (from the sixteenth on) in physical sequence. It cannot, however, read any preceding record without closing and reopening the file and beginning again with the first record.

Sequential Record Access to Relative Files — During the sequential access of records in the relative file organization, the contents of the record cells in the file establish the order in which a program processes records. RMS recognizes whether successively numbered record cells are empty or contain records.

When a program issues read requests in sequential record access mode for a relative file, RMS ignores empty record cells and searches successive cells for the first one containing a record. When a program adds new records in sequential record access mode to a relative file, RMS places a record in the cell whose relative number is one higher than the relative number of the previous request, as long as that cell does not already contain a record. RMS allows a program to write new records only into empty cells in the file.

Sequential Record Access to Indexed Files — A program can use the sequential record access mode to retrieve rec-

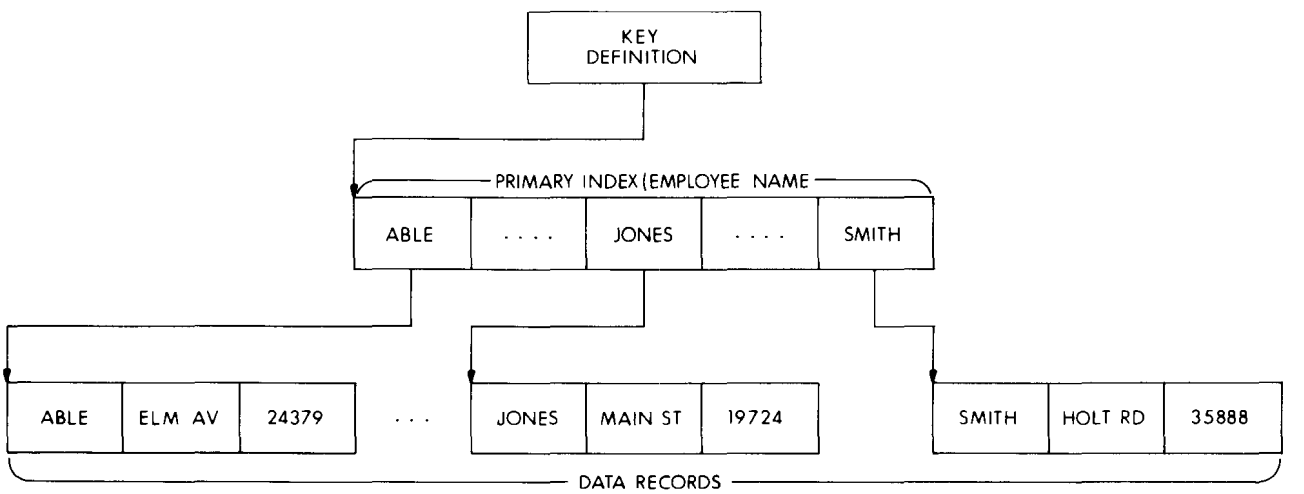


Figure 9-6

Indexed File Organization

Table 9-1
Record Access Modes and File Organizations

File Organization	Record Access Mode			
	Sequential	Random		RFA
		Record #		Key Value
Sequential	Yes	Yes[2]	No	Yes[1]
Relative[1]	Yes	Yes	No	Yes
Indexed[1]	Yes	No	Yes	Yes

1 Disk files only.
2 Fixed length record format disk files only.

ords from an indexed file in the order represented by any index. The entries in an index are arranged in ascending order by key values. If more than one key is defined for the file, each separate index associated with a key represents a different logical ordering of the records in the file.

When reading records in sequential record access mode from an indexed file, a program initially specifies a key (primary key, first alternate key, second alternate key, etc.) to RMS. Thereafter, RMS uses the index associated with that specified key to retrieve records in the sequence represented by the entries in the index. Each successive record RMS returns in response to a read request contains a value in the specified key field that is equal to or greater than that of the previous record returned.

When writing records to an indexed file, RMS uses the definition of the primary key field to place the record in the file.

Random Record Access Mode
In random record access mode, the program establishes the order in which records are processed. Each program request for access to a record operates independently of the previous record accessed. Each request in random record access mode identifies the particular record of interest. Successive requests in random mode can identify and access records anywhere in the file.

Random Record Access to Sequential Files — Native programs can access sequential files on disk using relative record number to randomly locate a record, provided that the records are in fixed-length record format.

Random Record Access to Relative Files — Programs can read or write records in a relative file by specifying the relative record number. RMS interprets each number as the corresponding cell in the file. A program can read records at random by successively requesting, for example, record number 47, record number 11, record number 31, and so forth. If no record exists in a specified cell, RMS notifies the requesting program. Similarly, a program can store records in a relative file by identifying the cell in the file that a record is to occupy. If a program attempts to write a new record in a cell already containing a record, RMS notifies the program.

Random Record Access to Indexed Files — For indexed files, a key value rather than a relative record number identifies the record. Each program read request in random record access mode specifies a key value and the index (primary index, first alternate index, second alternate index, etc.) that RMS must search. When RMS finds the key value in the specified index, it reads the record that the index entry points to and passes the record to the user program.

Program requests to write records randomly in an indexed file do not require the separate specification of a key value. All key values (primary and, if any, alternate key values) are in the record itself. When an indexed file is opened, RMS retrieves all definitions stored in the file. RMS knows the location and length of each key field in a record. Before writing a record into the file, RMS examines the values contained in the key fields and creates new entries in the indexes. In this way RMS ensures that the record can be retrieved by any of its key values.

Record's File Address (RFA) Record Access Mode
Record's File Address (RFA) record access mode can be used to retrieve records in any file organization as long as the file resides on a disk volume. Like random record access mode, RFA record access allows a specific record to be identified for retrieval, using the record's unique address. The actual format of this address depends on the organization of the file.

After every successful read or write operation, RMS returns the RFA of the subject record to the program. The program can then save this RFA to use again to retrieve the same record. It is not required that this RFA be used only during the current execution of the program. RFAs can be saved and used at any subsequent time.

Dynamic Access
Dynamic access is not strictly an access mode. It is the ability to switch from one record access mode to another while processing a file. For example, a program can access a record randomly, then switch to sequential record access mode for processing subsequent records. There is no limitation on the number of times such switching can occur. The only limitation is that the file organization must support the record access mode selected.

FILE AND RECORD ATTRIBUTES

When creating an RMS file, a program or user defines its logical and physical characteristics, or attributes. These characteristics are defined by source language statements in an application program or by an RMS utility. The program or user assigns the file a name, the owner's User Identification Code, and a protection code, and selects the file organization. The program or user also defines or selects other attributes, including:

- device
- file size
- file location
- record format and size
- keys (for indexed files only)

Selection of device is related to the organization of the file. Sequential files can be created on Files-11 disk volumes or ANSI magnetic tape volumes. Sequential files can also be read from mailboxes, terminals, and card readers, and written to mailboxes, terminals, and line printers. Relative and indexed files can be created on Files-11 disk volumes.

The logical limit on file size is $2^{31}-1$ blocks, with a more realistic limit being the volume set on which a file can reside. When creating an RMS file on a disk volume, the user can specify an initial allocation size. If no file size is given, RMS allocates the minimum amount of storage needed to contain the defined attributes of the file. The initial size can be extended dynamically. The user can let RMS locate the file, or the user can allocate the file to specific locations on the disk to optimize disk access time. The file's starting location can be specified optionally using a volume-relative block number, or a physical cylinder address.

When creating a file on a magnetic tape volume, a user or program does not specify an initial allocation size. The blocks are simply written one after another down the tape, beginning after the last file, if any, written on the tape. Once a tape file has been created, another file can replace it or be appended to it, but all subsequent files on the tape, if any, are lost.

Record Formats

The user provides the format and maximum size specifications for the records the file will contain. The specified format establishes how each record appears in the file. The size specification allows RMS to verify that records written into the file do not exceed the length specified when the file was created.

Fixed length record format refers to records of a file that are all equal in size. Each record occupies an identical amount of space in the file. All file organizations support fixed length record format.

Variable-length record format records can be either equal or unequal in length. All file organizations support variable-length record format. RMS prefixes a count field to each variable-length record it writes. The count field describes the length (in bytes) of the record. RMS removes this count field before it passes a record to the program. RMS produces two types of count fields, depending on the storage medium on which the file resides:

- Variable-length records in files on Files-11 disk volumes have a 2-byte binary count field preceding the data field portion of each record. The specified size excludes the count field.
- Variable-length records on ANSI magnetic tapes have 4-character decimal count fields preceding the data portion of each record. The specified size includes the count field. In the context of ANSI tapes, this record format is known as D format.

Variable-with-fixed-control (VFC) records consist of two distinct parts, the fixed control area and a variable-length data record. Although stored together, the two parts are returned to the program separately when the record is read. The size of the fixed control area is identical for all records of the file. The contents of the fixed control area are completely under the control of the program and can be used for any purpose. For example, fixed control areas can be used to store the identifier (relative record number or RFA) of related records. Indexed file organizations do not support VFC record format.

Key Definitions for Indexed Files

To define a key for an indexed file, the user specifies the position and length of particular data fields within the records. At least one key, the primary key, must be defined for an indexed file. Additionally, up to 254 alternate keys can be defined. In general, most files have two or three keys. Because indexes require storage space and RMS updates indexes as records are added or modified, no more than six to eight keys should be defined where storage space or access time is important.

Each primary and alternate key represents from 1 to 255 bytes in each record of the file. RMS permits six key field data types.

- string
- signed 15-bit integer
- unsigned 16-bit binary
- signed 31-bit integer
- unsigned 32-bit binary
- packed decimal

The string key field can be composed of simple or segmented keys. A simple key is a single, contiguous string of characters in the record; in other words, a single field. A segmented key, however, can consist of from two to eight fields within records. These fields need not be contiguous. When processing records that contain segmented keys, RMS treats the separate fields (segments) as a logically contiguous character string. The integer, binary, and packed decimal data types can only be simple keys.

When defining keys at file creation time, two characteristics for each key can be specified:

- duplicate key values are or are not allowed
- key value can or cannot change

When duplicate key values are allowed, more than one record can have the same value in a given key. For example, the creator of a personnel file could define the department name field as an alternate key. As programs wrote records into the file, the alternate index for the department name key field would contain multiple entries for each key value (e.g., PAYROLL, SALES, ADMINISTRATION), since departments are composed of more than one employee. When such duplication occurs, RMS stores the records so that they can be retrieved in first-in/first-out (FIFO) order.

If key values can change, records can be read and then written back into the file with a modified key value. For example, this specification would allow a program to access a record in the personnel file and change the contents of a department name field to reflect the transfer of an employee from one department to another. This characteristic can be specified only for alternate keys. If key values can change, the user must also specify that the duplicate

key values are allowed. If the primary key value can change, the user may not change the record length.

Figures 9-7 and 9-8 show excerpts from a COBOL program which operates upon an indexed customer information file via the dynamic access method. The program searches through the file and generates various reports based upon the customer's financial status and additional input typed in by the user at the terminal. In Figure 9-7, the program describes the organization of the file and specifies the access method to be used. In Figure 9-8, the program searches for the first non-zero customer number. Using the "approximate key" match facility (greater than), the program searches for the first non-zero customer. When RMS has located the first non-zero customer number, the program changes access method and the file is read sequentially.

```
INPUT-OUTPUT SECTION.
FILE-CONTROL.
SELECT CUSTOMER-FILE
        ASSIGN TO "CUSTOM.DAT"
        ORGANIZATION IS INDEXED
        ACCESS MODE IS DYNAMIC
        RECORD KEY IS CUST-CUSTOMER-NUMBER
        ALTERNATE RECORD IS KEY IS CUST-CUSTOMER-NAME
        FILE STATUS IS CUSTOMER-FILE-STATUS.
```

Figure 9-7

ISAM File Description

```
        OPEN INPUT CUSTOMER-FILE.
        MOVE "000000" TO CUST-CUST-NUMBER.
        START CUSTOMER-FILE.
              KEY IS > CUST-CUST-NUMBER.
        OPEN OUTPUT STATEMENT-REPORT.
**************************************************
MAINLINE SECTION.
SBEGIN.
        READ CUSTOMER-FILE NEXT
              AT END
                    GO TO END-PROCESS.
        ADD 1 TO RECORD-COUNT.
```

Figure 9-8

Dynamic Access Processing

PROGRAM OPERATIONS ON RMS FILES

After RMS has created a file according to the user's description of file characteristics, a program can access the file and store and retrieve data.

When a program accesses the file as a logical structure (i.e., a sequential, relative, or indexed file), it uses record I/O operations such as add, update, and delete record. The organization of the file determines the types of record operations permitted.

If the record accessing capabilities of RMS are not used, programs can access the file as an array of virtual blocks. To process a file at this level, programs use a type of access known as block I/O.

File Processing

At the file level, that is, independent of record processing, a program can:

- create a file
- open an existing file
- modify file attributes
- extend a file
- close the file
- delete a file

Once a program has opened a file for the first time, it has access to the unique internal ID for the file. If the program intends to open the file subsequently, it can use that internal ID to open the file and avoid any directory search.

Record I/O Processing

The organization of a file, defined when the file is created, determines the types of operations that the program can perform on records. Depending on file organization, RMS permits a program to perform the following record operations:

- Read a record. RMS returns an existing record within the file to the program.
- Write a record. RMS adds a new record that the program constructs to the file. The new record cannot replace an already existing record.
- Find a record. RMS locates an existing record in the file. It does not return the record to the program, but establishes a new current position in the file.
- Delete a record. RMS removes an existing record from the file. The delete record operation is not valid for the sequential file organization.
- Update a record. The program modifies the contents of a record in the file. RMS writes the modified record into the file, replacing the old record. The update record operation is not valid for sequential file organizations, except for sequentially organized disk files.

Sequential File Record I/O — In a sequential file organization, a program can read existing records from the file using sequential, RFA, or, if the file contains fixed-length records, random record access mode. New records can be added only to the end of the file and only through the use of sequential or random record access mode.

The find operation is supported in both sequential and RFA record access modes. In sequential record access mode the program can use a find operation to skip records. In RFA record access mode, the program can use the find operation to establish a random starting point in the file for sequential read operations.

The sequential file organization does not support the delete operation, since the structure of the file requires that records be adjacent in and across virtual blocks. A program can, however, update existing records in sequential disk files as long as the modification of a record does not alter its size.

Relative File Record I/O — Relative file organization permits programs greater flexibility in performing record operations than does sequential organization. A program can read existing records from the file using sequential, random, or RFA record access mode.

New records can be sequentially or randomly written as long as the intended record cell does not already contain a record. Similarly, any record access mode can be used to perform a find operation. After a record has been found or read, RMS permits the delete operation. Once a record has been deleted, the record cell is available for a new record. A program can also update records in the file. If the format of the records is variable, update operations can modify record length up to the maximum size specified when the file was created.

Indexed File Record I/O — Indexed file organization provides the greatest flexibility in performing record operations. A program can read existing records from the file in sequential, RFA, or random record access mode. When reading records in random record access mode, the program can choose one of four types of matches that RMS performs using the program-provided key value. The four types of matches are:

- exact key match
- approximate key match
- generic key match
- approximate and generic key match

Exact key match requires that the contents of the key in the record retrieved precisely match the key value specified in the program read operation.

The approximate match facility allows the program to select either of the following relationships between the key of the record retrieved and the key value specified by the program:

- equal to or greater than
- greater than

The advantage of this kind of match is that if the requested key value does not exist in any record of the file, RMS returns the record that contains the next higher key value. This allows the program to retrieve records without knowing an exact key value.

Generic key match means that the program need specify only an initial portion of the key value. RMS returns to the program the first occurrence of a record whose key contains a value beginning with those characters. This allows the program to retrieve a class of records, for example, all employee records in the personnel file with a name field beginning with M.

The final type of key match combines both generic and approximate facilities. The program specifies only an initial portion of the key value, as with generic match. Additionally, a program specifies that the key data field of the record retrieved must be either:

- equal to or greater than the program-supplied value
- greater than the program-supplied value

RMS also allows any number of new records to be written into an indexed file. It rejects a write operation only if the value contained in a key of the record violates a user-defined key characteristic (e.g., duplicate key values not permitted).

The find operation, similar to the read operation, can be performed in sequential, RFA, or random record access mode. When finding records in random record access mode, the program can specify any one of the four types of key matches provided for read operations.

In addition to read, write, and find operations, the program can delete any record in an indexed file and update any record. The only restriction RMS applies during an update operation is that the contents of the modified record must not violate any user-defined key characteristic (e.g., key values cannot change and duplicate key values are not permitted).

Block I/O Processing

Block I/O allows a program to bypass the record processing capabilities of RMS entirely. Rather than performing record operations through the use of supported record access modes, a program can process a file as a structure consisting solely of blocks.

Using block I/O, a program reads or writes blocks by identifying a starting virtual block number in the file and a transfer length. Regardless of the organization of the file, RMS accesses the identified block or blocks on behalf of the program.

Since RMS files, particularly relative and indexed files, contain internal information meaningful only to RMS itself, DIGITAL does not recommend that a file be modified by using block I/O. The presence of the block I/O facility, however, does permit user-created record formats on a Files-11 disk volume or ANSI magnetic tape volume.

RMS RUN TIME ENVIRONMENT

The environment within which a program processes RMS files at run time has two levels, the file processing level and the record processing level.

At the file processing level, RMS and the operating system provide an environment permitting concurrently executing programs to share access to the same file. RMS ascertains the amount of sharing permissible from information provided by the programs themselves. Additionally, at the file processing level, RMS provides facilities allowing programs to exercise as little or as much control over buffer space requirements for file processing as desired.

At the record processing level, RMS allows programs to access records in a file through one or more record access streams. Each record access stream represents an independent and simultaneously active series of record operations directed toward the file. Within each stream, programs can perform record operations synchronously or asynchronously. That is, RMS allows programs to choose between receiving control only after a record operation request has been satisfied (synchronous operation) or receiving control before the request has been satisfied (asynchronous operation).

For both synchronous and asynchronous record operations, RMS provides two record transfer modes, move mode and locate mode. Move mode causes RMS to copy a record to/from an I/O buffer from/to a program-provided location. Locate mode allows programs to process retrieved records directly in an I/O buffer.

Run Time File Processing

RMS allows executing programs to share files rather than requiring them to process files serially. The manner in which a file can be shared depends on the organization of the file. Program-provided information further establishes the degree of sharing of a particular file.

File Organization and Sharing — With the exception of magnetic tape files, which cannot be shared, an RMS file can be shared by any number of programs that are reading, but not writing, the file. Sequential disk files can be shared by multiple readers and multiple writers, but they are responsible for any record locking required to handle multiple readers and writers properly.

Program Sharing Information — A program specifies what kind of sharing actually occurs at run time. The user controls the sharing of a file through information the program provides RMS when it opens the file. First, a program must declare what operations (e.g., read, write, delete, update) it intends to perform on the file. Second, a program must specify whether other programs can read the file or both read and write the file concurrently with this program.

These two types of information allow RMS to determine if multiple user programs can access a file at the same time. Whenever a program's sharing information is compatible with the corresponding information another program provides, both programs can access the file concurrently.

Buffer Handling — To a program, record processing under RMS appears as the direct movement of records between a file and the program itself. Transparently to the program, however, RMS reads or writes the blocks of a file into or from internal memory areas known as I/O buffers. Records within these buffers are then made available to the program. Users can control the number and size of buffers. For sequential record access, users can choose an optional I/O read-ahead and write-behind buffer management. For magnetic tape file access, they can control the number of buffers for multiple buffering. For sequential disk files, users can specify the number of blocks that are to be transferred whenever RMS performs an I/O operation.

Run Time Record Processing

After opening a file, a program can access records in the file through the RMS record processing environment. This environment provides three facilities:

- record access streams
- synchronous or asynchronous record operations
- record transfer modes

Record Access Streams — In the record processing environment, a program accesses records in a file through a record access stream. A record access stream is a serial sequence of record operation requests. For example, a program can issue a read request for a particular record, receive the record from RMS, modify the contents of the record, and then issue an update request that causes RMS to write the record back into the file. The sequence of read and update record operation requests can then be performed for a different record, or other record operations can be performed, again in a serial fashion. Thus, within a record access stream, there is at most one record being processed at any time.

For relative and indexed files, RMS permits a program to establish multiple record access streams for record operations to the same file. The presence of such multiple record access streams allows programs to process in parallel more than one record of a file. Each stream represents an independent and concurrently active sequence of record operations.

As an example of multiple record access streams, a program could open an indexed file and establish two record access streams to the file. The program could use one record access stream to access records in the file in random access mode through the primary index. At the same time, the program could use the second record access stream to access records sequentially in the order specified by an alternate index.

Synchronous and Asynchronous Record Operations — Within each record access stream, a program can perform any record operation either synchronously or asynchronously. When a record operation is performed synchronously, RMS returns control to a program only after the record operation request has been satisfied (e.g., a record has been read and passed on to the program).

If the programming language allows asynchronous processing, RMS can return control to a program before the

record operation request has been satisfied. A program can use the time required for the physical transfer to perform other computations. A program cannot, however, issue a second record operation through the same stream until the first record operation has completed. To ascertain when a record operation has actually been performed, a program can specify completion routines or issue a wait request and regain control when the record operation is complete.

Record Transfer Modes — A program can use either of two record transfer modes to gain access to each record in memory:

- move mode
- locate mode

Move mode means that the individual records are copied between the I/O buffer and a program. For read operations, RMS reads a block into an I/O buffer, finds the desired record within the buffer, and moves the record to a program-specified location in its work space. For write operations, the program builds or modifies a record in its own work space and RMS moves the record to an I/O buffer. RMS supports move mode record operations for all file organizations.

Locate mode enables programs to read records directly in an I/O buffer. Locate mode reduces the amount of data movement, thereby saving processing time. RMS provides the program with the address and size of the record in the I/O buffer. RMS supports locate mode record transfers on all file organizations for read operations only.

RMS Record Locking

VAX-11 RMS provides a record locking capability for files that use the relative and indexed organization. In addition, RMS record locking is supported for sequential files with 512-byte fixed length records. This provides control over operation when the file is being accessed simultaneously by more than one program and/or more than one stream in a program. Record locking makes certain that when a program is adding, deleting, or modifying a record on a given stream, another program or stream is not allowed access to the same record or record cell. RMS-11 executing in compatibility mode does not support record locking and file sharing. There are two varieties of record locking and unlocking:

- Automatic Record Locking — The lock occurs on every execution of a $FIND or $GET macro instruction, and the lock is released when the next record is accessed, the current record is updated or deleted, the record stream is disconnected, or the file is closed. The $FREE macro instruction explicitly unlocks all records previously locked for a particular record stream. The $RELEASE macro instruction explicitly unlocks a specified record in a record stream.
- Manual Record Locking — In manual record locking, varying degrees of locking may be specified by setting bits in the record processing options field (ROP) of the RAB. The ULK bit specifies manual (as opposed to automatic) locking and unlocking. This bit specifies that locking will occur on the execution of a $GET, $FIND, or $PUT macro instruction and that unlocking may take place explicitly only via a $FREE or $RELEASE macro instruction. The NLK bit specifies that the record accessed with either a $GET or $FIND macro instruction is not to be locked, while the RLK bit specifies that a record may be accessible for read purposes but may not otherwise be accessed.

UTILITY LANGUAGES

VAX/VMS supports a number of data management facilities: DATATRIEVE, VAX-11 SORT, and the Forms Management System (FMS) utility package.

DATATRIEVE

DATATRIEVE is user application software that provides direct, easy, and fast access to data contained in VAX-11 RMS (Record Management System) files. The system is designed for relatively unsophisticated computer users; everyday use of DATATRIEVE requires no programming skills. While providing the user with an inquiry language and a report writing facility, DATATRIEVE also supports a user-specifiable Data Dictionary which describes VAX-11 RMS record formats. DATATRIEVE data management facilities include interactive retrieval, sort, update, and display of data records, in addition to maintenance commands for the Data Dictionary.

DATATRIEVE Inquiry Facility

DATATRIEVE accepts English-like commands from the user, and reacts by modifying, updating, or extracting data from the specified VAX-11 RMS file. In those cases where certain sequences of commands need to be issued on a recurring basis, DATATRIEVE provides a feature that permits the definition and use of procedures. A procedure is a group of DATATRIEVE statements and commands identifiable (callable) by a unique procedure name. At any time during the interactive session, this group of DATATRIEVE statements and commands can be invoked simply by calling the procedure name.

DATATRIEVE Report Writer Facility

In addition to query commands, DATATRIEVE provides a report facility to generate reports from VAX-11 RMS files. The report facility allows the user to specify the following parameters:

- Spacing
- Titles
- Column headings
- Page headings and footnotes
- Report headings

Commands to the report facility are simply an extension of query facility commands. Although the report facility provides extensive formatting capabilities, its default settings are suitable for many applications, further simplifying its use. Furthermore, errors in commands are discovered immediately (as in the query facility), so the user can correct the commands before printing wrong or incomplete reports.

Basic Commands

DATATRIEVE uses a simple English-like command language for data retrieval, modification, and display. Prompting is automatic for both command and data entry.

The major commands are:

- HELP — provides a summary of each DATATRIEVE command.
- READY — identifies a domain for processing and controls the access mode to the appropriate file.
- FIND — establishes a collection (subset) of records contained in either a domain or a previously established collection based on a Boolean expression.
- SORT — reorders a collection of records in either the ascending or descending sequence of the contents of one or more fields in the records.
- PRINT — prints one or more fields of one or more records. Output can optionally be directed to a line printer or disk file. Format control can be specified. A column header is generated automatically.
- SELECT — identifies a single record in a collection for subsequent individual processing.
- MODIFY — alters the values of one or more fields for either the selected record or all records in a collection. Replacement values are prompted for by name.
- STORE — creates a new record. The value for each field contained in the record is prompted for by name.
- ERASE — removes one or more records from the RMS file corresponding to the appropriate domain.
- FOR — executes a subsequent command once for each record in the record collection, providing a simple looping facility.
- EXTRACT — copies domains, records, procedures, and tables from the Data Dictionary to an external file.
- SHOW FIELDS — prints field names and data types for all fields in ready domains.
- DEFINE DICTIONARY — allows creation of private dictionaries.

In addition to the simple data manipulation commands, a number of more complex commands are available for the advanced user. These commands, such as REPEAT, BEGIN-END, and IF-THEN-ELSE, may be used to combine two or more DATATRIEVE commands into a single compound command. These, in turn, may be stored in the Data Dictionary as procedures for invocation by less experienced users.

DATATRIEVE provides a full set of arithmetic operators (addition, subtraction, multiplication, division, and negation), a set of statistical operators (total, average, maximum, minimum, and count), and provides automatic conversion between data types used in the FORTRAN, COBOL, DIBOL, and BASIC languages.

Terminology

Files, domains, collections, records, and fields are terms of fundamental importance to the file structure of DATATRIEVE.

Records are groups of related items of data that are treated as a unit. For example, all the pieces of data describing a model of a yacht in a marina could be grouped to constitute the record for that yacht.

Each of the individual pieces of data in a record is referred to as a field. The yacht's model number, length, and price are all potential fields in its record.

The term files refers to the logically related groups of data that are kept by RMS. For example, we might put all of the yacht records for a current inventory of yachts into one file.

Domains are named groups of data containing records of a single type. A DATATRIEVE domain consists of all the records in a particular RMS file, in addition to a record definition of this file contained in the Data Dictionary. In this case, we could say that all the yacht records for the current inventory are kept in the YACHTS domain. The number of records in any domain may change as new records are stored or old records are erased.

A record collection is a subset of a domain. It may consist of no records, one record, or up to all the records in the domain. Using our previous example, we could say that all the yachts manufactured by Grampian could be made to form the Grampian collection, while those yachts manufactured by Islander could be used to form the Islander collection. To carry this example one step further, if the inventory is currently out of stock of yachts manufactured by Seaworthy, the Seaworthy collection will be empty, or null.

The Data Dictionary is a location where the definitions for procedures, records, and domains are kept in a standard fashion by DATATRIEVE. The data administrator will be concerned with the creation and maintenance of Data Dictionary information. Certain users will be able to display certain information from this dictionary, but only management will be concerned with defining it.

Keywords

DATATRIEVE utilizes language elements called keywords which have a specific denotation and associated function. If they are used in any other context, they may serve to confuse the system about user intentions. Thus, it is good policy to avoid the use of these words as names of domains, procedures, records, fields, and collections.

Additional DATATRIEVE Features

Among the many DATATRIEVE features supported by VAX/VMS are:

- Application Design Tool (ADT)
  ADT allows less experienced users to set up simple DATATRIEVE applications. Through a simple dialogue, ADT generates a command file containing the record, domain, and file definitions.
- Nested Procedures
  Procedures may contain references to other procedures (nested procedures) provided that no procedure invokes itself either directly or indirectly. The maximum depth of nesting varies from about 10 to about 30 depending on the amount of memory available, number and size of established collection, etc.
- Data Hierarchies
  Use of hierarchies allows manipulation of complex data containing lists and sublists. A hierarchy may be defined as a single file with a repeating group or multiple domains automatically cross-linked. Extensions related to hierarchies include the inner print list (to override default formatting of a sublist) and the ANY Boolean expression, which allows DATATRIEVE to search a sublist for the existence of a particular record.

- Views
  Views can be used to restrict the set of fields accessible, to apply an automatic selection criterion to a file, or to cross-link a number of elementary domains to/from an apparent hierarchy. Once defined, a view is indistinguishable from an RMS domain to the user.
- DATE Data Type
  This allows easy inclusion of dates in DATATRIEVE records, direct comparison of dates, computation of elapsed dates. Dates may be formatted for printing in virtually any form. Similarly, DATATRIEVE accepts the entry of dates in virtually any form. The DATATRIEVE date format is compatible with the VAX/VMS date standard.
- Tables
  Tables are generally used to translate encoded values into something that can be edited by the DATATRIEVE editor. Table lookups are performed by the VIA value expression; table searches (for table membership) are specified with the Boolean IN expression.
- TOTAL Statement
  The TOTAL statement allows very simple computation of totals and subtotals.
- CONTAINING Relational Operator
  CONTAINING is used in a record selection expression to retrieve records with a field containing a particular substring. The substring may be anywhere in the field, and need not match the case (uppercase/lowercase) of the search string. For example, the command:

  FIND BOOKS WITH TITLE CONTAINING "LASER"

  will find all records in BOOKS with the word "LASER" somewhere in the field TITLE.
- OCCURS Clause
  Use of the OCCURS clause permits definition of records containing a repeating group (sublist). The sublist may be of fixed or variable length.
- Value Validation
  A Boolean validation expression may be included as part of a field description in a record definition. If specified on a field, the validation expression is automatically executed every time the field is modified, to insure that only legal values are stored in a data base. If a validation error is detected, the user is re-prompted for a new value if possible, or the DATATRIEVE statement is aborted.
- COMPUTED BY Fields
  A field in a record definition may be defined as a COMPUTED BY field by specifying a value expression to be computed for its value. A COMPUTED BY field takes no space in the actual RMS record, and is computed on reference. A COMPUTED BY field may be used in conjunction with a table to provide completely automatic table lookup.
- Tutorial Software (Guide Mode)
  A CRT-based tutorial is included in DATATRIEVE. The tutorial feature can be used only by VT52, VT52-compatible, and VT100 terminals. A tutorial session is entered by the DATATRIEVE command:

  SET GUIDE

  The software is self-documenting.
- Procedure Editor
  A DATATRIEVE procedure editor has been added. The editor is invoked by the command:

  EDIT procedure-name

  where "procedure-name" is the name of an existing procedure.

  The command EDIT procedure-name invokes an editor which can insert, replace, or delete text from procedures defined in the data dictionary.

VAX-11 SORT/MERGE

VAX-11 SORT/MERGE is a native mode utility that may be run interactively, as a batch job, or it can be callable from a user-written VAX-11 native mode program.

The SORT utility allows the user to reorder data from any input file into a new file in a sequence based upon key fields within the input data records. A user can specify up to ten input files and SORT will produce one sorted output file. The sorting sequence is determined by user-specified control fields, also known as key fields, within the data themselves. If the user does not wish to reorder the data base, SORT can still be used to extract key information, sort that information, and store the sorted information as a permanent file. Later that file can be used to access the data base in the order of the key information in the sorted file. The contents of the sorted file may be entire records, key fields, or record file addresses which point to the position of each record within the file.

SORT provides four sorting techniques:

- **Record Sort** produces a reordered data file sorted by specified keys, moving the entire contents of each record during the sort. A record sort can be used on any acceptable VMS input device and can process any valid VAX-11 RMS format.
- **Tag Sort** produces a reordered data file by sorting specified keys, but moving only the record keys during the sort. SORT then randomly reaccesses the input file to create a resequenced output file according to those record keys. The tag sort method conserves temporary storage, but can accept only input files residing on disk.
- **Address Sort** produces an address file without reordering the input file. The address file contains RFAs, a pointer to each record's location in the file which can later be used as an index to read the data base in the desired sequence. Any number of address files may be created for the same data base. A customer master file, for instance, may be referenced by customer-number index or sales territory index for different reports. Address sort is the fastest of the four sorting methods.
- **Index Sort** Index sort produces an address file containing the key field of each data record and a pointer to its location in the input file. The index file can be used to randomly access data from the original file in the desired sequence.

The MERGE utility permits the user to merge data from as few as two, to as many as ten similarly sorted input files. The MERGE utility merges the data according to key field(s) defined by the user and generates a single output file. The input files to be merged must be in sorted order, i.e., the SORT and MERGE key fields must be the same.

The following example illustrates the sorting of a sales record file by customer last name. The name of the initial file is SALES.DAT. Each record contains six fields: date of sale, department code, salesperson, account number, customer name, and amount of sale. The numerical ranges listed below the set of records indicate the position and size of each information field within the record.

DATE	DPT	SALESP	ACCT	CUST-NAME	AMT
091580	25	Fielding	980342	Coolidge Carol	24999
091580	25	Sanchez	643881	McKee Michael	2499
091580	25	Bradley	753735	Rice Anne	10875
091580	19	Arndt	166392	Wilson Brent	1298
091580	28	Meredith	272731	Karsten Jane	4000
091580	25	Bradley	829582	Olsen Allen	3350
091580	19	Erkkila	980342	Coolidge Carol	7200
1-7	8-10	11-21	22-28	29-58	59-65

The user may now rearrange the sales records in file SALES.DAT according to any of the file's information fields. For instance, to sort the file in alphabetical order of customer's last name, the user would type the following command sequence:

$ SORT/KEY=(POSITION=29,SIZE=30) SALES.DAT BILLING.LIS <cr>

In this command sequence, the user is defining the SORT key to be the customer's last name and the output file to be BILLING.LIS

The user may now obtain a listing of the sorted data file by using either the TYPE or PRINT commands.

$ TYPE BILLING.LIS

DATE	DPT	SALESP	ACCT	CUST-NAME	AMT
091580	19	Erkkila	980342	Coolidge Carol	7200
091580	25	Fielding	980342	Coolidge Carol	24999
091580	28	Meredith	272731	Karsten Jane	4000
091580	25	Sanchez	643881	McKee Michael	2499
091580	25	Bradley	829582	Olsen Allen	3350
091580	25	Bradley	753735	Rice Anne	10875
091580	19	Arndt	166392	Wilson Brent	1298

To perform the MERGE function, the MERGE utility expects presorted data files upon which to operate. In the following example, MERGE is operating upon two presorted (by alphabetical order) sales data files, STORE1.FIL and STORE2.FIL.

STORE1.FIL

DATE	DPT	SALESP	ACCT	CUST-NAME	AMT
091580	19	Erkkila	980342	Coolidge Carol	7200
091580	25	Fielding	980342	Coolidge Carol	24999
091580	28	Meredith	272731	Karsten Jane	4000
091580	25	Sanchez	643881	McKee Michael	2499
091580	25	Bradley	829582	Olsen Allen	3350
091580	25	Bradley	753735	Rice Anne	10875
091580	19	Arndt	166392	Wilson Brent	1298

STORE2.FIL

DATE	DPT	SALESP	ACCT	CUST-NAME	AMT
091580	20	OConnor	358419	Beaulieu Ronald	1598
091580	04	Docus	980342	Coolidge Carol	575500
091580	25	Fielding	669011	Fernandez Felicia	12000
091580	35	Leith	848105	Kingsfield Stanley	5550
091580	04	Kramer	561903	Landsman Melissa	230000
091580	20	OConnor	643881	McKee Michael	995
091580	19	Erkkila	454389	VanDerling Julie	5480

To merge the two data files, the user must type the following command sequence:

$ MERGE/KEY=(POSITION=29,SIZE=30)
STORE1.FIL,STORE2.FIL CENTR.FIL <cr>

The user has indicated in the above command sequence that the files are to merged via the alphabetical order of the customer's last name. The user can examine the output file via the PRINT or TYPE commands.

$ TYPE CENTR.FIL <cr>

DATE	DPT	SALESP	ACCT	CUST-NAME	AMT
091580	20	OConnor	358419	Beaulieu Ronald	1598
091580	19	Erkkila	980342	Coolidge Carol	7200
091580	25	Fielding	980342	Coolidge Carol	24999
091580	04	Docus	980342	Coolidge Carol	575500
091580	25	Fielding	669011	Fernandez Felicia	12000
091580	28	Meredith	272731	Karsten Jane	4000
091580	35	Leith	848105	Kingsfield Stanley	5550
091580	04	Kramer	561903	Landsman Melissa	230000
091580	25	Sanchez	643881	McKee Michael	2499
091580	20	OConnor	643881	McKee Michael	995
091580	25	Bradley	829582	Olsen Allen	3350
091580	25	Bradley	753735	Rice Anne	10875
091580	19	Erkkila	454389	VanDerling Julie	5480
091580	19	Arndt	166392	Wilson Brent	1298

VAX-11 SORT/MERGE FEATURES

VAX-11 SORT can perform the following functions:

- reorder data files (records are sorted in ascending or descending order by up to ten keys which can be in any order)
- merge up to ten sorted input files into one sorted output file
- create reordered address files of RFAs and keys for software use
- SORT/MERGE fixed, variable, and VFC records
- SORT/MERGE ASCII character keys in ASCII or EBCDIC sequence
- SORT/MERGE sequential, relative, indexed-sequential files
- SORT/MERGE character, decimal, binary, unsigned binary, F_, D_, G_, and H_floating data types
- SORT can determine its own work file requirements based on input file RMS information received

- SORT can be controlled by a command string or specification file
- SORT can be tuned for maximum efficiency
- SORT provides four processing techinques: record, tag, address, index
- SORT/MERGE input files from any VAX/VMS input device
- Output sorted data files to any VAX/VMS output device
- SORT automatically prints out statistics upon completion
- Be invoked by a single command string, or can prompt the operator for input and then output file specification
- Respond with unique SORT/MERGE error messages in VAX/VMS format
- Optional sequence checking of input files on merge

VAX-11 SORT/MERGE supports the following key formats:

- character data are ASCII
- ASCII and EBCDIC collating sequence
- binary data are VAX representation
- packed decimal data are VAX representation
- zoned decimal data are VAX representation
- unsigned binary and F_, D_, G_, and H_floating
- string decimal data format can be:
  - leading separate sign
  - leading overpunched sign
  - trailing separate sign
  - trailing overpunched sign

SORT/MERGE as a Set of Callable Subroutines

SORT/MERGE can be used as a set of callable subroutines from any native VAX language. This subroutine package provides two functional interfaces to choose from: a file I/O interface and a record I/O interface. Both interfaces share the same set of routines, and the same calls are used for all languages.

SORT and MERGE subroutines are callable from VAX-11 COBOL using the standard COBOL SORT and MERGE verbs.

For either interface, the user can supply a key comparison routine. This feature allows the user the flexibility of departing from the key types supported by SORT/MERGE.

With this release of VAX/VMS, full MERGE capability has entered the list of SORT features callable as a subroutine. This allows a much increased file flexibility.

File I/O Interface

The file I/O interface allows the user to specify the input files and an output file to SORT or MERGE. SORT then reads the data from the input file(s) and sorts the data into the output file. MERGE also reads the data form the input file(s) and merges it into one output file.

Record I/O Interface

The record I/O interface allows the user to pass each individual data record to SORT/MERGE, let SORT/MERGE order them and then receive each record back in the correct order, individually, from SORT/MERGE.

Programming Considerations

Any program can use either SORT/MERGE subroutine interface with any of the VAX-11 native mode languages.

SORT/MERGE PERFORMANCE FEATURES

- SORT/MERGE compiles a key comparison routine specific to each sort or merge. This results in a substantial reduction of CPU usage.
- Work files are not created until they are needed. This reduces overhead when sorting small files.
- The internal record size has been reduced, therefore, less I/O is required to do intermediate merge passes for SORT.
- For some sorts, the number of intermediate merge passes has been reduced, thereby providing a substantial increase in speed of the SORT.

VAX-11 FORMS MANAGEMENT SYSTEM (FMS)

VAX-11 Forms Management System (FMS) is a utility package used to provide video form support for applications on the VT100 video terminal. VAX-11 FMS provides a flexible, easy-to-use interface between the form application program and the terminal user. FMS allows a terminal user to develop form applications using any native mode language processor.

Using Forms in an Application

VAX-11 FMS forms include a video screen image comprising data fields and constant background text, along with protection and validation attributes for individual data fields. The data fields and background text can be highlighted using any combination of the VT100 video attributes: reverse video, underline, blink, and bold characters. Split screen and scrolling capabilities allow the user to view more data than can be displayed on the screen at one time.

Individual data fields can be display-only, enter-only (no echo), or can be restricted to modification by privileged users. Data fields can be formatted with fill characters, default values, and other formatting characters—such as the dash in a phone number—which assist the terminal operator, but which are not visible to the application program. Fields may be right- or left-justified or may use a special fixed decimal data field type to normalize floating point decimal numbers into fixed point for easier use in computation.

Field validation includes checking each keystroke in a field for the proper data type (e.g., alphabetic, numeric, etc.). Fields may also be defined as "must enter" or "must complete."

A line of HELP information may be associated with each field, and a chain of one or more HELP forms may be associated with each form. If people need additional instructions while using a form, they press the HELP key to display the HELP line for the current field. A second HELP

keystroke displays the first HELP screen for the current form, so that from any point in the application form the user can get to an entire series of HELP forms. In this way the entire user manual for an application can be put on-line, automatically keyed to the appropriate user form.

Almost any class of application can benefit from using VAX-11 FMS. Source data entry and inquiry/response/update are the most obvious types of forms-oriented uses, but other types of programs can benefit equally well. For example, a simulation or numerical analysis program could use FMS forms to explain and accept input parameters, and then to format and scroll through the output of the run. Forms might constitute the front end of a student registration or a computer-aided instruction system. Alternatively, they could be used to review data acquired from laboratory or factory instrumentation, or to format the operator input of control parameters to such processes. Almost any application which uses alphanumeric video terminals can be enhanced by using FMS forms to talk to the terminal user.

Developing Applications with VAX-11 FMS

VAX-11 FMS forms are created and modified interactively on the screen using a special FMS utility called the Form Editor (FED). Because the video image is typed and manipulated directly on the screen, there is no need to lay out a form on a paper chart or to code complex specifications into a form definition program written in a difficult language. Rather, the form creator always sees the form on the screen exactly as it will appear to the application user. A set of 24 editing and data manipulation functions invoked through the function keypad of the VT100 terminal allow easy alteration of the form description.

Fields are defined interactively via the function keypad and by typing the COBOL-like picture character for each position of the field directly on the screen. The remaining field attributes, such as field name, default value, and the contents of the HELP line, are described by interactively filling in a questionnaire form on the screen. Another form is used to define certain characteristics which apply to the form as a whole, such as the name of the form and of the first HELP form associated with it, and whether the screen is to be placed into reverse video or 132-column mode. A third form is used to specify application constants, called "named data," which can be stored with the form instead of hard-coded into the application program. This last feature allows application parameters such as small data tables, file names, names of subsequent forms, etc., to be stored with the form and edited almost interactively.

When the application developer is satisfied with the appearance and content of the form, the Form Editor writes the form out into a work file. The Form Utility (FUT) is then used to insert the form into a new or existing form library, from which it will be retrieved when the application program is executed. The Form Utility may be used to perform other maintenance functions on form libraries, as well as to generate hard-copy descriptions of forms suitable for inclusion in application documentation. FUT can also generate COBOL Data Division code to correspond to the form description.

Once the form has been stored in a form library, one or more application programs to use it must be coded. These application programs control the interaction between themselves, the form, and the operator by making calls to a library of VAX-11 FMS subroutines called the Form Driver (FDV). Under the direction of the calling application program, the Form Driver displays forms, performs all screen management an application requires, handles all terminal input and output, and validates each operator entry by checking it against the field description for the field. A broad selection of subroutine calls allows the program to communicate with the screen on either a full-screen or field-by-field basis. While the terminal operator is typing data, all data validation and formatting, error messages, and HELP requests occur completely transparently to the application program.

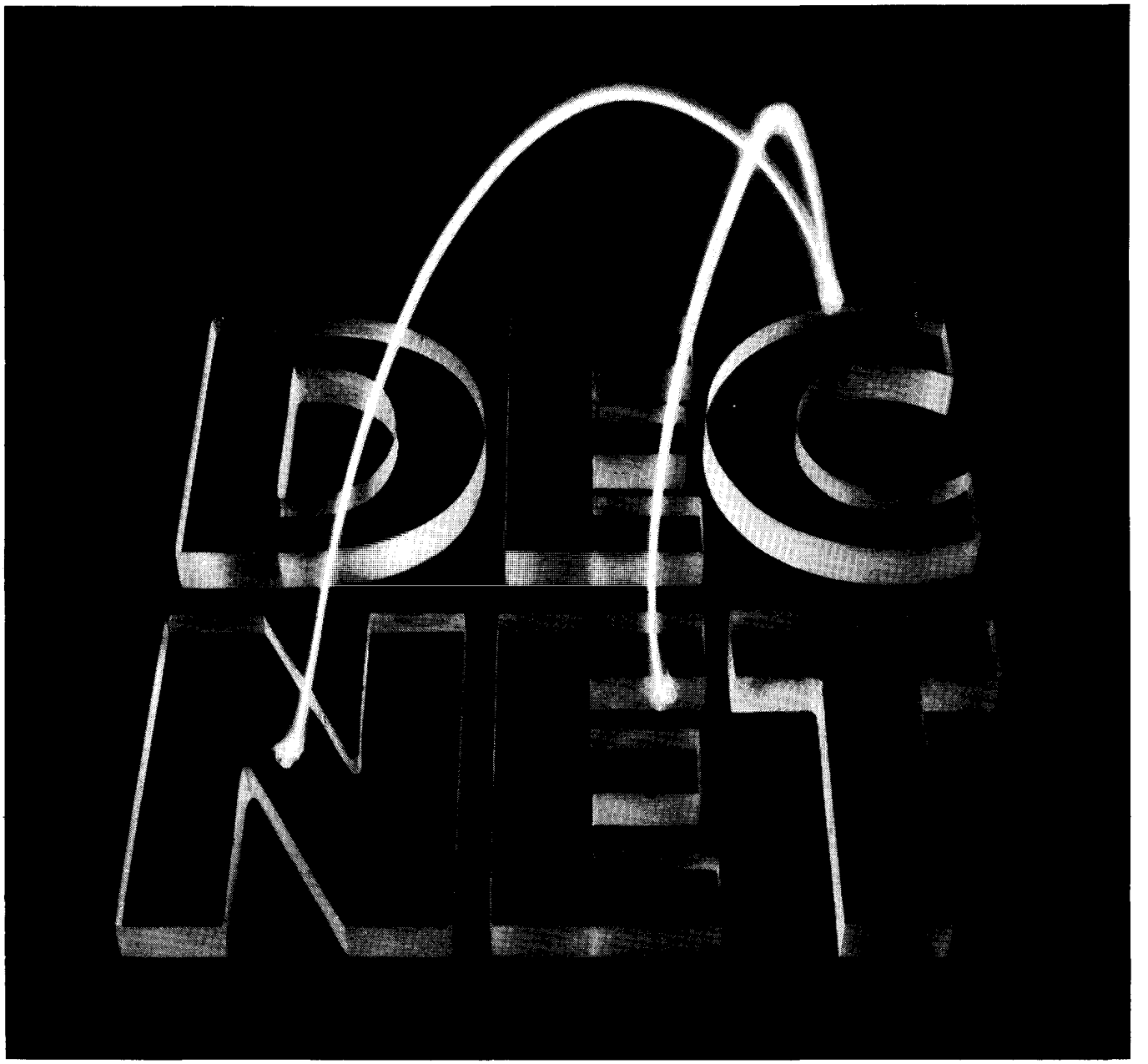
Maintaining VAX-11 FMS Applications

Several features of VAX-11 FMS make maintenance of applications using forms both rapid and reliable. The most obvious such feature is the interactive editing capability of the Form Editor. Capabilities such as Open Line for insert, Cut and Paste, etc., make modifications to existing forms quick and easy. Furthermore, when fields are moved around on the screen, their attributes are preserved, so that re-entry of the form is not required.

Perhaps the most important contribution to application maintainability comes from the fact that the application makes all references to screen data by field name. These field name references are not resolved until the program executes, so that the form description is actually independent of the application program. This means that it is easy to write programs that do not know or depend upon the specific order of the fields, or even know the names or each field. This capability, combined with the storage of forms in libraries, means that in many cases the form can be rearranged, or new fields added, without requiring that the application program be recompiled, or even relinked!

The last feature of VAX-11 FMS that promotes application maintainability is the ability to store application parameters with forms as named data. Parameters such as the names of related forms or programs, file specifications, small look-up tables, range check boundaries, and error messages specific to the particular form may be stored with the form and edited with the same rapidity and ease. For example, imagine an application that will be used by operators who speak a variety of different languages. The first thing the operator would do when logging into the application would be to select a language. The program would simply open the form library with all the forms translated into that language. Named data would be used to store all other language-dependent application parameters, such as program-generated error messages, abbreviations (Y=YES or O=OUI or S=SI), etc. The same program code would then be executed regardless of the language the application would "speak." A new language could be added by simply modifying the initial language selection form and creating a form library with the forms and named data translated into the new language.

10 Data Communications Facilities



DECnet is a family of network products developed by Digital Equipment Corporation that adds networking capability to DIGITAL's computer families and operating systems. Using DECnet, various DIGITAL computer systems can be linked together to facilitate remote communications, resource sharing, and distributed computation. DECnet is highly modular and flexible. It can be viewed as a set of tools or services from which a user selects those appropriate to build a network to satisfy the requirements of a particular application.

DIGITAL Network Architecture (DNA) provides the common network architecture upon which all DECnet products are built. The architecture is designed to handle a broad range of application requirements because all the functions of the network from the user interface to physical link control are completely modular. DNA allows nodes to operate as switches, front-ends, terminal concentrators, or hosts.

DECnet-VAX:

- provides an interprocess communication facility that is highly transparent and easy to use
- provides a higher-level language programming interface
- allows programs to access files at other systems
- allows users and programs to transfer files between systems
- allows users to transmit command files to be executed on other systems
- allows an operator to down-line load RSX-11S system images into other systems

VAX/VMS also supports protocol emulators (Internets), which enable DIGITAL systems to communicate with other vendors' systems.

INTRODUCTION

DIGITAL computers can communicate with other DIGITAL computers either remotely or locally via a network. By utilizing protocol emulators (Internets), they can communicate with computers from other suppliers.

DECnet is the family of products that allow DIGITAL systems to participate in a cooperative multiprocessing environment known as a network. A network is a configuration of two or more independent computer systems, called nodes, linked together to facilitate remote communications, share resources, and perform distributed processing. Network nodes are not all required to run on the same type of operating system. Within the scope of a single network, several nodes with different operating systems and different features can interact to provide increased processing flexibility.

Adjacent network nodes are linked together via carriers known as physical links. Physical links can be relatively permanent bonds, such as telephone lines or cable wires laid from one node to another, or they can be temporary connections that change with each use, such as dialed-up telephone calls.

In a network of DECnet nodes, several tasks can use the same physical link to exchange data. That is, more than one data path can be handled simultaneously by a physical link. This data path is known as a logical link. A task is an image running in the context of a process.

With DECnet, a variety of computer networks can be implemented. They typically fall into one of three classes:

- Communications Networks. These networks exist to move data from one, often distant, physical location to another. The data may be file-oriented (as is often the case for remote job entry systems) or record-oriented (as occurs with the concentration of interactive terminal data). Interfaces to common carriers, using both switched and leased-line facilities, are normally a part of such networks. Such networks are often characterized by the concentration of all user applications programs and data bases on one or two large host systems in the network. Figure 10-1 illustrates such a network.

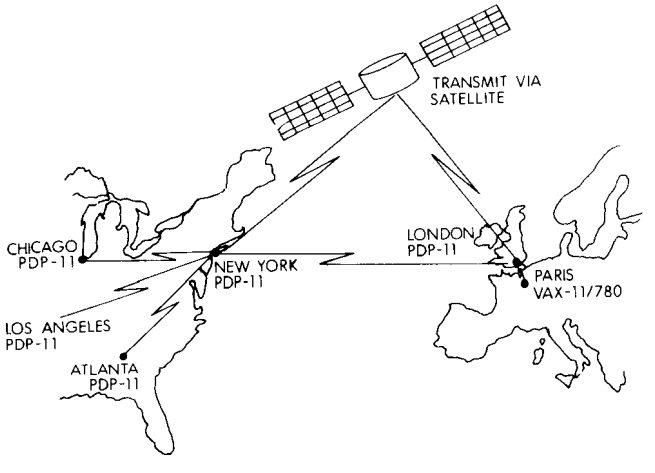


Figure 10-1

Communications Network

- Resource-Sharing Networks. These networks exist to permit sharing expensive computer resources among several computer systems. Shared resources not only include peripherals such as mass storage devices, but they can also include logical entities, such as a centralized data base which is made available to other systems in the network. Such networks are often characterized by the concentration of high-performance peripherals, extensive data bases, and large programs on one or two host systems in the network, while the satellite systems have less expensive peripherals and smaller programs. Figure 10-2 illustrates a resource-sharing network.

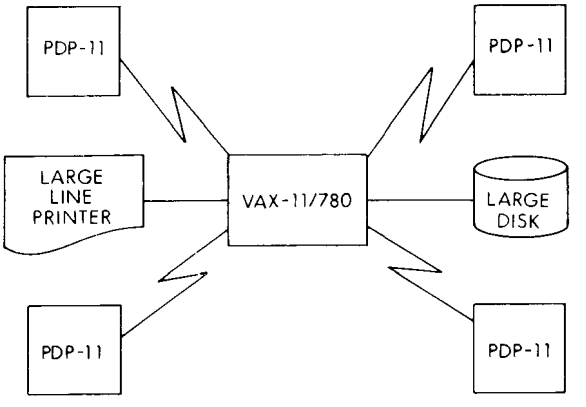


Figure 10-2

Resource-Sharing Network

- Distributed Computing Networks. These networks coordinate the activities of several independent computing systems and exchange data among them. Networks of this nature may have specific geometries (star, ring, hierarchy), but often have no regular arrangement of links and nodes. Such networks are usually configured so that the resources of a system are close to the users of those resources. Distributed computing networks are usually characterized by multiple computers with applications programs and data bases distributed throughout the network. Figures 10-3 and 10-4 illustrate two such networks.

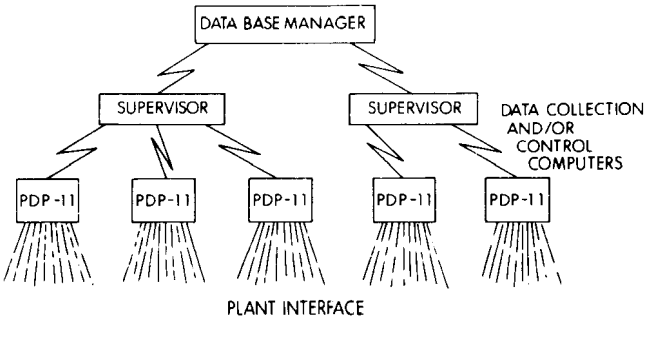


Figure 10-3

Typical Manufacturing Network

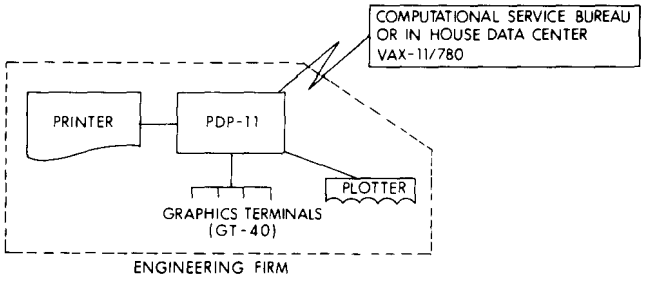


Figure 10-4

Computational Network

When DECnet is used to connect heterogeneous systems, each node of the network has both common DECnet attributes and system-specific attributes. Programs executing in native mode can access the following network facilities:

- Interprocess (Task-to-Task) Communication: Programs executing on one system can exchange data with programs executing on other systems.
- Intersystem File Transfer: A program or user can transfer an entire data file from one system to another.
- Intersystem Resource Sharing: Programs executing on one system can access files and devices physically located at other systems in the network. Access to devices in other systems is provided through the file system of the target node and is subject to that node's file system restrictions.
- Network Command Terminal: A terminal on one VAX system can appear to be connected to another VAX system in the network.
- Down-Line System Loading: Initial load images for RSX-11S systems in the network can be stored on the host VAX system, and be loaded into adjacent PDP-11 systems configured for the RSX-11S operating system.
- Down-Line Command File Loading: Command language users can send command files to a remote node to be executed there. However, no status information or error messages are returned.

DIGITAL NETWORK ARCHITECTURE

The DIGITAL Network Architecture (DNA) is a set of protocols (rules) governing the format, control, and sequencing of message exchange for all DECnet implementations. DNA controls all data that travel throughout a DECnet network and provides a modular design for DECnet.

Its functional components are defined within four distinct layers: User Layer, Network Service Layer, Data Link Layer, and Physical Link Layer. Each layer performs a well-defined set of network functions (via network protocols) and presents a level of abstraction and capability to the layer above it.

User Layer

This layer contains all user and DIGITAL supplied functions. Modules in this layer include network remote file access modules, a remote file transfer utility, and a remote system loader module. The protocol used for remote file access and file transfer is the Data Access Protocol (DAP).

Network Service Layer

This layer provides a location-independent communication mechanism for the user layer. The means by which they communicate is called a **logical** link. Two network application modules may communicate with each other by means of the network service layer regardless of their network locations. The protocol used between network service modules is the Network Services Protocol (NSP).

Data Link Layer

This layer provides the transport layer with an error-free communication mechanism between adjacent nodes. The data link module specified for this layer implements the DIGITAL Data Communications Message Protocol (DDCMP). The functions provided by this layer are independent of communication facility characteristics. For DECnet-VAX, DDCMP is incorporated into the microprocessor of the communications interface.

Physical Link Layer

This layer, the lowest layer in the DNA structure, provides the data link layer with a communication mechanism between adjacent nodes. Several modules are specified for this layer, one for each type of communication device that can be used in a DECnet network.

DNA is system-independent. It enables a variety of DIGITAL computers running a variety of DIGITAL operating systems to be tied together in a DECnet network.

A DECnet network can grow both in size and in the number of functions it provides. It can, therefore, be adapted to new technological developments in both hardware and software. Existing DECnet implementations can take advantage of these new technologies. DECnet components can be replaced if better communications hardware becomes available or if technical innovations in networking occur. A DECnet network can accommodate the change of a function from software into hardware.

DECNET-VAX FEATURES

The capabilities offered the DECnet-VAX programmer and terminal user extend through a wide range of network functions.

File Handling Using a Terminal

By using DECnet-VAX DIGITAL Command Language (DCL) commands, the user can copy files from one node to another, delete files stored on a remote node, take directories of files on remote nodes, and transfer a command file to another node and then execute the command file on the remote node.

File Handling Using Record Management Services

A wide range of VAX/VMS Record Management Services (RMS) can be used to handle files and records stored on remote nodes. At the file level, these operations include opening, closing, creating, deleting, and updating files stored on a remote node. Also, at the record level, RMS can be used to read, write, update, and delete records stored on a remote node.

Network Command Terminal

The network command terminal facility allows a local terminal to operate as if it were physically connected to a remote computer.

Intertask Communications

Any native-mode language programmer can write programs that perform intertask (interprocess) communication. Intertask communication is a method of creating a logical link between two tasks, exchanging data between the tasks, and disconnecting the link when the communication is complete.

Intertask communication routines can be coded using one of two methods, transparent or nontransparent.

Transparent Intertask Communication — The program opens the network interchange as if it were preparing for device access, and then performs a series of reads and writes just as it would to a pair of serial devices, one for input and the other for output.

By its very nature, transparent access has no calls specifically associated with DECnet. The calls used for interprocess communication are the same as the calls used for accessing a sequential file in a higher-level language: OPEN, CLOSE, READ, WRITE, etc. The programmer can choose to include the target node name in the OPEN statement, or can defer assignment using logical names.

Nontransparent Intertask Communication — In nontransparent access, a program can obtain information about the network status to control the nature of its communication with other processes or tasks. This method of coding intertask communications is available to the MACRO programmer. And if you don't do AST processing or attempt to accept multiple connects, you may program in any language. Nontransparent access is available only through calls to operating system service procedures. A program can issue the following requests:

- CONNECT—establish a logical link (the analog of OPEN)
- CONNECT REJECT—reject a connect initiation
- RECEIVE—receive a message (the analog of GET or READ)
- SEND—transmit a message (the analog of PUT or WRITE)
- SEND INTERRUPT MESSAGE—transmit a high-priority message
- DISCONNECT—terminate a conversation (the analog of CLOSE)

The process can send optional data along with the connect request; for example, the size or number of messages that it wants to send. The receiving process or task can accept or reject the connect initiation. A process can accept multiple connect requests.

A process can send or receive mailbox messages to or from another process or task. Mailbox message traffic is essentially no different from data message traffic except that it uses a mailbox associated with the I/O channel over which the logical link was created. (This is the same mechanism used, for example, for telling programs that unsolicited terminal data are available.) A logical link, therefore, has two subchannels over which messages can be transmitted: one for normal messages and another for high-priority messages. In DECnet-VAX, an interrupt message is written to a mailbox that a process supplies for that purpose.

In DECnet-VAX, a program using nontransparent access normally opens a control path directly to a Network Ancillary Control Process (NETACP), and designates one or more mailboxes for receiving information from the NETACP about the logical or physical links over which the process is communicating. The NETACP can notify a process when:

- a partner requests a synchronous disconnect
- a partner requests a disconnect abort
- a partner exits
- a physical link goes down
- an NSP protocol error is detected

DIGITAL COMMAND LANGUAGE (DCL) FILE HANDLING

A VAX/VMS DCL user can transfer files from one node to another and delete files at other nodes. However, to perform operations on files stored on a remote node, the user must prefix the file specification with the remote node's name, and an optional access control string as follows:

nodename"access control string"::filename.filetype;version

where:

nodename =	A 1- to 6-character name (numerics or uppercase alphabetics) identifying the remote network node.
access control string =	Typically, a "username password." If omitted, default login information comes from an entry located in the local configuration data base. The double colon (::) following the nodename separates the nodename from the file specifier.
filename = filetype version	Use the following format for a DECnet-VAX node: device:[directory]filename.filetype;version

DECnet-VAX supports a subset of VAX/VMS (DCL) commands. They are:

APPEND
ASSIGN
COPY

DEASSIGN
DEFINE
DELETE
DIRECTORY
SUBMIT
TYPE

The following examples illustrate the COPY and SUBMIT commands:

$ COPY BOSTON::DBA1:TEST.DAT DENVER::DMA2:

transfers a file named TEST.DAT from the disk (DBA1:) at the node named BOSTON to the disk (DMA2:) at the node named DENVER.

Using the VAX/VMS command SUBMIT, a terminal user can have a command file executed at another node in the network. For example, the command:

$ SUBMIT/REMOTE WASHDC::INITIAL.COM

preceded by a DCL COPY command will transfer the command file named INITIAL.COM from the host system to the node named WASHDC, where the command file is executed. The SUBMIT command assumes that the file already exists at the remote node. Command files must be written in the command language of the system. No status information or messages are returned to the sender.

RECORD MANAGEMENT SERVICES FILE HANDLING

By using a subset of the VAX-11 Record Management Services (RMS), the user can manipulate records and files stored on remote DECnet nodes. However, before using VAX-11 RMS to perform operations on files and records stored on a remote node, the user must prefix the file specification with the node name of the remote node and an optional access control string, just as with any other remote file application.

Much of the VAX-11 RMS functionality is supported by DECnet-VAX, including managing sequential, relative, and indexed file organizations. A large number of the VAX-11 RMS macros are available to network users.

SAMPLE VAX-11 FORTRAN INTERTASK COMMUNICATION

This section describes the basic communication protocol involving VAX-11 FORTRAN intertask communications. The user communicates with another task in much the same way as an access to a sequential file, i.e., via OPEN, READ, WRITE and CLOSE statements. Similar capabilities exist in any of the native mode languages.

Three major steps in VAX-11 FORTRAN intertask communication are:

1. Creating a logical link between tasks.
2. Sending and receiving messages (each message can be 1 to 512 bytes in length).
3. Destroying the link at the end of the message dialogue.

Creating a Logical Link Between Tasks

A logical link between tasks can be created only if they agree to cooperate with each other. That is, one task must request that a logical link be created, and the other must agree to accept the request. The task requesting the logical link is called the source task; the one agreeing to accept the logical link request is called the target task.

Sending and Receiving Messages

After the logical link has been created, the tasks must "cooperate" with each other. That is, for each message sent by a task (WRITE statement), the receiving task must issue a corresponding receive (READ statement).

In addition, the tasks must ensure that enough buffer space is allocated for messages, must ensure that the end of dialogue can be determined, and must determine which of the tasks will disconnect the logical link (CLOSE statement).

Disconnecting the Logical Link

Either task can disconnect the logical link by calling CLOSE. CLOSE aborts all pending sends and receives, disconnects the link immediately, and frees the channel number associated with the logical link.

VAX-11 FORTRAN Intertask Communication Example

Figure 10-5 illustrates intertask communications using normal VAX-11 FORTRAN I/O instructions.

MACRO TRANSPARENT INTERTASK COMMUNICATION

This section describes the fundamentals of coding a MACRO program for transparent intertask communications utilizing a subset of the existing macro calls available under VAX/VMS system services. These macro calls allow the user to perform intertask communications in much the same way as normal I/O operations are performed.

The term "transparent" simply implies that the calls are identical in format to all other I/O calls.

Thus, communication with another task is performed in much the same way as an I/O channel is assigned to a device ($ASSIGN). Reads and writes are then performed as if to a pair of sequential devices (that is, $QIO with the IO$_WRITEVBLK function or $OUTPUT, and $QIO with the IO$_READVBLK function or $INPUT). Finally, $DASSGN the device when communication is complete.

Three major functions in transparent intertask communication are:

1. Create a logical link between tasks.
2. Send and receive messages (each message can be 0 to 65,535 bytes long).
3. Delete the link at the end of the message dialogue.

Creating a Logical Link Between Tasks

A logical link between tasks can be created only if the tasks agree to cooperate with each other. That is, one task must request that a logical link be created, and the other task must agree to accept the request.

A logical link is requested by including a task specifier in the source task's $ASSIGN call. A task specifier identifies the remote node and the target task to which it will be connected.

The target task identifies the source task requesting the logical link connect request by specifying SYS$NET as the

Source Task Code

```
        PROGRAM DEMO2.FOR
C
C       This program prompts the user for a request, communicates
C       with a remote task to obtain the requested data, and displays
C       the answer for the user. The remote task is referenced by
C       the logical name TASK. If the remote task is named DEMO3.EXE
C       at node TULSA, the following procedure is used to run the
C       two programs:
C
C       $ DEFINE TASK TULSA::""TASK=DEMO3""
C       $ RUN DEMO2
C
        LOGICAL*1 CODE(4),BUFFER(20)
C
100     FORMAT          ('$Enter request code: ',4A1)
200     FORMAT          (4A1)
300     FORMAT          (Q,20A1)
400     FORMAT          ('0Stock number for code ',4A1,'is: ',20A1)
C
C       Request the remote task to be run and establish a logical
C       link into it.
C
        OPEN            (UNIT=1,NAME='TASK,'ACCESS='SEQUENTIAL',FORM='FORMATTED')
C
C       Prompt the user for a request code, send the code to the
C       remote task, read the reply from the remote task, and display it to
C       the user.
C       Repeat the cycle until the user enters 'Exit' as his request code.
C
10      ACCEPT          100,CODE
        IF(CODE(1).EQ.'E') GOTO 20
        WRITE           (1,200END=20)CODE
        READ            (1,300) NCHAR,(BUFFER(K),K=1,NCHAR)
        TYPE            400,CODE,(BUFFER(K),K=1,NCHAR
        GOTO            10
C
C       Finished.
C
20      CLOSE           (UNIT=1)
        END
```

Target Task Code

```
        PROGRAM DEMO3.FOR
C
C       This is the companion task for DEMO2. For each request it
C       receives from the remote task it replies with a 1- to 20-
C       character response. This program does not know the name of the
C       requesting task. To complete the logical link with its initiator, it
C       opens the 'file' specified by the logical name SYS$NET.
C
        LOGICAL*1 CODE(4),BUFFER(20)
C
100     FORMAT          (4A1)
200     FORMAT          (20A1)
C
```

Figure 10-5

VAX-11 FORTRAN Intertask Communications

```
C       Establish a communication path with the remote task.
C
        OPEN            (UNIT=1,NAME='SYS$NET:','ACCESS='SEQUENTIAL',FORM='FORMATTED')
C
C       Process requests until end-of-file encountered.
C
10      READ            (100,END=20)CODE
C
C       Perform appropriate processing to obtain result to
C       transmit back to the requesting task.
C
        WRITE           (1,200) (BUFFER(K),K=1,NCHAR)
        GOTO            10
C
C       Finished.
C
20      CLOSE           (UNIT=1)
        END
```

Figure 10-5 (Con't)
VAX-11 FORTRAN Intertask Communications

devnam argument in the $ASSIGN statement. This action completes the creation of the logical link.

Sending and Receiving Messages

After the logical link is created, the tasks must "cooperate" with each other. That is, for each message sent by a task ($QIO with the IO$_WRITEVBLK function or $OUTPUT), the receiving task must issue a corresponding receive ($QIO with the IO$_READVBLK function or $INPUT).

In addition, the tasks must ensure that enough buffer space is allocated for messages, must ensure that the end of dialogue can be determined, and must decide which of the tasks will disconnect the logical link ($DASSGN).

Disconnecting the Logical Link

Either task can disconnect the logical link by calling $DASSGN. $DASSGN aborts all pending sends and receives, disconnects the link immediately, and frees the channel number associated with the logical link.

MACRO CALLS

Listed below are the VAX/VMS system service macro calls that can be used for transparent intertask communications.

- $ASSIGN—Assign I/O Channel
- $QIO—Send a Message to a Remote Task $QIO (IO$_WRITEVBLK)
- $QIO—Receive a Message from a Remote Task $QIO (IO$_READVBLK)
- $INPUT—Read a Message
- $OUTPUT—Write a Message
- $DASSGN—Disconnect the Logical Link

MACRO NONTRANSPARENT INTERTASK COMMUNICATION

Nontransparent intertask communication may consist of two or more tasks interacting to establish a logical link. After establishing the logical link, the tasks exchange messages over the link, then disconnect the link when communication is complete.

The MACRO system service calls discussed in this section provide the user with greater flexibility and control over network operations. The following features can be used when performing nontransparent intertask communication:

- Associate a mailbox with the I/O channel (over which the logical link will be created). The mailbox can then receive mailbox messages sent by a remote task, or notifications affecting the status of the logical link. For example, status returned through a mailbox includes whether the remote task accepted or rejected a connect, or the cooperating task disconnected or destroyed the link.
- A task can declare itself as a network task to accept multiple logical link connect requests.
- A source task can send a logical link connect request to the target task. The source task can optionally send up to 16 bytes of data to the target task at the same time it issues the connect request.
- The target task can accept or reject the connect request. It can send up to 16 bytes of optional data back to the source task at the same time it accepts or rejects the connect request.
- A task using the nontransparent interface can also accept or reject connect requests received from tasks written using transparent intertask communication system service calls.
- Either task can send or receive a 1- to 16-byte interrupt message after the logical link is created.
- Either task can abort the link immediately, or issue a synchronous disconnect. The task disconnecting or aborting the logical link can send up to 16 bytes of op-

tional data to the remote task at the same time it disconnects or aborts a logical link.

Task Messages

There are two types of messages in nontransparent intertask communications: data messages and mailbox messages.

Data Messages — A data message is a message sent by one task, and expected by the cooperating task. That is, for each message sent by a task $QIO with the IO$_WRITEBLK function or $OUTPUT, the receiving task must issue a receive $QIO with the IO$_READVBLK function or $INPUT.

Thus, a data message in nontransparent intertask communications is the same as a data message sent in transparent communication.

Mailbox Messages — All other messages received by a task employing a nontransparent interface are classified as mailbox messages. These include any one of the following message types:

1. A logical link connect request—this message is received by the target task. It requests a logical link connection to the source task.
2. A connect accept—this message is received by the source task. The message confirms that the target task accepted the logical link connect request.
3. A connect reject—this message is also received by the source task. The message informs the source task that the target rejected the logical link connect request.
4. An interrupt message—either task can receive a 1- to 16-byte interrupt message sent by a cooperating task. The 1 to 16 bytes of data are placed in the task's mailbox.
5. A synchronous disconnect—this message informs the task that the cooperating task synchronously disconnected the logical link.
6. A disconnect abort—this message informs the task that the cooperating task aborted the link. The link is destroyed immediately.
7. A network status message—this message informs the task of some unusual network occurrence, for example, the data link has been restarted.

After a logical link is created between cooperating tasks, DECnet places a received mailbox message into the mailbox associated with the channel representing the logical link to which the mailbox message applies.

In the case of a task that can accept multiple inbound connect requests, inbound connect requests are placed into the mailbox associated with the I/O channel over which the network name was declared.

Note that the mailbox was previously created using the $CREMBX system service call. The task must then explicitly retrieve the unsolicited message from the mailbox using the $QIO(IO$_READVBLK) system service call.

PROTOCOL EMULATORS (INTERNETS)

VAX/VMS supports a number of software emulators that enable and promote coexistence between the VAX family and products supplied by other vendors. In this way, VAX computers become even more flexible, particularly in extending existing mainframe facilities to include powerful minicomputer data processing.

VAX-11 2780/3780 Protocol Emulator

This product provides the VAX/VMS user with a mechanism for transferring files between the VAX system and another system equipped to handle IBM 2780 or 3780 communications protocols. It does this by emulating the synchronous line protocol used by a 2780 or 3780 Remote Batch Terminal.

The emulator may be invoked either interactively or by a command procedure. The emulator's command set is designed to facilitate sharing a communication line among several users. With the appropriate modem options, the emulator is capable of automatically answering incoming calls.

Sophisticated operations can be performed by a combination of command procedures, allowing, for example, unattended operation. This would include the capability to detect an incoming call, establish the connection, and then transmit and receive files and recover from transmission failures, all without the intervention of the operator.

Several data formats are supported with the use of a particular format selected by user command. Users may select various forms to control translation schemes (records can be padded with spaces to card images before transmission), translation to and from EBCDIC, and BSC transparency. All file I/O is performed through the VAX/VMS record management facility. Print and punch stream recognition is implemented in such a way that the data manipulation scheme can differ with each stream.

The following remote batch terminal features are supported:

- 2780 Extended and Multiple Record Option
- Variable Horizontal Forms Control
- BSC Transparency
- 3780 Space Compression

All of the above features are supported on a simultaneous, multiline basis. The product can concurrently run up to four physical lines, each with a different set of attributes (e.g., some may be 2780, others, 3780) at speeds up to 9600 baud per line.

MUX200/VAX Multiterminal Emulator

MUX200/VAX is a VAX-based software package which provides communication with a CDC6000, CYBER series, or other host computer systems capable of using 200 UT mode 4A communications protocols.

Any VAX interactive terminal may be used to control remote job entry or to communicate at command level with the host system. Input files may be sent from, and output files received onto, any VAX-supported mass storage, unit record, or terminal device.

MUX200/VAX communicates with the host using the Mode 4A communications protocol as defined in CDC publication 82128000. The software package can be configured to support either the ASCII or the external BCD versions of the protocol.

MUX200/VAX provides for one synchronous communication circuit to a host computer system. The product sup-

ports a single switched or dedicated leased line two- or four-wire common carrier facility at speeds up to 9600 baud.

MUX200/VAX enables several users to communicate simultaneously with a host system over a single line. The VAX/VMS system, though using a single physical drop, appears to the host as a number of multidrops and terminals on the circuit.

MUX200/VAX features include:

- Output received from the host system may be spooled to the line printer upon detection of a text string predefined by the user.
- Up to eight VAX/VMS files may be specified for transmission to the host in a single command.
- VAX/VMS terminals may be detached for other use while the software package is operating. Data received from the host directed to a terminal are saved for printing until the terminal is reattached.
- In many applications the host system can be offloaded by taking advantage of the local processing power of the VAX/VMS system. This reduces host processing and line costs; for example, file editing can be performed locally rather than on the host.

Figure 10-6 illustrates a schematic of the MUX200.

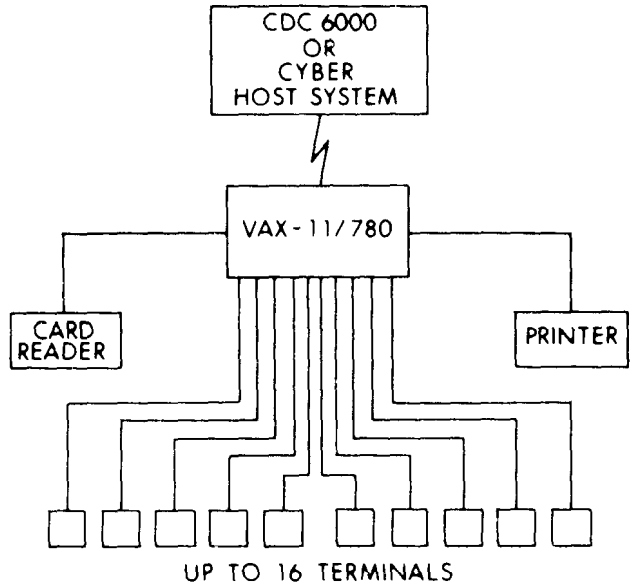


Figure 10-6
MUX200 Schematic

APPENDIX A

COMMONLY USED MNEMONICS

ACP	Ancillary Control Process
ANS	American National Standard
ASCII	American Standard Code for Information Interchange
AST	Asynchronous System Trap
ASTLVL	Asynchronous System Trap level
CCB	Channel Control Block
CM	Compatibility Mode bit in the hardware PSL
CRB	Channel Request Block
CRC	Cyclic Redundancy Check
DAP	Data Access Protocol
DDB	Device Data Block
DDCMP	DIGITAL Data Communications Message Protocol
DDT	Driver Data Table
DV	Decimal Overflow trap enable bit in the PSW
ECB	Exit Control Block
ECC	Error Correction Code
ESP	Executive Mode Stack Pointer
ESR	Exception Service Routine
F11ACP	Files-11 Ancillary Control Process
FAB	File Access Block
FCA	Fixed Control Area
FCB	File Control Block
FCS	File Control Services
FDT	Function Decision Table
FP	Frame pointer
FPD	First Part (of an instruction) Done
FU	Floating Underflow trap enable bit in the PSW
GSD	Global Section Descriptor
GST	Global Symbol Table
IDB	Interrupt Dispatch Block
IPL	Interrupt Priority Level
IRP	I/O Request Packet
ISECT	Image Section
ISD	Image Section Descriptor
ISP	Interrupt Stack Pointer
IS	Interrupt Stack bit in PSL
ISR	Interrupt Service Routine
IV	Integer Overflow trap enable bit in the PSW
KSP	Kernel Mode Stack Pointer
MBA	MASSBUS Adapter
MBZ	Must Be Zero
MCR	Monitor Console Routine
MFD	Master File Directory
MFPR	Move From Process Register instruction
MME	Memory Mapping Enable
MTPR	Move To Process Register instruction
MUTEX	Mutual Exclusion semaphore
NSP	Network Services Protocol
OPCOM	Operator Communication Manager
P0BR	Program region base register
P0LR	Program region length register
P0PT	Program region page table
P1BR	Control region base register
P1LR	Control region limit register
P1PT	Control region page table
PC	Program Counter
PCB	Process Control Block
PCBB	Process Control Block Base register
PFN	Page Frame Number
PID	Process Identification Number
PME	Performance Monitor Enable bit in PCB
PSECT	Program Section
PSL	Processor Status Longword
PSW	Processor Status Word
PTE	Page Table Entry
QIO	Queue Input/Output Request system service
RAB	Record Access Block
RFA	Record's File Address
RMS	Record Management Services
RWED	Read, Write, Execute, Delete
SBI	Synchronous Backplane Interconnect
SBR	System Base Register
SCB	System Control Block
SCBB	System Control Block Base register
SLR	System Length Register
SP	Stack Pointer
SPT	System Page Table
SSP	Supervisor Mode Stack Pointer
SVA	System virtual address
TP	Trace trap Pending bit in PSL
UBA	UNIBUS Adapter
UCB	Unit Control Block
UETP	User Environment Test Package
UFD	User File Directory
UIC	User Identification Code
USP	User Mode Stack Pointer
VCB	Volume Control Block
VPN	Virtual Page Number
WCB	Window Control Block
WCS	Writable Control Store
WDCS	Writable Diagnostic Control Store

APPENDIX B

```
IDENTIFICATION DIVISION.
PROGRAM-ID.        MERGE EXAMPLE.
*****************************************************************************
*     This program MERGEs three identically sequenced                       *
*     regional sales files into one total sales file.                       *
*     The program adds sales amounts and writes one                         *
*     record for each product-code.                                         *
*****************************************************************************
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.        VAX-11.
OBJECT-COMPUTER.        VAX-11.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
            SELECT REGION1-SALES          ASSIGN TO "REG1SLS".
            SELECT REGION2-SALES          ASSIGN TO "REG2SLS".
            SELECT REGION3-SALES          ASSIGN TO "REG3SLS".
            SELECT MERGE-FILE             ASSIGN TO "MRGFILE".
            SELECT TOTAL-SALES            ASSIGN TO "TOTLSLS".
DATA DIVISION.
FILE SECTION.
FD          REGION1-SALES
            LABEL RECORDS ARE STANDARD.
01          REGION1-RECORD                          PIC X(100).
FD          REGION2-SALES
            LABEL RECORDS ARE STANDARD.
01          REGION2-RECORD                          PIC X(100).
FD          REGION3-SALES
            LABEL RECORDS ARE STANDARD.
01          REGION3-RECORD                          PIC X(100).
SD          MERGE-FILE.
            01          MERGE-REC.
                        03 M-REGION-CODE           PIC XX.
                        03 M-PRODUCT-CODE          PIC X(10).
                        03 M-SALES-AMT             PIC S9(7)V99.
                        03 FILLER                  PIC X(79).
FD          TOTAL-SALES
            LABEL RECORDS ARE STANDARD.
01          TOTAL-RECORD                            PIC X(100).
WORKING-STORAGE SECTION.
01          INITIAL-READ            PIC X           VALUE "Y".
01          THE-COUNTERS.
            03 PRODUCT-AMT                         PIC S9(7)V99.
            03 REGION1-AMT                         PIC S9(9)V99.
            03 REGION2-AMT                         PIC S9(9)V99.
            03 REGION3-AMT                         PIC S9(9)V99.
            03 TOTAL-AMT                           PIC S9(11)V99.
01          SAVE-MERGE-REC.
            03 S-REGION-CODE                       PIC XX.
            03 S-PRODUCT-CODE                      PIC X(10).
            03 S-SALES-AMT                         PIC S9(7)V99.
            03 FILLER                              PIC X(79).
PROCEDURE DIVISION.
000-START SECTION.
```

```
010-MERGE-FILES.
      OPEN OUTPUT TOTAL-SALES.
      MERGE MERGE-FILE ON ASCENDING KEY M-PRODUCT-CODE
            USING REGION1-SALES REGION2-SALES REGION3-SALES
            OUTPUT PROCEDURE IS          020-BUILD-TOTAL-SALES
                  THRU                   100-DONE-TOTAL-SALES.
      DISPLAY "TOTAL SALES FOR REGION 1 "
REGION1-AMT.
      DISPLAY "TOTAL SALES FOR REGION 2 "
REGION2-AMT.
      DISPLAY "TOTAL SALES FOR REGION 3 "
REGION3-AMT.
      DISPLAY "TOTAL ALL SALES "           TOTAL-AMT.
      CLOSE TOTAL-SALES.
      DISPLAY "END OF PROGRAM MERGE01".
      STOP RUN.
020-BUILD-TOTAL-SALES SECTION.
030-GET-MERGE-RECORDS.
      RETURN MERGE-FILE AT END
            MOVE PRODUCT-AMT TO S-SALES-AMT
            WRITE TOTAL-RECORD FROM SAVE-MERGE-REC
            GO TO 100-DONE-TOTAL SALES.
      IF INITIAL-READ = "Y"
            MOVE "N" TO INITIAL-READ
            MOVE MERGE-REC TO SAVE-MERGE-REC
            PERFORM 050-TALLY-AMOUNTS
            GO TO 030-GET-MERGE-RECORDS.
040-COMPARE-PRODUCT-CODE.
      IF M-PRODUCT-CODE = S-PRODUCT-CODE
            PERFORM 050-TALLY-AMOUNTS
            GO TO 030-GET-MERGE-RECORDS.
      MOVE PRODUCT-AMT TO S-SALES-AMT.
      MOVE ZEROES TO PRODUCT-AMT.
      WRITE TOTAL-RECORD FROM SAVE-MERGE-REC.
      MOVE MERGE-REC TO SAVE-MERGE-REC.
      GO TO 040-COMPARE-PRODUCT-CODE.
050-TALLY-AMOUNTS.
      ADD M-SALES-AMT TO PRODUCT-AMT TOTAL-AMT.
      IF M-REGION-CODE = "01"
            ADD M-SALES-AMT TO REGION1-AMT.
      IF M-REGION-CODE = "02"
            ADD M-SALES-AMT TO REGION2-AMT.
      IF M-REGION-CODE = "03"
            ADD M-SALES-AMT TO REGION3-AMT.
100-DONE-TOTAL-SALES SECTION.
120-DONE.
      EXIT.
```

APPENDIX C

VT100 examples:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.         VIDEO3.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.        VAX-11.
OBJECT-COMPUTER.        VAX-11.
SPECIAL-NAMES.
          SYMBOLIC CHARACTERS      ESCAPER   PARM-1    PARM-2    PARM-3    PARM-4

                                ARE      28        92        60        103       75.

*****************************************************************************************
*     ESCAPER = ESC      PARM-1 = [ PARM-2 = ;   PARM-3=f   PARM-4 = J              *
*****************************************************************************************
INPUT-OUTPUT SECTION.
FILE-CONTROL.
          SELECT NAME-FILE ASSIGN TO "NAMFIL".
DATA DIVISION.
FILE SECTION.
FO              NAME - FILE
                LABEL RECORDS ARE STANDARD.
01              NAME-REC.
                03      N-CUST-NUM                  PIC X(8).
                03      N-CUST-NAME                 PIC X(25)
                03      N-ADDRESS                   PIC X(25)
                03      N-CITY                      PIC X(20)
                03      N-STATE                     PIC XX.
                03      N-ZIP                       PIC X(5).
WORKING-STORAGE SECTION.
01              HOME-UP.
                03              FILLER              PIC X       VALUE ESCAPER.
                03              FILLER              PIC X       VALUE PARM-1.
                03              H-LINE              PIC 99      VALUE 00.
                03              FILLER              PIC X       VALUE PARM-2.
                03              H-COL               PIC 99      VALUE 00.
                03              FILLER              PIC X       VALUE PARM-3.
01              CLEAR-SCREEN.
                03              FILLER              PIC X       VALUE ESCAPER.
                03              FILLER              PIC X       VALUE PARM-1.
                03              FILLER              PIC X       VALUE PARM-4.
01              THE-FORM.
                03              FORM-LINE-1.
                     04         FILLER                  PIC X           VALUE ESCAPER.
                     04         FILLER                  PIC X           VALUE PARM-1.
                     04         LINE-4                  PIC 99          VALUE 04.
                     04         FILLER                  PIC X           VALUE PARM-2.
                     04         COL-3                   PIC 99          VALUE 03.
                     04         FILLER                  PIC X           VALUE PARM-3.
                     04         FILLER                  PIC X(24)       VALUE
                                  "CUSTOMER NUMBER:_____".
     03              FORM-LINE-2.
                     04         FILLER                  PIC X           VALUE ESCAPER.
                     04         FILLER                  PIC X           VALUE PARM-1.
                     04         LINE-6                  PIC 99          VALUE 06.
                     04         FILLER                  PIC X           VALUE PARM-2.
                     04         COL-3                   PIC 99          VALUE 03.
                     04         FILLER                  PIC X           VALUE PARM-3.
                     04         FILLER                  PIC X(39)       VALUE
                                  "CUSTOMER NAME:"._____
```

```
    03          FORM-LINE-3.
                04      FILLER                  PIC X           VALUE ESCAPER.
                04      FILLER                  PIC X           VALUE PARM-1.
                04      LINE-8                  PIC 99          VALUE 08.
                04      FILLER                  PIC X           VALUE PARM-2.
                04      COL-3                   PIC 99          VALUE 03.
                04      FILLER                  PIC X           VALUE PARM-3.
                04      FILLER                  PIC X(42)       VALUE
                        "CUSTOMER ADDRESS:_____".
    03          FORM-LINE-4.
                04      FILLER                  PIC X           VALUE ESCAPER.
                04      FILLER                  PIC X           VALUE PARM-1.
                04      LINE-10                 PIC 99          VALUE 10.
                04      FILLER                  PIC X           VALUE PARM-2.
                04      COL-3                   PIC 99          VALUE 03.
                04      FILLER                  PIC X           VALUE PARM-3.
                04      FILLER                  PIC X(48)       VALUE
                    "CITY:_____     STATE:__    ZIP:____".
01          CURSOR-POSITIONER.
            03          FILLER                  PIC X       VALUE ESCAPER.
            03          FILLER                  PIC X       VALUE PARM-1.
            03          LINE-POS                PIC 99.
            03          FILLER                  PIC X       VALUE PARM-2.
            03          COL-POS                 PIC 99.
            03          FILLER                  PIC X       VALUE PARM-3.
01  INPUT-AREA                                 PIC X(25).
PROCEDURE DIVISION.
000-OPEN.
        OPEN OUTPUT NAME-FILE.
005-BEGIN.
        DISPLAY HOME-UP WITH NO ADVANCING.
010-CLEAR-SCREEN.
        DISPLAY CLEAR-SCREEN WITH NO ADVANCING.
020-PAINT-FORM.
        DISPLAY THE-FORM.
        GO TO 050-GET-INPUT-DATA.
050-GET-INPUT-DATA.
        MOVE SPACES TO INPUT-AREA.
*********************************************
* Position cursor to accept customer number.*
*********************************************
        MOVE 4 TO LINE-POS.
        MOVE 19 TO COL-POS.
        DISPLAY CURSOR-POSITIONER WITH NO ADVANCING.
        ACCEPT INPUT-AREA.
        IF INPUT-AREA = "DONE"
            PERFORM 005-BEGIN THRU 010-CLEAR-SCREEN
            CLOSE NAME-FILE STOP RUN.
        MOVE INPUT-AREA TO N-CUST-NUM.
        MOVE SPACES TO INPUT-AREA.
*********************************************
* Position cursor to accept customer name.*
*********************************************
        MOVE 6 TO LINE-POS.
        MOVE 17 TO COL-POS.
        DISPLAY CURSOR-POSITIONER WITH NO ADVANCING.
        ACCEPT INPUT-AREA.
        MOVE INPUT-AREA TO N-CUST-NAME.
        MOVE SPACES TO INPUT-AREA.
```

```
*************************************************
* Position cursor to accept customer address.*
*************************************************
           MOVE 8 TO LINE-POS.
           MOVE 20 TO COL-POS.
           DISPLAY CURSOR-POSITIONER WITH NO ADVANCING.
           ACCEPT INPUT-AREA.
           MOVE INPUT-AREA TO N-ADDRESS.
           MOVE SPACES TO INPUT-AREA.
***********************************
* Position cursor to accept city.  *
***********************************
           MOVE 10 TO LINE-POS.
           MOVE 8 TO COL-POS.
           DISPLAY CURSOR-POSITIONER WITH NO ADVANCING.
           ACCEPT INPUT-AREA.
           MOVE INPUT-AREA TO N-CITY.
           MOVE SPACES TO INPUT-AREA.
***********************************
* Position cursor to accept state.  *
***********************************
           MOVE 38 TO COL-POS.
           DISPLAY CURSOR-POSITIONER WITH NO ADVANCING.
           ACCEPT INPUT-AREA.
           MOVE INPUT-AREA TO N-STATE.
           MOVE SPACES TO N-STATE.
***********************************
* Position cursor to accept zip.  *
***********************************
           MOVE 46 TO COL-POS.
           DISPLAY CURSOR-POSITIONER WITH NO ADVANCING.
           ACCEPT INPUT-AREA.
           MOVE INPUT-AREA TO N-ZIP.
           WRITE NAME-REC.
           GO TO 005-BEGIN.
```

The Glossary

glos·sa·ry

abort An exception that occurs in the middle of an instruction and potentially leaves the registers and memory in an indeterminate state, such that the instruction cannot necessarily be restarted.

absolute indexed mode An indexed addressing mode in which the base operand specifier is addressed in absolute mode.

absolute mode In absolute mode addressing, the PC is used as the register in autoincrement deferred mode. The PC contains the address of the location containing the actual operand.

absolute time Time values expressing a specific date (month, day, and year) and time of day. Absolute time values are always expressed in the system as positive numbers.

access mode 1. Any of the four processor access modes in which software executes. Processor access modes are, in order from most to least privileged and protected: kernel (mode 0), executive (mode 1), supervisor (mode 2), and user (mode 3). When the processor is in kernel mode, the executing software has complete control of, and responsibility for, the system. When the processor is in any other mode, the processor is inhibited from executing privileged instructions. The Processor Status Longword contains the current access mode field. The operating system uses access modes to define protection levels for software executing in the context of a process. For example, the Executive runs in kernel and executive mode and is most protected. The command interpreter is less protected and runs in supervisor mode. The debugger runs in user mode and is no more protected than normal user programs. 2. See record access mode.

access type 1. The way in which the processor accesses instruction operands. Access types are: read, write, modify, address, and branch. 2. The way in which a procedure accesses its arguments. 3. See also record access type.

access violation An attempt to reference an address that is not mapped into virtual memory or an attempt to reference an address that is not accessible by the current access mode.

account name A string that identifies a particular account used to accumulate data on a job's resource use. This name is the user's accounting charge number, not the user's UIC.

address A number used by the operating system and user software to identify a storage location. See also virtual address and physical address.

address access type The specified operand of an instruction is not directly accessed by the instruction. The address of the specified operand is the actual instruction operand. The context of the address calculation is given by the data type of the operand.

addressing mode The way in which an operand is specified; for example, the way in which the effective address of an instruction operand is calculated using the general registers. The basic general register addressing modes are called: register, register deferred, autoincrement, autoincrement deferred, autodecrement, displacement, and displacement deferred. In addition, there are six indexed addressing modes using two general registers, and literal mode addressing. The PC addressing modes are called: immediate (for register deferred mode using the PC), absolute (for autoincrement deferred mode using the PC), and branch.

address space The set of all possible addresses available to a process. Virtual address space refers to the set of all possible virtual addresses. Physical address space refers to the set of all possible physical addresses sent out on the SBI.

allocate a device To reserve a particular device unit for exclusive use. A user process can allocate a device only when that device is not allocated by any other process.

alphanumeric character An upper or lower case letter (A-Z, a-z), a dollar sign ($), an underscore (_), or a decimal digit (0-9).

alternate key An optional key within the data records in an indexed file; used by VAX-11 RMS to build an alternate index. See key (indexed files) and primary key.

American Standard Code for Information Interchange (ASCII) A set of 8-bit binary numbers representing the alphabet, punctuation, numerals, and other special symbols used in text representation and communications protocol.

Ancillary Control Process (ACP) A process that acts as an interface between user software and an I/O driver. An ACP provides functions supplemental to those performed in the driver, such as file and directory management. Three examples of ACPs are: the Files-11 ACP (F11ACP), the magnetic tape ACP (MTACP), and the networks ACP (NETACP).

area Areas are VAX-11 RMS maintained regions of an indexed file. They allow a user to specify placement and/or specific bucket sizes for particular portions of a file. An area consists of any number of buckets, and there may be from 1 to 255 areas in a file.

Argument Pointer General register 12 (R12). By convention, AP contains the address of the base of the argument list for procedures initiated using the CALL instructions.

assign a channel To establish the necessary software linkage between a user process and a device unit before a user process can transfer any data to or from that device. A user process requests the system to assign a channel and the system returns a channel number.

asynchronous record operation A mode of record processing in which a user program can continue to execute after issuing a record retrieval or storage request without having to wait for the request to be fulfilled.

Asynchronous System Trap A software-simulated interrupt to a user-defined service routine. ASTs enable a user process to be notified asynchronously with respect to its execution of the occurrence of a specific event. If a user process has defined an AST routine for an event, the system interrupts the process and executes the AST routine when that event occurs. When the AST routine exits, the system resumes the process at the point where it was interrupted.

Asynchronous System Trap level (ASTLVL) A value kept in an internal processor register that is the highest access mode for which an AST is pending. The AST does not occur until the current access mode drops in priority (raises in numeric value) to a value greater than or equal to ASTLVL. Thus, an AST for an access mode will not be serviced while the processor is executing in a higher priority access mode.

authorization file See user authorization file.

autodecrement indexed mode An indexed addressing mode in which the base operand specifier uses autodecrement mode addressing.

autodecrement mode In autodecrement mode addressing, the contents of the selected register are decremented, and the result is used as the address of the actual operand for the instruction. The contents of the register are decremented according to the data type context of the register: 1 for byte, 2 for word, 4 for longword and floating, 8 for quadword and double floating.

autoincrement deferred indexed mode An indexed addressing mode in which the base operand specifier uses autoincrement deferred mode addressing.

autoincrement deferred mode In autoincrement deferred mode addressing, the specified register contains the address of a longword which contains the address of the actual operand. The contents of the register are incremented by 4 (the number of bytes in a longword). If the PC is used as the register, this mode is called absolute mode.

autoincrement indexed mode An indexed addressing mode in which the base operand specifier uses autoincrement mode addressing.

autoincrement mode In autoincrement mode addressing, the contents of the specified register are used as the address of the operand, then the contents of the register are incremented by the size of the operand.

balance set The set of all process working sets currently resident in physical memory. The processes whose working sets are in the balance set have memory requirements that balance with available memory. The balance set is maintained by the system swapper process.

base operand address The address of the base of a table or array referenced by index mode addressing.

base operand specifier The register used to calculate the base operand address of a table or array referenced by index mode addressing.

base priority The process priority that the system assigns a process when it is created. The scheduler never schedules a process below its base priority. The base priority can be modified only by the system manager or the process itself.

base register A general register used to contain the address of the first entry in a list, table, array, or other data structure.

binding See linking.

bit string See variable-length bit field.

block 1. The smallest addressable unit of data that the specified device can transfer in an I/O operation (512 contiguous bytes for most disk devices). 2. An arbitrary number of contiguous bytes used to store logically related status, control, or other processing information.

block I/O The set of VAX-11 RMS procedures that allow you direct access to the blocks of a file regardless of file organization.

bootstrap block A block in the index file on a system disk that contains a program that can load the operating system into memory and start its execution.

branch access type An instruction attribute which indicates that the processor does not reference an operand address, but that the operand is a branch displacement. The size of the branch displacement

is given by the data type of the operand.

branch mode In branch addressing mode, the instruction operand specifier is a signed byte or word displacement. The displacement is added to the contents of the updated PC (which is the address of the first byte beyond the displacement), and the result is the branch address.

bucket A storage structure, consisting of from 1 to 32 blocks, used for building and processing relative and indexed files. A bucket contains one or more records or record cells. Buckets are the unit of contiguous transfer between VAX-11 RMS buffers and the disk.

buffered I/O See system buffered I/O.

bug check The operating system's internal diagnostic check. The system logs the failure and crashes the system.

byte A byte is eight contiguous bits starting on an addressable byte boundary. Bits are numbered from the right, 0 through 7, with bit 0 the low-order bit. When interpreted arithmetically, a byte is a 2's complement integer with significance increasing from bits 0 through 6. Bit 7 is the sign bit. The value of the signed integer is in the range −128 to 127 decimal. When interpreted as an unsigned integer, significance increases from bits 0 through 7 and the value of the unsigned integer is in the range 0 to 255 decimal. A byte can be used to store one ASCII character.

cache memory A small, high-speed memory placed between slower main memory and the processor. A cache increases effective memory transfer rates and processor speed. It contains copies of data recently used by the processor, and fetches several bytes of data from memory in anticipation that the processor will access the next sequential series of bytes.

call frame See stack frame.

call instructions The processor instructions CALLG (Call Procedure with General Argument List) and CALLS (Call Procedure with Stack Argument List).

call stack The stack, and conventional stack structure, used during a procedure call. Each access mode of each process context has one call stack, and interrupt service context has one call stack.

channel A logical path connecting a user process to a physical device unit. A user process requests the operating system to assign a channel to a device so the process can transfer data to or from that device.

character A symbol represented by an ASCII code. See also alphanumeric character.

character string A contiguous set of bytes. A character string is identified by two attributes: an address and a length. Its address is the address of the byte containing the first character of the string. Subsequent characters are stored in bytes of increasing addresses. The length is the number of characters in the string.

character string descriptor A quadword data structure used for passing character data (strings). The first word of the quadword contains the length of the character string. The second word can contain type information. The remaining longword contains the address of the string.

cluster 1. A set of contiguous blocks that is the basic unit of space allocation on a Files-11 disk volume. 2. A set of pages brought into memory in one paging operation. 3. An event flag cluster.

command An instruction, generally an English word, typed by the user at a terminal or included in a command file which requests the software monitoring a terminal or reading a command file to perform some well-defined activity. For example, typing the COPY command requests the system to copy the contents of one file into another file.

command file A file containing command strings. See also command procedure.

command interpreter Procedure-based system code that executes in supervisor mode in the context of a process to receive, syntax check, and parse commands typed by the user at a terminal or submitted in a command file.

command parameter The positional operand of a command delimited by spaces, such as a file specification, option, or constant.

command procedure A file containing commands and data that the command interpreter can accept in lieu of the user typing the commands individually on a terminal.

command string A line (or set of continued lines), normally terminated by typing the carriage return key, containing a command and, optionally, information modifying the command. A complete command string consists of a command, its qualifiers, if any, and its parameters (file specifications, for example), if any, and their qualifiers, if any.

common A FORTRAN term for a program section that contains only data.

common event flag cluster A set of 32 event flags that enables cooperating processes to post event notification to each other. Common event flag clus-

ters are created as they are needed. A process can associate with up to two common event flag clusters.

compatibility mode A mode of execution that enables the central processor to execute non-privileged PDP-11 instructions. The operating system supports compatibility mode execution by providing an RSX-11M programming environment for an RSX-11M task image. The operating system compatibility mode procedures reside in the control region of the process executing a compatibility mode image. The procedures intercept calls to the RSX-11M Executive and convert them to the appropriate operating system functions.

condition An exception condition detected and declared by software. For example, see failure exception mode.

condition codes Four bits in the Processor Status Word that indicate the results of previously executed instructions.

condition handler A procedure that a process wants the system to execute when an exception condition occurs. When an exception condition occurs, the operating system searches for a condition handler and, if found, initiates the handler immediately. The condition handler may perform some action to change the situation that caused the exception condition and continue execution for the process that incurred the exception condition. Condition handlers execute in the context of the process at the access mode of the code that incurred the exception condition.

condition value A 32-bit quantity that uniquely identifies an exception condition.

context The environment of an activity. See also process context, hardware context, and software context.

context indexing The ability to index through a data structure automatically because the size of the data type is known and used to determine the offset factor.

context switching Interrupting the activity in progress and switching to another activity. Context switching occurs as one process after another is scheduled for execution. The operating system saves the interrupted process' hardware context in its hardware process control block (PCB) using the Save Process Context instruction, and loads another process' hardware PCB into the hardware context using the Load Process Context instruction, scheduling that process for execution.

continuation character A hyphen at the end of a command line signifying that the command string continues on to the next command line.

console The manual control unit integrated into the central processor. The console includes an LSI-11 microprocessor and a serial line interface connected to a hard copy terminal. It enables the operator to start and stop the system, monitor system operation, and run diagnostics.

console terminal The hard copy terminal connected to the central processor console.

control region The highest-addressed half of per-process space (the P1 region). Control region virtual addresses refer to the process-related information used by the system to control the process, such as: the kernel, executive, and supervisor stacks, the permanent I/O channels, exception vectors, and dynamically used system procedures (such as the command interpreter and RSX-11M programming environment compatibility mode procedures). The user stack is also normally found in the control region, although it can be relocated elsewhere.

Control Region Base Register (P1BR) The processor register, or its equivalent in a hardware process control block, that contains the base virtual address of a process control region page table.

Control Region Length Register (P1LR) The processor register, or its equivalent in a hardware process control block, that contains the number of non-existent page table entries for virtual pages in a process control region.

copy-on-reference A method used in memory management for sharing data until a process accesses it, in which case it is copied before the access. Copy-on-reference allows sharing of the initial values of a global section whose pages have read/write access but contain pre-initialized data available to many processes.

counted string A data structure consisting of a byte-sized length followed by the string. Although a counted string is not used as a procedure argument, it is a convenient representation in memory.

current access mode The processor access mode of the currently executing software. The Current Mode field of the Processor Status Longword indicates the access mode of the currently executing software.

cylinder The tracks at the same radius on all recording surfaces of a disk.

D_floating (point) datum A floating point datum consisting of B contiguous bytes (64 bits) starting on an arbitrary byte boundary. The value of the D_floating datum is in the approximate range (+ or −) 0.29×10^{-38} to 1.7×10^{38}. The precision is approximately one part in 2^{55} or typically sixteen decimal digits.

data base 1. All the occurrences of data described by a data base management system. 2. A collection of related data structures.

data structure Any table, list, array, queue, or tree whose format and access conventions are well-defined for reference by one or more images.

data type In general, the way in which bits are grouped and interpreted. In reference to the processor instructions, the data type of an operand identifies the size of the operand and the significance of the bits in the operand. Operand data types include: byte, word, longword and quadword integer, floating and double floating, character string, packed decimal string, and variable-length bit field.

deferred echo Refers to the fact that terminal echoing does not occur until a process is ready to accept input entered by type ahead.

delta time A time value expressing an offset from the current date and time. Delta times are always expressed in the system as negative numbers whose absolute value is used as an offset from the current time.

demand zero page A page, typically of an image stack or buffer area, that is initialized to contain all zeros when dynamically created in memory as a result of a page fault. This feature eliminates the waste of disk space that would otherwise be required to store blocks (pages) that contain only zeros.

descriptor A data structure used in calling sequences for passing argument types, addresses and other optional information. See character string descriptor.

detached process A process that has no owner. The parent process of a tree of subprocesses. Detached processes are created by the job controller when a user logs on the system or when a batch job is initiated. The job controller does not own the user processes it creates; these processes are therefore detached.

device The general name for any physical terminus or link connected to the processor that is capable of receiving, storing, or transmitting data. Card readers, line printers, and terminals are examples of record-oriented devices. Magnetic tape devices and disk devices are examples of mass storage devices. Terminal line interfaces and interprocessor links are examples of communications devices.

device interrupt An interrupt received on interrupt priority level 16 through 23. Device interrupts can be requested only by devices, controllers, and memories.

device name The field in a file specification that identifies the device unit on which a file is stored. Device names also include the mnemonics that identify an I/O peripheral device in a data transfer request. A device name consists of a mnemonic followed by a controller identification letter (if applicable), followed by a unit number (if applicable), and ends with a colon (:).

device queue See spool queue.

device register A location in device controller logic used to request device functions (such as I/O transfers) and/or report status.

device unit One drive, and its controlling logic, of a mass storage device system. A mass storage system can have several drives connected to it.

diagnostic A program that tests logic and reports any faults it detects.

direct I/O An I/O operation in which the system locks the pages containing the associated buffer in memory for the duration of the I/O operation. The I/O transfer takes place directly from the process buffer. Contrast with system buffered I/O.

direct mapping cache A cache organization in which only one address comparison is needed to locate any data in the cache because any block of main memory data can be placed in only one possible position in the cache. Contrast with fully associative cache.

directory A file used to locate files on a volume that contains a list of file names (including type and version number) and their unique internal identifications.

directory name The field in a file specification that identifies the directory file in which a file is listed. The directory name begins with a left bracket ([or <) and ends with a right bracket (] or >).

displacement deferred indexed mode An indexed addressing mode in which the base operand specifier uses displacement deferred mode addressing.

displacement deferred mode In displacement deferred mode addressing, the specifier extension is a byte, word, or longword displacement. The displacement is sign extended to 32 bits and added to a base address obtained from the specified register. The result is the address of a longword which contains the address of the actual operand. If the PC is used as the register, the updated contents of the PC are used as the base address. The base address is the address of the first byte beyond the specifier extension.

displacement indexed mode An indexed addressing mode in which the base operand specifier uses displacement mode addressing.

displacement mode In displacement mode addressing, the specifier extension is a byte, word, or longword displacement. The displacement is sign extended to 32 bits and added to a base address obtained from the specified register. The result is the address of the actual operand. If the PC is used as the register, the updated contents of the PC are used as the base address. The base address is the address of the first byte beyond the specifier extension.

drive The electro-mechanical unit of a mass storage device system on which a recording medium (disk cartridge, disk pack, or magnetic tape reel) is mounted.

driver The set of code that handles physical I/O to a device.

dynamic access A technique in which a program switches from one record access mode to another while processing a file.

echo A terminal handling characteristic in which the characters typed by the terminal user on the keyboard are also displayed on the screen or printer.

effective address The address obtained after indirect or indexing modifications are calculated.

entry mask A word whose bits represent the registers to be saved or restored on a subroutine or procedure call using the call and return instructions.

entry point A location that can be specified as the object of a call. It contains an entry mask and exception enables known as the entry point mask.

equivalence name The string associated with a logical name in a logical name table. An equivalence name can be, for example, a device name, another logical name, or a logical name concatenated with a portion of a file specification.

error logger A system process that empties the error log buffers and writes the error messages into the error file. Errors logged by the system include memory system errors, device errors and timeouts, and interrupts with invalid vector addresses.

escape sequence An escape is a transition from the normal mode of operation to a mode outside the normal mode. An escape character is the code that indicates the transition from normal to escape mode. An escape sequence refers to the set of character combinations starting with an escape character that the terminal transmits without interpretation to the software set up to handle escape sequences.

event A change in process status or an indication of the occurrence of some activity that concerns an individual process or cooperating processes. An incident reported to the scheduler that affects a process' ability to execute. Events can be synchronous with the process' execution (a wait request), or they can be asynchronous (I/O completion). Some other events include: swapping; wake request; page fault.

event flag A bit in an event flag cluster that can be set or cleared to indicate the occurrence of the event associated with that flag. Event flags are used to synchronize activities in a process or among many processes.

event flag cluster A set of 32 event flags which are used for event posting. Four clusters are defined for each process: two process-local clusters, and two common event flag clusters. Of the process-local flags, eight are reserved for system use.

exception An event detected by the hardware (other than an interrupt or jump, branch, case, or call instruction) that changes the normal flow of instruction execution. An exception is always caused by the execution of an instruction or set of instructions (whereas an interrupt is caused by an activity in the system independent of the current instruction). There are three types of hardware exceptions: traps, faults, and aborts. Examples are: attempts to execute a privileged or reserved instruction, trace faults, compatibility mode faults, breakpoint instruction execution, and arithmetic faults such as floating point overflow, underflow, and divide by zero.

exception condition A hardware- or software-detected event other than an interrupt or jump, branch, case, or call instruction that changes the normal flow of instruction execution.

exception dispatcher An operating system procedure that searches for a condition handler when an exception condition occurs. If no exception handler is found for the exception or condition, the image that incurred the exception is terminated.

exception enables See trap enables.

exception vector See vector.

executable image An image that is capable of being run in a process. When run, an executable image is read from a file for execution in a process.

executive The generic name for the collection of procedures included in the operating system software that provides the basic control and monitoring functions of the operating system.

executive mode The second most privileged processor access mode (mode 1). The record management services (RMS) and many of the operating

system's programmed service procedures execute in executive mode.

exit An image exit is a rundown activity that occurs when image execution terminates either normally or abnormally. Image rundown activities include deassigning I/O channels and disassociation of common event flag clusters. Any user- or system-specified exit handlers are called.

exit handler A procedure executed when an image exits. An exit handler enables a procedure that is not on the call stack to gain control and clean up procedure-own data bases before the actual image exit occurs.

extended attribute block (XAB) An RMS user data structure that contains additional file attributes beyond those expressed in the file access block (FAB), such as boundary types (aligned on cylinder, logical block number, virtual block number) and file protection information.

extension The amount of space to allocate at the end of a file each time a sequential write exceeds the allocated length of the file.

extent The contiguous area on a disk containing a file or a portion of a file. Consists of one or more clusters.

F_floating (point) datum A floating point datum consisting of 4 contiguous bytes (32 bits) starting on an arbitrary byte boundary. The value of the F_floating datum is in the approximate range (+ or −) 0.29×10^{-38} to 1.7×10^{38}. The precision is approximately one part in 2^{23} or typically seven decimal digits.

failure exception mode A mode of execution selected by a process indicating that it wants an exception condition declared if an error occurs as the result of a system service call. The normal mode is for the system service to return an error status code for which the process must test.

fault A hardware exception condition that occurs in the middle of an instruction and that leaves the registers and memory in a consistent state, such that elimination of the fault and restarting the instruction will give correct results.

field 1. See variable-length bit field. 2. A set of contiguous bytes in a logical record.

file An organized collection of related items (records) maintained in an accessible storage area, such as disk or tape.

file access block (FAB) An RMS user data structure that represents a request for data operations related to the file as a whole, such as OPEN, CLOSE, or CREATE.

file header A block in the index file describing a file on a Files-11 disk structure. The file header identifies the locations of the file's extents. There is a file header for every file on the disk.

file name The field preceding a file type in a file specification that contains a 1- to 9-character logical name for a file.

filename extension See file type.

file organization The physical arrangement of data in the file. You select the specific organization from those offered by VAX-11 RMS, based on your individual needs for efficient data storage and retrieval. See indexed file organization, relative file organization, and sequential file organization.

Files-11 The name of the on-disk structure used by the RSX-11, IAS and VAX/VMS operating systems. Volumes created under this structure are transportable between these operating systems.

file specification A unique name for a file on a mass storage medium. It identifies the node, the device, the directory name, the file name, the file type, and the version number under which a file is stored.

file structure The way in which the blocks forming a file are distributed on a disk or magnetic tape to provide a physical accessing technique suitable for the way in which the data in the file is processed.

file system A method of recording, cataloging, and accessing files on a volume.

file type The field in a file specification that is preceded by a period or dot (.) and consists of a zero- to three-character type identification. By convention, the type identifies a generic class of files that have the same use or characteristics, such as ASCII text files, binary object files, etc.

fixed control area An area associated with a variable length record available for controlling or assisting record access operations. Typical uses include line numbers and printer format control information.

fixed-length record format Property of a file in which all records are of the same size. This format provides simplicity in determining the exact location of a record in the file and eliminates the need to prefix a record size field to each record.

floating (point) datum A numeric data type in which the number is represented by a fraction (less than 1 and greater than or equal to ½) multiplied by 2 raised to a power. There are four floating point data types: F_floating (4 bytes), D_floating (8 bytes), G_floating (8 bytes), and H_floating (16 bytes)

foreign volume Any volume other than a Files-11 formatted volume which may or may not be file structured.

fork process A dynamically created system process such as a process that executes device driver code or the timer process. Fork processes have minimal context. Fork processes are scheduled by the hardware rather than by the software. The timer process is dispatched directly by software interrupt. I/O driver processes are dispatched by a fork dispatcher. Fork processes execute at software interrupt levels and are dispatched for execution immediately. Fork processes remain resident until they terminate.

frame pointer General register 13 (R13). By convention, FP contains the base address of the most recent call frame on the stack.

fully associative cache A cache organization in which any block of data from main memory can be placed anywhere in the cache. Address comparision must take place against each block in the cache to find any particular block. Contrast with direct mapping cache.

G_floating (point) datum A floating point datum consisting of 8 contiguous bytes (64 bits) starting on an arbitrary byte boundary. The value of the G_floating datum is in the approximate range (+ or −) 0.56×10^{-308} to 9×10^{308}. The precision is approximately one part in 2^{52} or typically fifteen.

general register Any of the sixteen 32-bit registers used as the primary operands of the native mode instructions. The general registers include 12 general purpose registers which can be used as accumulators, as counters, and as pointers to locations in main memory, and the Frame Pointer (FP), Argument Pointer (AP), Stack Pointer (SP), and Program Counter (PC) registers.

generic device name A device name that identifies the type of device but not a particular unit; a device name in which the specific controller and/or unit number is omitted.

giga Metric term used to represent the number 1 followed by nine zeros.

global page table The page table containing the master page table entries for global sections.

global section A data structure (e.g., FORTRAN global common) or shareable image section potentially available to all processes in the system. Access is protected by privilege and/or group number of the UIC.

global symbol A symbol defined in a module that is potentially available for reference by another module. The linker resolves (matches references with definitions) global symbols. Contrast with local symbol.

global symbol table (GST) In a library, an index of strongly defined global symbols used to access the modules defining the global symbols. The linker will also put global symbol tables into an image. For example, the linker appends a global symbol table to executable images that are intended to run under the symbolic debugger, and it appends a global symbol table to all shareable images.

group 1. A set of users who have special access privileges to each other's directories and files within those directories (unless protected otherwise), as in the context "system, owner, group, world," where group refers to all members of a particular owner's group. 2. A set of jobs (processes and their subprocesses) who have access privileges to a group's common event flags and logical name tables, and may have mutual process controlling privileges, such as scheduling, hibernation, etc.

group number The first number in a User Identification Code (UIC).

H_floating (point) datum A floating point datum consisting of 16 contiguous bytes (128 bits) starting on an arbitrary byte boundary. The value of the H_floating datum is in the approximate range (+ or −) 0.84×10^{4932} to 0.59×10^{4932}. The precision is approximately one part in 2^{112} or typically 33 decimal digits.

hardware context The values contained in the following registers while a process is executing: the Program Counter (PC); the Processor Status Longword (PSL); the 14 general registers (R0 through R13); the four processor registers (P0BR, P0LR, P1BR and P1LR) that describe the process virtual address space; the Stack Pointer (SP) for the current access mode in which the processor is executing; plus the contents to be loaded in the Stack Pointer for every access mode other than the current access mode. While a process is executing, its hardware context is continually being updated by the processor. While a process is not executing, its hardware context is stored in its hardware PCB.

hardware process control block (PCB) A data structure known to the processor that contains the hardware context when a process is not executing. A process' hardware PCB resides in its process header.

hibernation A state in which a process is inactive, but known to the system with all of its current status. A hibernating process becomes active again when a wake request is issued. It can schedule a wake request before hibernating, or another process can issue its wake request. A hibernating process also becomes active for the time sufficient to service any AST it may receive while it is hibernating. Contrast with suspension.

home block A block in the index file that contains the volume identification, such as volume label and protection.

image An image consists of procedures and data that have been bound together by the linker. There are three types of images: executable, shareable, and system.

image activator A set of system procedures that prepare an image for execution. The image activator establishes the memory management data structures required both to map the image's virtual pages to physical pages and to perform paging.

image exit See exit.

image I/O segment That portion of the control region that contains the RMS internal file access blocks (IFAB) and I/O buffers for the image currently being executed by a process.

image name The file name of the file in which an image is stored.

image privileges The privileges assigned to an image when it is linked. See process privileges.

image section (isect) A group of program sections (psects) with the same attributes (such as read-only access, read/write access, absolute, relocatable, etc.) that is the unit of virtual memory allocation for an image.

immediate mode In immediate mode addressing, the PC is used as the register in autoincrement mode addressing.

index The structure which allows retrieval of records in an indexed file by key value. See key (indexed files).

index file The file on a Files-11 volume that contains the access information for all files on the volume and enables the operating system to identify and access the volume.

index file bit map A table in the index file of a Files-11 volume that indicates which file headers are in use.

index register A register used to contain an address offset.

indexed addressing mode In indexed mode addressing, two registers are used to determine the actual instruction operand: an index register and a base operand specifier. The contents of the index register are used as an index (offset) into a table or array. The base operand specifier supplies the base address of the array (the base operand address or BOA). The address of the actual operand is calculated by multiplying the contents of the index register by the size (in bytes) of the actual operand and adding the result to the base operand address. The addressing modes resulting from index mode addressing are formed by adding the suffix "indexed" to the addressing mode of the base operand specifier: register deferred indexed, autoincrement indexed, autoincrement deferred indexed (or absolute indexed), autodecrement indexed, displacement indexed, and displacement deferred indexed.

indexed file organization A file organization which allows random retrieval of records by key values and sequential retrieval of records in sorted order by key value. See key (indexed files).

indirect command file See command procedure.

input stream The source of commands and data. One of: the user's terminal, the batch stream, or an indirect command file.

instruction buffer A buffer in the processor used to contain bytes of the instruction currently being decoded and to pre-fetch instructions in the instruction stream. The control logic continously fetches data from memory to keep the buffer full.

interleaving Assigning consecutive physical memory addresses alternately between two memory controllers.

interlocked The property of a read followed by a write to the same datum with no possibility of an intervening reference by a second processor or I/O device. Examples are the Branch on Bit Interlocked and Add Aligned Word Interlocked instructions.

interprocess communication facility A common event flag, mailbox, or global section used to pass information between two or more processes.

interrecord gap A blank space deliberately placed between data records on the recording surface of a magnetic tape.

interrupt An event other than an exception or branch, jump, case, or call instruction that changes the normal flow of instruction execution. Interrupts are generally external to the process executing when the interrupt occurs. See also device interrupt, software interrupt, and urgent interrupt.

interrupt priority level (IPL) The interrupt level at which the processor executes when an interrupt is generated. There are 31 possible interrupt priority levels. IPL 1 is lowest, 31 highest. The levels arbitrate contention for processor service. For example, a device cannot interrupt the processor if the processor is currently executing at an interrupt priority level greater than the interrupt priority level of the device's interrupt service routine.

interrupt service routine The routine executed when a device interrupt occurs.

interrupt stack The system-wide stack used when executing in interrupt service context. At any time, the processor is either in a process context executing in user, supervisor, executive or kernel mode, or in system-wide interrupt service context operating with kernel privileges, as indicated by the interrupt stack and current mode bits in the PSL. The interrupt stack is not context switched.

interrupt stack pointer The stack pointer for the interrupt stack. Unlike the stack pointers for process context stacks, which are stored in the hardware PCB, the interrupt stack pointer is stored in an internal register.

interrupt vector See vector.

I/O driver See driver.

I/O function An I/O operation that is interpreted by the operating system and typically results in one or more physical I/O operations.

I/O function code A 6-bit value specified in a Queue I/O Request system service that describes the particular I/O operation to be performed (e.g., read, write, rewind).

I/O function modifier A 10-bit value specified in a Queue I/O Request system service that modifies an I/O function code (e.g., read terminal input no echo).

I/O lockdown The state of a page such that it cannot be paged or swapped out of memory until any I/O in progress to that page is completed.

I/O rundown An operating system function in which the system cleans up any I/O in progress when an image exits.

I/O space The region of physical address space that contains the configuration registers, and device control/status and data registers.

I/O status block A data structure associated with the Queue I/O Request system service. This service optionally returns a status code, number of bytes transferred, and device- and function-dependent information in an I/O status block. It is not returned from the service call, but filled in when the I/O request completes.

job 1. A job is the accounting unit equivalent to a process and the collection of all the subprocesses, if any, that it and its subprocesses create. Jobs are classified as batch and interactive. For example, the job controller creates an interactive job to handle a user's requests when the user logs onto the system and it creates a batch job when the symbiont manager passes a command input file to it. 2. A print job.

job controller The system process that establishes a job's process context, starts a process running the LOGIN image for the job, maintains the accounting record for the job, manages symbionts, and terminates a process and its subprocesses.

job queue A list of files that a process has supplied for processing by a specific device, for example, a line printer.

kernel mode The most privileged processor access mode (mode 0). The operating system's most privileged services, such as I/O drivers and the pager, run in kernel mode.

key
indexed files: A character string, a packed decimal number, a 2- or 4-byte unsigned binary number, or a 2- or 4-byte signed integer within each data record in an indexed file. You define the length and location within the records; VAX-11 RMS uses the key to build an index. See primary key, alternate key, and random access by key value.

relative files: The relative record number of each data record in a data file; VAX-11 RMS uses the relative record numbers to identify and access data records in a relative file in random access mode. See relative record number.

lexical function A command language construct that the command interpreter evaluates and substitutes before it performs expression analysis on a command string. Lexical functions return information about the current process, such as UIC or default directory; and about character strings, such as length or substring locations.

librarian A program that allows the user to create, update, modify, list, and maintain object library, image library, and assembler macro library files.

library file A direct access file containing one or more modules of the same module type.

limit The size or number of given items requiring system resources (such as mailboxes, locked pages, I/O requests, open files, etc.) that a job is allowed to have at any one time during execution, as specified by the system manager in the user authorization file. See also quota.

line number A number used to identify a line of text in a file processed by a text editor.

linker A program that reads one or more object files created by language processors and produces an executable image file, a shareable image file, or a system image file.

linking The resolution of external references between object modules used to create an image, the acquisition of referenced library routines, service entry points, and data for the image, and the assignment of virtual addresses to components of an image.

literal mode In literal mode addressing, the instruction operand is a constant whose value is expressed in a 6-bit field of the instruction. If the operand data type is byte, word, longword, quadword, or octaword, the operand is zero-extended and can express values in the range 0 through 63 (decimal). If the operand data type is F_, D_, G_, or H_floating, the 6-bit field is composed of two 3-bit fields, one for the exponent and the other for the fraction. The operand is extended to F_, D_, G_, or H_floating format.

locality See program locality.

local symbol A symbol meaningful only to the module that defines it. Symbols not identified to a language processor as global symbols are considered to be local symbols. A language processor resolves (matches references with definitions) local symbols. They are not known to the linker and cannot be made available to another object module. They can, however, be passed through the linker to the symbolic debugger. Contrast with global symbol.

locate mode Technique used for a record input operation in which the data records are not copied from the I/O buffer. See move mode.

locking a page in memory Making a page in an image ineligible for either paging or swapping. A page stays locked in memory until it is specifically unlocked.

locking a page in the working set Making a page in an image ineligible for paging out of the working set for the image. The page can be swapped when the process is swapped. A page stays locked in a working set until it is specifically unlocked.

logical block number A number used to identify a block on a mass storage device. The number is a volume-relative address rather than its physical (device-oriented) address or its virtual (file-relative) address. The blocks that constitute the volume are labeled sequentially starting with logical block 0.

logical I/O function A set of I/O operations (e.g., read and write logical block) that allow restricted direct access to device level I/O operations using logical block addresses.

logical name A user-specified name for any portion or all of a file specification. For example, the logical name INPUT can be assigned to a terminal device from which a program reads data entered by a user. Logical name assignments are maintained in logical name tables for each process, each group, and the system. A logical name can be created and assigned a value permanently or dynamically.

logical name table A table that contains a set of logical names and their equivalence names for a particular process, a particular group, or the system.

logical I/O functions A set of I/O functions that allow restricted direct access to device level I/O operations.

logical record A group of related fields treated as a unit.

longword Four contiguous bytes (32 bits) starting on an addressable byte boundary. Bits are numbered from right to left, 0 through 31. The address of the longword is the address of the byte containing bit 0. When interpreted arithmetically, a longword is a 2's complement integer with significance increasing from bit 0 to bit 30. When interpreted as a signed integer, bit 31 is the sign bit. The value of the signed integer is in the range −2,147,483,648 to 2,147,483,-647. When interpreted as an unsigned integer, significance increases from bit 0 to bit 31. The value of the unsigned integer is in the range 0 through 4,294,967,295.

macro A statement that requests a language processor to generate a predefined set of instructions.

mailbox A software data structure that is treated as a record-oriented device for general interprocess communication. Communication using a mailbox is similar to other forms of device-independent I/O. Senders perform a write to a mailbox, the receiver performs a read from that mailbox. Some system-wide mailboxes are defined: the error logger and OPCOM read from system-wide mailboxes.

main memory See physical memory.

mapping window A subset of the retrieval information for a file that is used to translate virtual block numbers to logical block numbers.

mass storage device A device capable of reading and writing data on mass storage media such as a disk pack or a magnetic tape reel.

member number The second number in a user identification code that uniquely identifies that code.

memory management The system functions that include the hardware's page mapping and protection and the operating system's image activator and pager.

Memory Mapping Enable (MME) A bit in a processor register that governs address translation.

modify access type The specified operand of an instruction or procedure is read, and is potentially modified and written, during that instruction's or procedure's execution.

module 1. A portion of a program or program library, as in a *source module*, *object module*, or *im-*

age module. 2. A board, usually made of plastic covered with an electrical conductor, on which logic devices (such as transistors, resistors, and memory chips) are mounted, and circuits connecting these devices are etched, as in a *logic module.*

Monitor Console Routine (MCR) The command interpreter in an RSX-11 system.

mount a volume 1. To logically associate a volume with the physical unit on which it is loaded (an activity accomplished by system software at the request of an operator). 2. To load or place a magnetic tape or disk pack on a drive and place the drive online (an activity accomplished by a system operator).

move mode Technique used for a record transfer in which the data records are copied between the I/O buffer and your program buffer for calculations or operations on the record. See locate mode.

mutex A semaphore that is used to control exclusive access to a region of code that can share a data structure or other resource. The mutex (mutual exclusion) semaphore ensures that only one process at a time has access to the region of code.

name block (NAM) An RMS user data structure that contains supplementary information used in parsing file specifications.

native image An image whose instructions are executed in native mode.

native mode The processor's primary execution mode in which the programmed instructions are interpreted as byte-aligned, variable-length instructions that operate on byte, word, longword, quadword, and octaword integer, F_, D_, G_ and H_floating format, character string, packed decimal, and variable-length bit field data. The instruction execution mode other than compatibility mode.

network A collection of interconnected individual computer systems.

nibble The low-order or high-order four bits of a byte.

node An individual computer system in a network.

null process A small system process that is the lowest priority process in the system and takes one entire priority class. One function of the null process is to accumulate idle processor time.

numeric string A contiguous sequence of bytes representing up to 31 decimal digits (one per byte) and possibly a sign. The numeric string is specified by its lowest addressed location, its length, and its sign representation.

object module The binary output of a language processor such as the assembler or a compiler, which is used as input to the linker.

object time system (OTS) See Run Time Procedure Library.

octaword Sixteen contiguous bytes (128 bits) starting on an addressable byte boundary. Bits are numbered from right to left, 0 to 127. An octaword is identified by the address of the byte containing the low-order bit (bit 0).

offset A fixed displacement from the beginning of a data structure. System offsets for items within a data structure normally have an associated symbolic name used instead of the numeric displacement. Where symbols are defined, programmers always reference the symbolic names for items in a data structure instead of using the numeric displacement.

opcode The pattern of bits within an instruction that specify the operation to be performed.

operand specifier The pattern of bits in an instruction that indicate the addressing mode, a register and/or displacement, which, taken together, identify an instruction operand.

operand specifier type The access type and data type of an instruction's operand(s). For example, the test instructions are of read access type, since they only read the value of the operand. The operand can be of byte, word, or longword data type, depending on whether the opcode is for the TSTB (test byte), TSTW (test word), or TSTL (test longword) instruction.

Operator Communication Manager (OPCOM) A system process that is always active. OPCOM receives input from a process that wants to inform an operator of a particular status or condition, passes a message to the operator, and tracks the message.

operator's console Any terminal identified as a terminal attended by a system operator.

owner In the context "system, owner, group, world," an owner is the particular member (of a group) to which a file, global section, mailbox, or event flag cluster belongs.

owner process The process (with the exception of the job controller) or subprocess that created a subprocess.

packed decimal A method of representing a decimal number by storing a pair of decimal digits in one byte, taking advantage of the fact that only four bits are required to represent the numbers 0 through 9.

packed decimal string A contiguous sequence of up to 16 bytes interpreted as a string of nibbles. Each nibble represents a digit except the low-order nibble of the highest addressed byte, which repre-

sents the sign. The packed decimal string is specified by its lowest addressed location and the number of digits.

page 1. A set of 512 contiguous byte locations used as the unit of memory mapping and protection. 2. The data between the beginning of file and a page marker, between two markers, or between a marker and the end of a file.

page fault An exception generated by a reference to a page which is not mapped into a working set.

page fault cluster size The number of pages read in on a page fault.

page frame number (PFN) The address of the first byte of a page in physical memory. The high-order 21 bits of the physical address of the base of a page.

page marker A character or characters (generally a form feed) that separates pages in a file that is processed by a text editor.

pager A set of kernel mode procedures that executes as the result of a page fault. The pager makes the page for which the fault occurred available in physical memory so that the image can continue execution. The pager and the image activator provide the operating system's memory management functions.

page table entry (PTE) The data structure that identifies the location and status of a page of virtual address space. When a virtual page is in memory, the PTE contains the page frame number needed to map the virtual page to a physical page. When it is not in memory, the page table entry contains the information needed to locate the page on secondary storage (disk).

paging The action of bringing pages of an executing process into physical memory when referenced. When a process executes, all of its pages are said to reside in virtual memory. Only the actively used pages, however, need to reside in physical memory. The remaining pages can reside on disk until they are needed in physical memory. In this system, a process is paged only when it references more pages than it is allowed to have in its working set. When the process refers to a page not in its working set, a page fault occurs. This causes the operating system's pager to read in the referenced page if it is on disk (and, optionally, other related pages depending on a cluster factor), replacing the least recently faulted pages as needed. A process pages only against itself.

parameter See command parameter.

per-process address space See process address space.

physical address The address used by hardware to identify a location in physical memory or on directly addressable secondary storage devices such as a disk. A physical memory address consists of a page frame number and the number of a byte within the page. A physical disk block address consists of a cylinder or track and sector number.

physical address space The set of all possible 30-bit physical addresses that can be used to refer to locations in memory (memory space) or device registers (I/O space).

physical block A block on a mass storage device referred to by its physical (device-oriented) address rather than a logical (volume-relative) or virtual (file-relative) address.

physical I/O functions A set of I/O functions that allow access to all device level I/O operations except maintenance mode.

physical memory The memory modules connected to the SBI that are used to store: 1) instructions that the processor can directly fetch and execute, and 2) any other data that a processor is instructed to manipulate. Also called main memory.

position-dependent code Code that can execute properly only in the locations in virtual address space that are assigned to it by the linker.

position-independent code Code that can execute properly without modification wherever it is located in virtual address space, even if its location is changed after it has been linked. Generally, this code uses addressing modes that form an effective address relative to the PC.

primary key The mandatory key within the data records of an indexed file; used by VAX-11 RMS to determine the placement of records within the file and to build the primary index. See key (indexed files) and alternate key.

primary vector A location that contains the starting address of a condition handler to be executed when an exception condition occurs. If a primary vector is declared, that condition handler is the first handler to be executed.

private section An image section of a process that is not shareable among processes. See also global section.

privilege See process privilege, user privilege, and image privilege.

privileged instructions In general, any instructions intended for use by the operating system or privileged system programs. In particular, instructions that the processor will not execute unless the

current access mode is kernel mode (e.g., HALT, SVPCTX, LDPCTX, MTPR, and MFPR).

procedure 1. A routine entered via a Call instruction. 2. See command procedure.

process The basic entity scheduled by the system software that provides the context in which an image executes. A process consists of an address space and both hardware and software context.

process address space See process space.

process context The hardware and software contexts of a process.

process control block (PCB) A data structure used to contain process context. The hardware PCB contains the hardware context. The software PCB contains the software context, which includes a pointer to the hardware PCB.

process header A data structure that contains the hardware PCB, accounting and quota information, process section table, working set list, and the page tables defining the virtual layout of the process.

process header slots That portion of the system address space in which the system stores the process headers for the processes in the balance set. The number of process header slots in the system determines the number of processes that can be in the balance set at any one time.

process identification (PID) The operating system's unique 32-bit binary value assigned to a process.

process I/O segment That portion of a process control region that contains the process permanent RMS internal file access block for each open file, and the I/O buffers, including the command interpreter's command buffer and command descriptors.

process name A 1- to 15-character ASCII string that can be used to identify processes executing under the same group number.

processor register A part of the processor used by the operating system software to control the execution states of the computer system. They include the system base and length registers, the program and control region base and length registers, the system control block base register, the software interrupt request register, and many more.

Processor Status Longword (PSL) A system programmed processor register consisting of a word of privileged processor status and the PSW. The privileged processor status information includes: the current IPL (interrupt priority level), the previous access mode, the current access mode, the interrupt stack bit, the trace fault pending bit, and the compatibility mode bit.

Processor Status Word (PSW) The low-order word of the Processor Status Longword. Processor status information includes: the condition codes (carry, overflow, zero, negative), the arithmetic trap enable bits (integer overflow, decimal overflow, floating underflow), and the trace enable bit.

process page tables The page tables used to describe process virtual memory.

process priority The priority assigned to a process for scheduling purposes. The operating system recognizes 32 levels of process priority, where 0 is low and 31 high. Levels 16 through 31 are used for time-critical processes. The system does not modify the priority of a time-critical process (although the system manager or process itself may). Levels 0 through 15 are used for normal processes. The system may temporarily increase the priority of a normal process based on the activity of the process.

process privileges The privileges granted to a process by the system, which are a combination of user privileges and image privileges. They include, for example, the privilege to: affect other processes associated with the same group as the user's group, affect any process in the system regardless of UIC, set process swap mode, create permanent event flag clusters, create another process, create a mailbox, and perform direct I/O to a file-structured device.

process section See private section.

process space The lowest-addressed half of virtual address space, where per-process instructions and data reside. Process space is divided into a program region and a control region.

Program Counter (PC) General register 15 (R15). At the beginning of an instruction's execution, the PC normally contains the address of a location in memory from which the processor will fetch the next instruction it will execute.

program locality A characteristic of a program that indicates how close or far apart the references to locations in virtual memory are over time. A program with a high degree of locality does not refer to many widely scattered virtual addresses in a short period of time.

programmer number See member number.

program region The lowest-addressed half of process address space (P0 space). The program region contains the image currently being executed by the process and other user code called by the image.

Program region Base Register (P0BR) The processor register, or its equivalent in a hardware process control block, that contains the base virtual address of the page table entry for virtual page number 0 in a process program region.

Program region Length Register (P0LR) The processor register, or its equivalent in a hardware process control block, that contains the number of entries in the page table for a process program region.

program section (psect) A portion of a program with a given protection and set of storage management attributes. Program sections that have the same attributes are gathered together by the linker to form an image section.

project number See group number or account number.

pure code See re-entrant code.

quadword Eight contiguous bytes (64 bits) starting on an addressable byte boundary. Bits are numbered from right to left, 0 to 63. A quadword is identified by the address of the byte containing the low-order bit (bit 0). When interpreted arithmetically, a quadword is a 2's complement integer with significance increasing from bit 0 to bit 62. Bit 63 is used as the sign bit. The value of the integer is in the range -2^{63} to $2^{63} - 1$.

qualifier A portion of a command string that modifies a command verb or command parameter by selecting one of several options. A qualifier, if present, follows the command verb or parameter to which it applies and is in the format: "/qualifier:option." For example, in the command string "PRINT filename/COPIES:3," the COPIES qualifier indicates that the user wants three copies of a given file printed.

queue 1. n. A circular, doubly-linked list. See system queues. v. To make an entry in a list or table, perhaps using the INSQUE instruction. 2. See job queue.

queue priority The priority assigned to a job placed in a spooler queue or a batch queue.

quota The total amount of a system resource, such as CPU time, that a job is allowed to use in an accounting period, as specified by the system manager in the user authorization file. See also limit.

random access by key Indexed files only: Retrieval of a data record in an indexed file by either a primary or alternate key within the data record. See key (indexed files).

random access by record's file address The retrieval of a record by its unique address, which is provided to the program by RMS. This method of access is the only means of randomly accessing a sequentially organized file containing variable length records.

random access by relative record number Retrieval of a record by its relative record number. See relative record number. For relative files, random access by relative record number is synonymous with random access by key. See random access by key (relative files only).

read access type An instruction or procedure operand attribute indicating that the specified operand is only read during instruction or procedure execution.

record A set of related data that your program treats as a unit.

record access block (RAB) An RMS user data structure that represents a request for a record access stream. A RAB relates to operations on the records within a file, such as UPDATE, DELETE, or GET.

record access mode The method used in RMS for retrieving and storing records in a file. One of three methods: sequential, random, and record's file address.

record blocking The technique of grouping multiple records into a single block. On magnetic tape, an IRG is placed after the block rather than after each record. This technique reduces the number of I/O transfers required to read or write the data; and, in addition (for magnetic tape), increases the amount of usable storage area. Record blocking also applies to disk files.

record cell A fixed-length area in a relative file that can contain a record. The concept of fixed-length record cells lets VAX-11 RMS directly calculate the record's actual position in the file.

record format The way a record physically appears on the recording surface of the storage medium. The record format defines the method for determining record length.

record length The size of a record; that is, the number of bytes in a record.

record locking A facility that prevents access to a record by more than one record stream or process until the initiating record stream or process releases the record.

Record Management Services A set of operating system procedures that are called by programs to process files and records within files. RMS allows programs to issue READ and WRITE requests at the record level (record I/O) as well as read and write blocks (block I/O). RMS is an integral part of the system software. RMS procedures run in executive mode.

record-oriented device A device such as a terminal, line printer, or card reader, on which the largest

unit of data a program can access in one I/O operation is the device's physical record.

record's file address The unique address of a record in a file, which is returned by RMS whenever a record is accessed, that allows records in disk files to be access randomly regardless of file organization. This address is valid only for the life of the file. If an indexed file is reorganized, then the RFA of each record will typically change.

re-entrant code Code that is never modified during execution. It is possible to let many users share the same copy of a procedure or program written as re-entrant code.

register A storage location in hardware logic other than main memory. See also general register, processor register, and device register.

register deferred indexed mode An indexed addressing mode in which the base operand specifier uses register deferred mode addressing.

register deferred mode In register deferred mode addressing, the contents of the specified register are used as the address of the actual instruction operand.

register mode In register mode addressing, the contents of the specified register are used as the actual instruction operand.

relative file organization The arrangement of records in a file where each record occupies a cell of equal length within a bucket. Each cell is assigned a successive number, called a relative record number, which represents the cell's position relative to the beginning of the file.

relative record number An identification number used to specify the position of a record cell relative to the beginning of the file; used as the key during random access by key mode to relative files.

resource A physical part of the computer system such as a device or memory, or an interlocked data structure such as a mutex. Quotas and limits control the use of physical resources.

resource wait mode An execution state in which a process indicates that it will wait until a system resource becomes available when it issues a service request requiring a resource. If a process wants notification when a resource is not available, it can disable resource wait mode during program execution.

return status code See status code.

RMS-11 A set of routines which is linked with compatibility mode programs, and provides similar functional capabilities to VAX-11 RMS. The file organizations and record formats used by RMS-11 are identical to those of VAX-11 RMS.

Run Time Procedure Library The collection of procedures available to native mode images at run time. These library procedures (such as trigonometric functions, etc.) are common to all native mode images, regardless of the language processor used to compile or assemble the program.

scatter/gather The ability to transfer in one I/O operation data from discontiguous pages in memory to contiguous blocks on disk, or data from contiguous blocks on disk to discontiguous pages in memory.

secondary storage Random access mass storage.

secondary vector A location that identifies the starting address of a condition handler to be executed when a condition occurs and the primary vector contains zero or the handler to which the primary vector points chooses not to handle the condition.

section A portion of process virtual memory that has common memory management attributes (protection, access, cluster factor, etc.). It is created from an image section, a disk file, or as the result of a Create Virtual Address Space system service. See global section, private section, image section, and program section.

self-relative queue A circularly linked list whose forward and backward links use the address of the entry in which they occur as the base address for the link displacement to the linked entry. Contrast with absolute addresses used to link a queue.

sequential file organization A file organization in which records appear in the order in which they were originally written. The records can be fixed length or variable length.

sequential record access mode Record storage or retrieval which starts at a designated point in the file and continues in one-after-the-other fashion through the file. That is, records are accessed in the order in which they physically appear in the file.

shareable image An image that has all of its internal references resolved, but which must be linked with an object module(s) to produce an executable image. A sharable image cannot be executed. A shareable image file can be used to contain a library of routines. A shareable image can be used to create a global section by the system manager.

shell process A predefined process that the job initiator copies to create the minimum context necessary to establish a process.

signal 1. An electrical impulse conveying information. 2. The software mechanism used to indicate that an exception condition was detected.

slave terminal A terminal from which it is not possible to issue commands to the command interpreter. A terminal assigned to application software.

small process A system process that has no control region in its virtual address space and has an abbreviated context. Examples are the working set swapper and the null process. A small process is scheduled in the same manner as user processes, but must remain resident during its execution.

software context The context maintained by the operating system that describes a process. See software process control block (PCB).

software interrupt An interrupt generated on interrupt priority level 1 through 15, which can be requested only by software.

software process control block (PCB) The data structure used to contain a process' software context. The operating system defines a software PCB for every process when the process is created. The software PCB includes the following kinds of information about the process: current state; storage address if it is swapped out of memory; unique identification of the process, and address of the process header (which contains the hardware PCB). The software PCB resides in system region virtual address space. It is not swapped with a process.

software priority See process priority and queue priority.

spooling output spooling: The method by which output to a low-speed peripheral device (such as a line printer) is placed into queues maintained on a high-speed device (such as disk) to await transmission to the low-speed device. Input spooling: the method by which input from a low-speed peripheral (such as the card reader) is placed into queues maintained on a high-speed device (such as disk) to await transmission to a job processing that input.

spool queue The list of files supplied by processes that are to be processed by a symbiont. For example, a line printer queue is a list of files to be printed on the line printer.

stack An area of memory set aside for temporary storage, or for procedure and interrupt service linkages. A stack uses the last-in, first-out concept. As items are added to ("pushed on") the stack, the stack pointer decrements. As items are retrieved from ("popped off") the stack, the stack pointer increments.

stack frame A standard data structure built on the stack during a procedure call, starting from the location addressed by the FP to lower addresses, and popped off during a return from procedure. Also called call frame.

stack pointer General register 14 (R14). SP contains the address of the top (lowest address) of the processor-defined stack. Reference to SP will access one of the five possible stack pointers (kernel, executive, supervisor, user, or interrupt) depending on the value in the current mode and interrupt stack bits in the Processor Status Longword (PSL).

state queue A list of processes in a particular processing state. The scheduler uses state queues to keep track of processes' eligibility to execute. They include: processes waiting for a common event flag, suspended processes, and executable processes.

status code A longword value that indicates the success or failure of a specific function. For example, system services always return a status code in R0 upon completion.

store through See write through.

strong definition Definition of a global symbol that is explicitly available for reference by modules linked with the module in which the definition occurs. The linker always lists a global symbol with a strong definition in the symbol portion of the map. The librarian always includes a global symbol with a strong definition in the global symbol table of a library.

strong reference A reference to a global symbol in an object module that requests the linker to report an error if it does not find a definition for the symbol during linking. If a library contains the definition, the linker incorporates the library module defining the global symbol into the image containing the strong reference.

subprocess A subsidiary process created by another process. The process that creates a subprocess is its owner. A subprocess receives resource quotas and limits from its owner. When an owner process is removed from the system, all its subprocesses (and their subprocesses) are also removed.

supervisor mode The third most privileged processor access mode (mode 2). The operating system's command interpreter runs in supervisor mode.

suspension A state in which a process is inactive, but known to the system. A suspended process becomes active again only when another process requests the operating system to resume it. Contrast with hibernation.

swap mode A process execution state that determines the eligibility of a process to be swapped out of the balance set. If process swap mode is disabled, the process working set is locked in the balance set.

swapping The method for sharing memory resources among several processes by writing an entire working set to secondary storage (swap out) and reading another working set into memory (swap in). For example, a process' working set can be written to secondary storage while the process is waiting for I/O completion on a slow device. It is brought back into the balance set when I/O completes. Contrast with paging.

switch See (command) qualifier.

symbiont A full process that transfers record-oriented data to or from a mass storage device. For example, an input symbiont transfers data from card readers to disks. An output symbiont transfers data from disks to line printers.

symbiont manager The function (in the system process called the job controller) that maintains spool queues, and dynamically creates symbiont processes to perform the necessary I/O operations.

symbol See local symbol, global symbol, and universal global symbol.

Synchronous Backplane Interconnect (SBI) The part of the hardware that interconnects the processor, memory controllers, MASSBUS adapters, and the UNIBUS adapter.

synchronous record operation A mode of record processing in which a user program issues a record read or write request and then waits until that request is fulfilled before continuing to execute.

system In the context "system, owner, group, world," the system refers to the group numbers that are used by operating system and its controlling users, the system operators and system manager.

system address space See system space and system region.

System Base Register (SBR) A processor register containing the physical address of the base of the system page table.

system buffered I/O An I/O operation, such as terminal or mailbox I/O, in which an intermediate buffer from the system buffer pool is used instead of a process-specified buffer. Contrast with direct I/O.

System Control Block (SCB) The data structure in system space that contains all the interrupt and exception vectors known to the system.

System Control Block Base register (SCBB) A processor register containing the base address of the system control block.

system device The random access mass storage device unit on which the volume containing the operating system software resides.

system dynamic memory Memory reserved for the operating system to allocate as needed for temporary storage. For example, when an image issues an I/O request, system dynamic memory is used to contain the I/O request packet. Each process has a limit on the amount of system dynamic memory that can be allocated for its use at one time.

System Identification Register A processor register which contains the processor type and serial number.

system image The image that is read into memory from secondary storage when the system is started up.

System Length Register (SLR) A processor register containing the length of the system page table in longwords, that is, the number of page table entries in the system region page table.

System Page Table (SPT) The data structure that maps the system region virtual addresses, including the addresses used to refer to the process page tables. The System Page Table (SPT) contains one Page Table Entry (PTE) for each page of system region virtual memory. The physical base address of the SPT is contained in a register called the SBR.

system process A process that provides system-level functions. Any process that is part of the operating system. See also small process, fork process.

system programmer A person who designs and/or writes operating systems, or who designs and writes procedures or programs that provide general purpose services for an application system.

system queue A queue used and maintained by operating system procedures. See also state queues.

system region The third quarter of virtual address space. The lowest-addressed half of system space. Virtual addresses in the system region are shareable between processes. Some of the data structures mapped by system region virtual addresses are: system entry vectors, the System Control Block (SCB), the System Page Table (SPT), and process page tables.

system services Procedures provided by the operating system that can be called by user processes.

system space The highest-addressed half of virtual address space. See also system region.

system virtual address A virtual address identifying a location mapped by an address in system space.

system virtual space See system space.

task An RSX-11/IAS term for a process and image bound together.

terminal The general name for those peripheral devices that have keyboards and video screens or printers. Under program control, a terminal enables people to type commands and data on the keyboard and receive messages on the video screen or printer. Examples of terminals are the LA36 DECwriter hard-copy terminal and VT100 video display terminal.

time-critical process A process assigned to a software priority level between 16 and 31, inclusive. The scheduling priority assigned to a time-critical process is never modified by the scheduler, although it can be modified by the system manager or process itself.

timer A system fork process that maintains the time of day and the date. It also scans for device timeouts and performs time-dependent scheduling upon request.

track A collection of blocks at a single radius on one recording surface of a disk.

transfer address The address of the location containing a program entry point (the first instruction to execute).

translation buffer An internal processor cache containing translations for recently used virtual addresses.

trap An exception condition that occurs at the end of the instruction that caused the exception. The PC saved on the stack is the address of the next instruction that would normally have been executed. All software can enable and disable some of the trap conditions with a single instruction.

trap enables Three bits in the Processor Status Word that control the processor's action on certain arithmetic exceptions.

two's complement A binary representation for integers in which a negative number is one greater than the bit complement of the positive number.

two-way associative cache A cache organization which has two groups of directly mapped blocks. Each group contains several blocks for each index position in the cache. A block of data from main memory can go into any group at its proper index position. A two-way associative cache is a compromise between the extremes of fully associative and direct mapping cache organizations that takes advantage of the features of both.

type ahead A terminal handling technique in which the user can enter commands and data while the software is processing a previously entered command. The commands typed ahead are not echoed on the terminal until the command processor is ready to process them. They are held in a type ahead buffer.

unit record device A device such as a card reader or line printer.

universal global symbol A global symbol in a shareable image that can be used by modules linked with that shareable image. Universal global symbols are typically a subset of all the global symbols in a shareable image. When creating a shareable image, the linker ensures that universal global symbols remain available for reference after symbols have been resolved.

unwind the call stack To remove call frames from the stack by tracing back through nested procedure calls using the current contents of the FP register and the FP register contents stored on the stack for each call frame.

urgent interrupt An interrupt received on interrupt priority levels 24 through 31. These can be generated only by the processor for the interval clock, serious errors, and power fail.

user authorization file A file containing an entry for every user that the system manager authorizes to gain access to the system. Each entry identifies the user name, password, default account, User Identification Code (UIC), quotas, limits, and privileges assigned to individuals who use the system.

user environment test package (UETP) A collection of routines that verify that the hardware and software systems are complete, properly installed, and ready to be used.

User File Directory (UFD) See directory.

User Identification Code (UIC) The pair of numbers assigned to users and to files, global sections, common event flag clusters, and mailboxes that specifies the type of access (read and/or write access, and in the case of files, execute and/or delete access) available to the owners, group, world, and system. It consists of a group number and a member number separated by a comma.

user mode The least privileged processor access mode (mode 3). User processes and the Run Time Library procedures run in user mode.

user name The name that a person types on a terminal to log on to the system.

user number See member number.

user privileges The privileges granted a user by the system manager. See process privileges.

utility A program that provides a set of related

general purpose functions, such as a program development utility (an editor, a linker, etc.), a file management utility (file copy or file format translation program), or operations management utility (disk backup/restore, diagnostic program, etc.).

value return registers The general registers R0 and R1 used by convention to return function values. These registers are not preserved by any called procedures. They are available as temporary registers to any called procedure. All other registers (R2, R3,...,R11, AP, FP, SP, PC) are preserved across procedure calls.

variable-length bit field A set of 0 to 32 contiguous bits located arbitrarily with respect to byte boundaries. A variable bit field is specified by four attributes: 1) the address A of a byte, 2) the bit position P of the starting location of the bit field with respect to bit 0 of the byte at address A, 3) the size, in bits, of the bit field, and 4) whether the field is signed or unsigned.

variable-length record format A file format in which records are not necessarily the same length.

variable with fixed-length control record format Property of a file in which records of variable-length contain an additional fixed control area capable of storing data that may have no bearing on the other contents of the record. Variable with fixed-length control record format is not applicable to indexed files.

VAX-11 Record Management Services (VAX-11 RMS) The file and record access subsystem of the VAX/VMS operating system for VAX. VAX-11 RMS helps your application program process records within files, thereby allowing interaction between your application program and its data.

vector 1. A interrupt or exception vector is a storage location known to the system that contains the starting address of a procedure to be executed when a given interrupt or exception occurs. The system defines separate vectors for each interrupting device controller and for classes of exceptions. Each system vector is a longword. 2. For exception handling, users can declare up to two software exception vectors (primary and secondary) for each of the four access modes. Each vector contains the address of a condition handler. 3. A one-dimensional array.

version number 1. The field following the file type in a file specification. It begins with a period (.) and is followed by a number which generally identifies it as the latest file created of all files having the identical file specification but for version number. 2. The number used to identify the revision level of program.

virtual address A 32-bit integer identifying a byte "location" in virtual address space. The memory management hardware translates a virtual address to a physical address. The term "virtual address" may also refer to the address used to identify a virtual block on a mass storage device.

virtual address space The set of all possible virtual addresses that an image executing in the context of a process can use to identify the location of an instruction or data. The virtual address space seen by the programmer is a linear array of 4,294,967,296 (2^{32}) byte addresses.

virtual block A block on a mass storage device referred to by its file-relative address rather than its logical (volume-oriented) or physical (device-oriented) address. The first block in a file is always virtual block 1.

virtual I/O functions A set of I/O functions that must be interpreted by an ancillary control process.

virtual memory The set of storage locations in physical memory and on disk that are referred to by virtual addresses. From the programmer's viewpoint, the secondary storage locations appear to be locations in physical memory. The size of virtual memory in any system depends on the amount of physical memory available and the amount of disk storage used for non-resident virtual memory.

virtual page number The virtual address of a page of virtual memory.

volume
Disks: An ordered set of 512-byte blocks. The basic medium that carries a Files-11 structure.

Magnetic tape: A reel of magnetic tape, which may contain a part of a file, a complete file, or more than one file.

volume set A collection of related volumes.

wait To become inactive. A process enters a process wait state when the process suspends itself, hibernates, or declares that it needs to wait for an event, resource, mutex, etc.

wake To activate a hibernating process. A hibernating process can be awakened by another process or by the timer process, if the hibernating process or another process scheduled a wake-up call.

weak definition Definition of a global symbol that is not explicitly available for reference by modules linked with the module in which the definition occurs. The librarian does not include a global symbol with a weak definition in the global symbol table of a library. Weak definitions are often used when creating libraries to identify those global symbols that are needed only if the module containing them is otherwise linked with a program.

weak reference A reference to a global symbol that requests the linker not to report an error or to search the default library's global symbol table to resolve the reference if the definition is not in the modules explicitly supplied to the linker. Weak references are often used when creating object modules to identify those global symbols that may not be needed at run time.

wild card A symbol, such as an asterisk, that is used in place of a file name, file type, directory name, or version number in a file specification to indicate "all" for the given field.

window See mapping window.

word Two contiguous bytes (16 bits) starting on an addressable byte boundary. Bits are numbered from the right, 0 through 15. A word is identified by the address of the byte containing bit 0. When interpreted arithmetically, a word is a 2's complement integer with significance increasing from bit 0 to bit 14. If interpreted as a signed integer, bit 15 is the sign bit. The value of the integer is in the range −32,768 to 32,767. When interpreted as an unsigned integer, significance increases from bit 0 through bit 15 and the value of the unsigned integer is in the range 0 through 65,535.

working set The set of pages in process space to which an executing process can refer without incurring a page fault. The working set must be resident in memory for the process to execute. The remaining pages of that process, if any, are either in memory and not in the process working set or they are on secondary storage.

working set swapper A system process that brings process working sets into the balance set and removes them from the balance set.

world In the context "system, owner, group, world," world refers to all users, including the system operators, the system manager, and users both in an owner's group and in any other group.

write access type The specified operand of an instruction or procedure is written only during that instruction's or procedure's execution.

write allocate A cache management technique in which cache is allocated on a write miss as well as on the usual read miss.

write back A cache management technique in which data from a write operation to cache are copied into main memory only when the data in cache must be overwritten. This results in temporary inconsistencies between cache and main memory. Contrast with write through.

write through A cache management technique in which data from a write operation are copied in both cache and main memory. Cache and main memory data are always consistent. Contrast with write back.

Index

digital

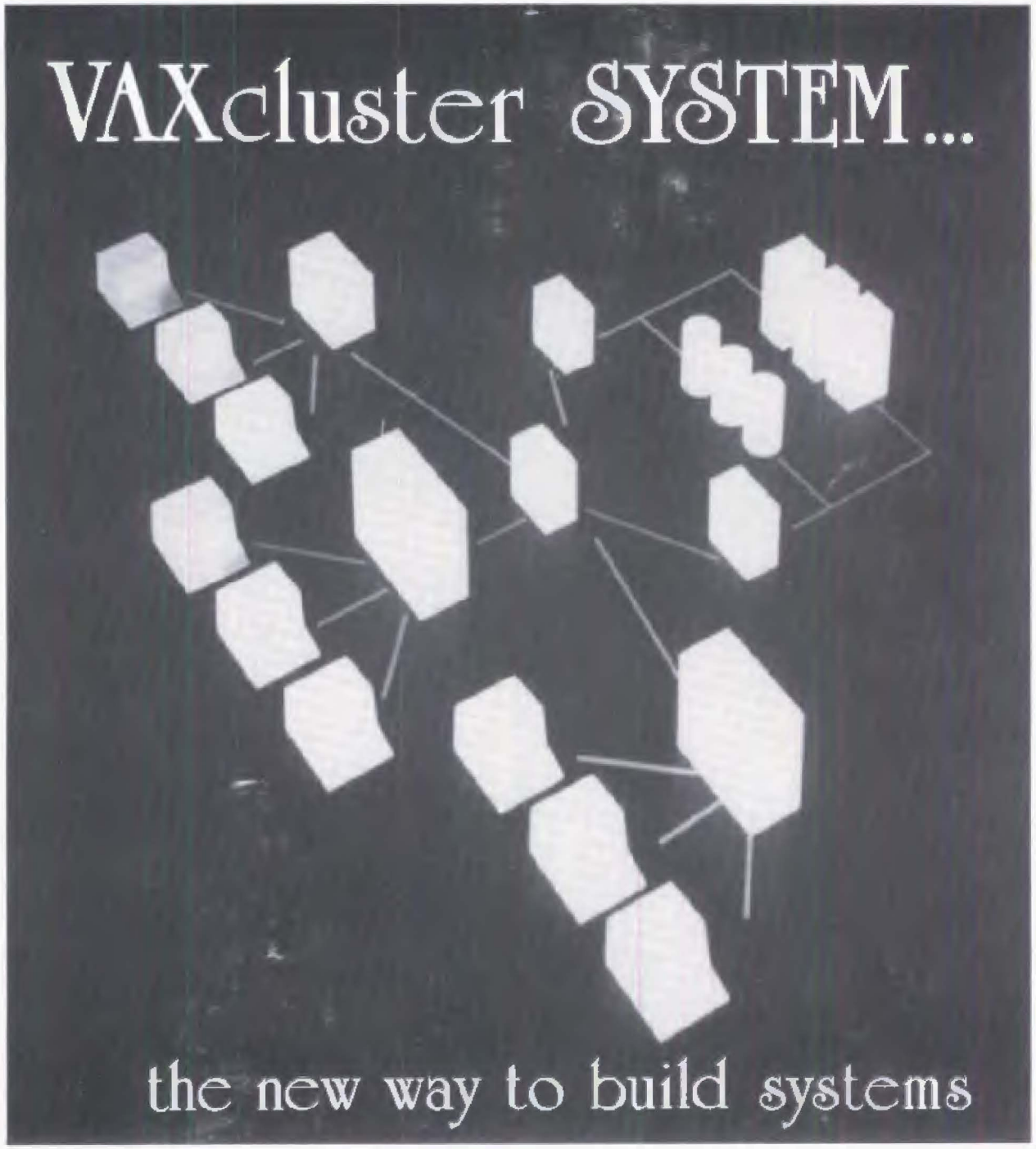
DIGITAL EQUIPMENT CORPORATION, Corporate Headquarters: Maynard, MA 01754, Tel. (617) 897-5111 — SALES AND SERVICE OFFICES; UNITED STATES — ALABAMA, Birmingham, Huntsville ARIZONA, Phoenix, Tucson ARKANSAS, Little Rock CALIFORNIA, Costa Mesa, El Segundo, Los Angeles, Oakland, Sacramento, San Diego, San Francisco, Monrovia, Santa Barbara, Santa Clara, Sherman Oaks COLORADO, Colorado Springs, Denver CONNECTICUT, Fairfield, Meriden DELAWARE, Newark FLORIDA, Miami, Orlando, Pensacola, Tampa GEORGIA, Atlanta HAWAII, Honolulu IDAHO, Boise ILLINOIS, Chicago, Peoria INDIANA, Indianapolis IOWA, Bettendorf KENTUCKY, Louisville LOUISIANA, New Orleans MARYLAND, Baltimore MASSACHUSETTS, Boston, Springfield, Waltham MICHIGAN, Detroit, Kalamazoo MINNESOTA, Minneapolis MISSOURI, Kansas City, St. Louis NEBRASKA, Omaha NEW HAMPSHIRE, Manchester NEW JERSEY, Cherry Hill, Parsippany, Princeton, Somerset NEW MEXICO, Albuquerque, Los Alamos NEW YORK, Albany, Buffalo, Long Island, New York City, Rochester, Syracuse, Westchester NORTH CAROLINA, Chapel Hill, Charlotte OHIO, Cincinnati, Cleveland, Columbus, Dayton OKLAHOMA, Tulsa OREGON, Portland PENNSYLVANIA, Harrisburg, Philadelphia, Pittsburgh RHODE ISLAND, Providence SOUTH CAROLINA, Columbia, Greenville TENNESSEE, Knoxville, Nashville TEXAS, Austin, Dallas, El Paso, Houston, San Antonio UTAH, Salt Lake City VERMONT, Burlington VIRGINIA, Fairfax, Richmond WASHINGTON, Seattle, Spokane WASHINGTON D.C. WEST VIRGINIA, Charleston WISCONSIN, Milwaukee INTERNATIONAL — EUROPEAN AREA HEADQUARTERS: Geneva, Tel: [41] (22)-93-33-11 INTERNATIONAL AREA HEADQUARTERS: Acton, MA 01754, U.S.A., Tel: (617) 263-6000 AUSTRALIA, Adelaide, Brisbane, Canberra, Hobart, Melbourne, Perth, Sydney, Townsville AUSTRIA, Vienna BELGIUM, Brussels BRAZIL, Rio de Janeiro, Sao Paulo CANADA, Calgary, Edmonton, Halifax, Kingston, London, Montreal, Ottawa, Quebec City, Regina, Toronto, Vancouver, Victoria, Winnipeg DENMARK, Copenhagen ENGLAND, Basingstoke, Birmingham, Bristol, Ealing, Epsom, Leeds, Leicester, London, Manchester, Reading, Welwyn FINLAND, Helsinki FRANCE, Bordeaux, Lyon, Paris, Puteaux, Strasbourg HOLLAND, Amstelveen, Delft, Utrecht HONG KONG IRELAND, Dublin ISRAEL, Tel Aviv ITALY, Milan, Rome, Turin JAPAN, Osaka, Tokyo MEXICO, Mexico City, Monterrey NEW ZEALAND, Auckland, Christchurch, Wellington NORTHERN IRELAND, Belfast NORWAY, Oslo, PUERTO RICO, San Juan SCOTLAND, Edinburgh REPUBLIC OF SINGAPORE SPAIN, Barcelona, Madrid SWEDEN, Gothenburg, Stockholm SWITZERLAND, Geneva, Zurich, WEST GERMANY, Berlin, Cologne, Frankfurt, Hamburg, Hannover, Munich, Nurnberg, Stuttgart

sales update

SPECIAL ISSUE

APRIL 25, 1983

VAXcluster SYSTEM...



the new way to build systems

digital

FOR INTERNAL USE ONLY

trademarks

The following are trademarks of DIGITAL EQUIPMENT CORPORATION:

DEC, DECmate, DECnet, DECsystem-10, DECSYSTEM-20, DECUS, DECwriter, DIBOL, the DIGITAL Logo, MASSBUS, PDP, P/OS, Professional, Rainbow, RSTS, RSX, UNIBUS, VAX, VMS, VT, Work Processor.

TABLE OF CONTENTS

Preface

With the VAXcluster Program announcement, Digital introduces the most flexible, cost-effective way to buy and grow computer systems. Customers can now start with the industry's most interactive, efficient, general-purpose system-- VAX, and continue to tailor and grow this basic system to respond to the application needs and continual growth of the user community. The modularity and continuity of the VAX/VMS product offering will provide the ability to manage the system without loss of investment in existing programs or need for re-programming, re-training, or re-planning.

This new way to grow systems will be introduced in a phased fashion, beginning with this announcement. The announcement includes specific hardware and VMS support, as well as the future direction of the VAXcluster program. The next phase of the Program will be announced in six to nine months.

Announcement of the VAXcluster is planned for May 2, 1983. Do not disclose the information contained in this Sales Update special issue until the announcement date.

VAXcluster Program "awareness" training is scheduled for the Q4/FY83 Sales Training Success Train. It will be a 1 1/2 hour session immediately following the Distributed Systems product training. Be sure to register for the training! Also, "Self Pace Instruction" material will be available later in Q4/FY83. Look for more information in the VAXcluster Announcement Sales Flash.

INTRODUCTION

A SPECIAL SALES UPDATE FOR VAXcluster SYSTEMS

Jack Shields
Vice-President,
Group Manager
DTN: 276-9890
OGO1-2/R12

This Sales Update issue is devoted to a new and exciting extension to our VAX products. Today's Sales Update announces VAXcluster systems, a new concept in VAX computing and a system product which is unique in our industry.

VAXcluster systems extend the winning characteristics of VAX to high capacity and high availability applications. They improve our competitive position at the high end of the VAX range. With this new concept in VAX computing, we can sell and win new customers and new applications.

VAXclusters give you the opportunity to sell to a different level of management. VAXclusters have features and benefits for managers responsible for multiple departments, multiple applications, and multiple functions. While Digital and VAX have been successful in selling to the individual departments, VAXcluster systems now give you an entry into the next higher management level.

In addition, VAXcluster systems provide a new and valuable growth alternative for our current VAX customers.

Read about VAXcluster systems in this special Sales Update issue.

And Good Selling!

ANNOUNCING VAXcluster SYSTEMS

Dave Chanoux
DTN: 247-2580
TWO/B02 RCS:TWSK

VAXcluster Systems: The New Way to Grow Systems.

Today's announcement of VAXcluster systems is a major program and product announcement for Digital. We are:

- Announcing hardware products which define a new system topology for VAX
- Announcing software support in VMS which will allow customers to build complete systems NOW (VMS V3.3 and V3.4)
- Describing the direction of our VMS software development to provide added functionality for VAXcluster systems (Product announcements for these cabapilities will occur in the future)

This announcement will give you and your customers the information needed to sell and install VAXcluster systems today and to plan their use in the future.

VAXcluster systems give you a new vehicle for achieving sales and business goals. With VAXcluster systems, Digital has a stronger competitive position at the high end of the VAX family range.

VAXcluster systems are a new concept in VAX computing. They add high system capacity and system availability features to the VAX Architecture. A VAXcluster system is a combination of VAX/VMS systems; 750s, 780s, and/or 782s; and HSC50 Hierarchical Storage Controllers interconnected with Digital's Computer Interconnect (CI).

VAXcluster systems provide a set of system products with unique features and customer benefits. These extensions to VAX have been designed with two objectives: First, to take VAX to new applications and customers in new markets; and second, to provide a new growth alternative for existing VAX customers in traditional markets.

VAXcluster SYSTEMS EXTEND VAX RANGE TO NEW CUSTOMERS AND NEW APPLICATIONS.

VAX has many strong product characteristics. VAXcluster systems preserve the characteristics which have made VAX a leader in our industry and extend VAX features to span a new range.

VAXcluster systems extend the range to high capacity systems. How is system capacity measured? Many factors, including number of users, processor performance, disk storage capacity, response to terminal users, functions performed, all contribute to a user's perception of system capacity.

- VAXcluster systems deliver more processing capacity.

  Multiple VAX processors achieve performance levels previously unattainable with VAX systems. Effective interconnection allows VAXcluster systems to compete where high compute performance is required. VAX processors are

combined in a cooperative manner to achieve higher levels of system process capacity. Customers can apply the VAX compute resource in many ways to achieve higher performance: devote a processor to an application, to a user group or to a department. Customers can select a processor to meet a designated need: 750, 780, and 782 all compute in clusters. Customers can tune each processor independently to optimize performance.

- VAXcluster systems deliver more disk storage capacity.

  Large disk-capacity VAXcluster systems coupled with industry-leading disk technology, very competitive disk products, and excellent information management software products allow VAX to compete in large database applications. Customers can attach disk storage units to VAX processors via UNIBUS and MASSBUS controllers or can configure disks through the HSC50. In either case, disks can be dedicated to a VAX processor, a department, a user group, an application, or can be shared across the VAXcluster system. The result is more disk storage capacity. Disk storage will soon be measured in gigabytes. VAXcluster systems and the Digital Storage Architecture deliver in this range.

With these characteristics, VAXcluster systems improve our competitive position at the high end of the VAX performance range. They will take VAX to large system, and large information management applications.

VAXcluster systems extend the VAX range to higher system availability, allowing us to better compete in this market segment. The VAXcluster system thus, has redundant features which customers can apply to achieve higher levels of system availability. These features are implemented in hardware such as the CI780, CI750, and HSC50; and in software as VMS components.

VAXcluster systems are unique. They allow customers to implement systems to achieving higher performance or higher availability or both. The customer's application will dictate the degree of each. This unique capability now takes VAX to new applications and to new customers.

VAX will now compete against mainframe suppliers, companies with special purpose point products and other emerging companies who are competing in this market segment. The competitive section of this Sales Update special issue details some product comparisons which will help you understand our product advantages in this market segment.

VAXcluster SYSTEMS: A NEW GROWTH ALTERNATIVE FOR THE VAX ACCOUNT.

For today's VAX customer, VAXcluster systems offer a new growth alternative. To achieve more processing capacity, a customer can add a processor to a cluster. Either a 750, 780, or 782 can be selected to meet the application need. To achieve more disk storage capacity, a customer can add more disk units and distribute those disk units on processors, or on HSC50 mass-storage controllers to achieve capacities never before attainable on VAX.

Every installed VAX-11/750, 780, or 782 is a VAXcluster system prospect. Every one of these systems can be upgraded to a VAXcluster system. Don't overlook the installed base as a source of bookings.

In summary, VAXcluster systems allow you to sell where you could not sell before. High capacity and high availability applications are prospects for VAXcluster systems. Present VAXcluster systems to your existing VAX accounts.

Show them how the unique and easy growth of VAXcluster systems can continue to deliver capacity as they may need it while preserving their investment in VAX products.

VAXcluster systems: The new way to grow systems.

OVERVIEW

V A X B A S E P R O D U C T M A R K E T I N G

VAXcluster SYSTEMS: PROGRAM SUMMARY

Terry Retford
DTN: 247-2930
TWO/B02 RCS:TWSK

VAXcluster systems represent a new concept in VAX computing. The concept and implementation offer users new ways to grow their systems in many dimensions including: capacity, systems availability, and data integrity.

Before introducing the product, it is important to understand some of the goals and design tradeoffs in adopting the VAXcluster System approach. There are two key system goals:

1. Adopt a computing approach that will grow and extend easily over time, allowing our customers to utilize a variety of technologies and to incorporate new technologies as they become available.

2. Offer customers high, overall systems availability.

To satisfy these goals we have adopted a loosely coupled multiprocessor systems approach. Loosely coupled multiprocessor systems are characterized by multiple processors, each with a copy of the software operating system and separated by a message oriented interprocessor systems protocol.

An alternate approach is tightly coupled multiprocessing, with processors in close proximity, connected through shared high-bandwidth memory, and with a single copy of the software operating system. An example of this approach is the VAX-11/782.

The loosely coupled approach allows the incorporation of processors of differing types into the system. They can be of different performance and different states of technology, thus as we develop new processors incorporating new technology we can easily integrate them into VAXcluster systems.

In addition to processors, mass-storage servers are separate elements within the system connected to processors through dual paths. This provides a system with no single point of failure in the component interconnection and therefore, higher overall systems availability. The use of separate processors with independent operating systems increases the survivability of the system in the presence of operating system failures.

V A X B A S E P R O D U C T M A R K E T I N G

ANNOUNCING VAXcluster SYSTEMS

Dave Chanoux
DTN: 247-2580
TWO/B02 RCS:TWSK

VAXcluster Systems: The New Way to Grow Systems.

Today's announcement of VAXcluster systems is a major program and product announcement for Digital. We are:

- Announcing hardware products which define a new system topology for VAX
- Announcing software support in VMS which will allow customers to build complete systems NOW (VMS V3.3 and V3.4)
- Describing the direction of our VMS software development to provide added functionality for VAXcluster systems (Product announcements for these cababilities will occur in the future)

This announcement will give you and your customers the information needed to sell and install VAXcluster systems today and to plan their use in the future.

VAXcluster systems give you a new vehicle for achieving sales and business goals. With VAXcluster systems, Digital has a stronger competitive position at the high end of the VAX family range.

VAXcluster systems are a new concept in VAX computing. They add high system capacity and system availability features to the VAX Architecture. A VAXcluster system is a combination of VAX/VMS systems; 750s, 780s, and/or 782s; and HSC50 Hierarchical Storage Controllers interconnected with Digital's Computer Interconnect (CI).

VAXcluster systems provide a set of system products with unique features and customer benefits. These extensions to VAX have been designed with two objectives: First, to take VAX to new applications and customers in new markets; and second, to provide a new growth alternative for existing VAX customers in traditional markets.

VAXcluster SYSTEMS EXTEND VAX RANGE TO NEW CUSTOMERS AND NEW APPLICATIONS.

VAX has many strong product characteristics. VAXcluster systems preserve the characteristics which have made VAX a leader in our industry and extend VAX features to span a new range.

VAXcluster systems extend the range to high capacity systems. How is system capacity measured? Many factors, including number of users, processor performance, disk storage capacity, response to terminal users, functions performed, all contribute to a user's perception of system capacity.

- VAXcluster systems deliver more processing capacity.

  Multiple VAX processors achieve performance levels previously unattainable with VAX systems. Effective interconnection allows VAXcluster systems to compete where high compute performance is required. VAX processors are

combined in a cooperative manner to achieve higher levels of system process capacity. Customers can apply the VAX compute resource in many ways to achieve higher performance: devote a processor to an application, to a user group or to a department. Customers can select a processor to meet a designated need: 750, 780, and 782 all compute in clusters. Customers can tune each processor independently to optimize performance.

- VAXcluster systems deliver more disk storage capacity.

  Large disk-capacity VAXcluster systems coupled with industry-leading disk technology, very competitive disk products, and excellent information management software products allow VAX to compete in large database applications. Customers can attach disk storage units to VAX processors via UNIBUS and MASSBUS controllers or can configure disks through the HSC50. In either case, disks can be dedicated to a VAX processor, a department, a user group, an application, or can be shared across the VAXcluster system. The result is more disk storage capacity. Disk storage will soon be measured in gigabytes. VAXcluster systems and the Digital Storage Architecture deliver in this range.

With these characteristics, VAXcluster systems improve our competitive position at the high end of the VAX performance range. They will take VAX to large system, and large information management applications.

VAXcluster systems extend the VAX range to higher system availability, allowing us to better compete in this market segment. The VAXcluster system thus, has redundant features which customers can apply to achieve higher levels of system availability. These features are implemented in hardware such as the CI780, CI750, and HSC50; and in software as VMS components.

VAXcluster systems are unique. They allow customers to implement systems to achieving higher performance or higher availability or both. The customer's application will dictate the degree of each. This unique capability now takes VAX to new applications and to new customers.

VAX will now compete against mainframe suppliers, companies with special purpose point products and other emerging companies who are competing in this market segment. The competitive section of this Sales Update special issue details some product comparisons which will help you understand our product advantages in this market segment.

VAXcluster SYSTEMS: A NEW GROWTH ALTERNATIVE FOR THE VAX ACCOUNT.

For today's VAX customer, VAXcluster systems offer a new growth alternative. To achieve more processing capacity, a customer can add a processor to a cluster. Either a 750, 780, or 782 can be selected to meet the application need. To achieve more disk storage capacity, a customer can add more disk units and distribute those disk units on processors, or on HSC50 mass-storage controllers to achieve capacities never before attainable on VAX.

Every installed VAX-11/750, 780, or 782 is a VAXcluster system prospect. Every one of these systems can be upgraded to a VAXcluster system. Don't overlook the installed base as a source of bookings.

In summary, VAXcluster systems allow you to sell where you could not sell before. High capacity and high availability applications are prospects for VAXcluster systems. Present VAXcluster systems to your existing VAX accounts.

Show them how the unique and easy growth of VAXcluster systems can continue to deliver capacity as they may need it while preserving their investment in VAX products.

VAXcluster systems: The new way to grow systems.

OVERVIEW

V A X B A S E P R O D U C T M A R K E T I N G

VAXcluster SYSTEMS: PROGRAM SUMMARY

Terry Retford
DTN: 247-2930
TWO/B02 RCS:TWSK

VAXcluster systems represent a new concept in VAX computing. The concept and implementation offer users new ways to grow their systems in many dimensions including: capacity, systems availability, and data integrity.

Before introducing the product, it is important to understand some of the goals and design tradeoffs in adopting the VAXcluster System approach. There are two key system goals:

1. Adopt a computing approach that will grow and extend easily over time, allowing our customers to utilize a variety of technologies and to incorporate new technologies as they become available.

2. Offer customers high, overall systems availability.

To satisfy these goals we have adopted a loosely coupled multiprocessor systems approach. Loosely coupled multiprocessor systems are characterized by multiple processors, each with a copy of the software operating system and separated by a message oriented interprocessor systems protocol.

An alternate approach is tightly coupled multiprocessing, with processors in close proximity, connected through shared high-bandwidth memory, and with a single copy of the software operating system. An example of this approach is the VAX-11/782.

The loosely coupled approach allows the incorporation of processors of differing types into the system. They can be of different performance and different states of technology, thus as we develop new processors incorporating new technology we can easily integrate them into VAXcluster systems.

In addition to processors, mass-storage servers are separate elements within the system connected to processors through dual paths. This provides a system with no single point of failure in the component interconnection and therefore, higher overall systems availability. The use of separate processors with independent operating systems increases the survivability of the system in the presence of operating system failures.

Loosely coupled multiprocessor system performance is determined by the bandwidth of the interconnect between the system elements and the software overhead associated with this interconnect. To provide maximum system performance, we have taken three steps:

1. We have developed a very high speed, message-oriented computer interconnect (CI).

2. We have developed an efficient systems-level protocol with functions tailored to the needs of highly available systems.

3. We have developed an intelligent hardware interface to the CI which implements certain key functions of the systems-level protocol.

These three key developments are important characteristics of VAXcluster systems.

Data integrity features supplement the high availability and growth considerations of VAXcluster systems. Through the use of future VMS facilities such as Common Journaling, Recovery Units, and Checkpointing, users can develop applications to ensure that the data in the VAXcluster system is always in a known state. For example, databases using the Common Journaling Facility may be rolled back or forward to restore them to a known state. Redundant hardware may be configured to ensure the availability of data in the event of failure. The combination of redundant hardware and VMS features allow the user to implement an environment where files with high levels of both data integrity and availability are present.

PRODUCT DEFINITION

A VAXcluster system consists of several components:

- a high performance communications path
- VAX processors
- VAX/VMS
- Intelligent mass storage servers

The basic backbone of the system is a 70 megabits-per-second high-speed, dual-redundant path system link that connects all of the other components in a VAXcluster system together. VAXcluster systems consist of nodes which are either VAX processors or mass-storage servers. VAX processor nodes can be VAX-11/750s, 780s, and 782s. The mass-storage servers are HSC50 Hierachical Storage Controllers for disks and tapes. VAX/VMS software is necessary to run the VAXcluster system and provides support for HSC50, 750, 780 and 782 as well as the disks and tapes connected to the HSC50 and disks and tapes locally connected to the VAX processors.

HARDWARE COMPONENTS -- (See Figure 1)

The CI (Computer Interconnect)

The CI is a high-speed, fault-tolerant, redundant bus which allows up to 16

nodes to be logically multidropped, in a computer room environment. Nodes in a cluster use a multi-access bus topology which allows every node to communicate directly with every other node.

The CI allows information transfer between nodes at a rate of 70 megabits per second. The CI has an immediate acknowledgement scheme where channel time is reserved at the end of the message for the destination node to acknowledge the message.

The built in redundancy of the CI prevents a single point of failure from interrupting the cluster operation. Additionally, since no single node is bus master, the removal of any node from the cluster does not preclude continued communication among the remaining nodes.

CI780 & CI750 Interfaces

The CI interfaces are microcoded intelligent controllers which connect VAX processors (11/780, 11/782 and 11/750) to the CI. Each interface attaches to one CI bus which consists of two transmit and two receive cables. Traffic is transmitted on whichever path is available. If both paths are available then a performance benefit results. If a path becomes unavailable, then all traffic uses the surviving path. VAX/VMS tests a failed path periodically. As soon as a failed path becomes available it is used again for normal traffic --automatically.

Star Coupler

The Star Coupler (SC008) is the common connection point for all nodes connected to the CI. It connects all CI cables from the individual nodes together in a radial or star arrangement. The maximum cable length is 45 meters, thus all nodes are required to be located within the 45 meters of the star coupler. This physical cabling arrangement does not affect the logical appearance of the cluster as a multi-access bus.

The coupler is available in an 8 node version and can be expanded via an 8 node upgrade to handle 16 nodes.

The Star coupler provides passive coupling of signals from all nodes via power splitter/combiner transformers, and will allow any node to be added or removed during system operation. IN and OUT connectors are provided per CI path for each node. A signal received from an IN connector is distributed to all OUT connectors. The Star Coupler terminates all cables with their characteristic impedance.

HSC50 (Hierarchical Storage Controller)

The HSC50 is a self-contained, intelligent, mass-storage server that interfaces one or more processors to a set of mass-storage devices (disks and tapes). The HSC50 is attached to the processor(s) via the CI, and uses MSCP (Mass-Storage Control Protocol) for processor communications. Communication between the HSC50 and mass-storage drives is through the Standard Disk Interface (SDI) and the Standard Tape Interface (STI).

The first implementation of the VAXcluster architecture is based on the Computer Interconnect which supports up to 16 nodes. Each node may be a VAX-11/782, VAX-11/780, VAX-11/750 processor or an HSC50 controller. Digital

is currently offering VAXcluster configurations in which each HSC50 controller can provide storage access for as many as four processors. Later, we will offer VAXclusters where each HSC50 will provide storage access to more processors.

The HSC50 can off-load the processors of some utility operations such as disk shadowing, volume copying, and image backups by performing these operations itself. The HSC can handle multiple, simultaneous operations on multiple drives and will optimize the physical operations (e.g., seek optimization, rotational position optimization) to maximize throughput.

On command, the server can create a shadowed volume set. The shadowed volumes are identical at the logical block level.

The subsystem has extensive internal diagnostics and a level of internal redundancy. It also has the ability to continue operation in a degraded mode (e.g., disabled disk and/or tape interface modules or reduced memory) with minimal reduction in throughput. The CI and SDI/STI use transformer coupling so they can be disconnected and reconnected without disrupting operations on either the CI or the SDI/STI.

Architecturally, each HSC50 has the ability to support a combination of up to 6 SDI or STI interfaces. Each SDI can support 4 disks to a maximum of 24 disks per HSC.

SOFTWARE SUPPORT -- (See Figure 2)

VAX/VMS V3.3

The cluster functionality available at this release will support a mix of VAX-11/780, VAX-11/782 and HSC50 controllers (disk support only) connected to a single CI (one CI780 on each processor). Each VAX-11/780 on the CI must have its own system disk although no local mass storage (attached to a processor's UNIBUS or MASSBUS) is required. For VMS distribution purposes, there must either be an RA60 present on the HSC or a local tape or disk drive. Sharing of disk volumes on the HSC50 is possible with access from any VAX-11/780 processor on the CI (under the constraint that only one processor can read/write to a volume while all others can read only).

DECnet has been implemented over the CI for cluster management and normal communications (throughput comparable to a DMR). If the customer has a DECnet/VAX license, local disks on other VAX processors can be accessed using DECnet/VAX. VAX-to-VAX communications at the user level are also possible using the DECnet/VAX interface.

Note: With this release, support for VAX-11/750 processors in VAXcluster systems is not available.

VAX/VMS V3.4

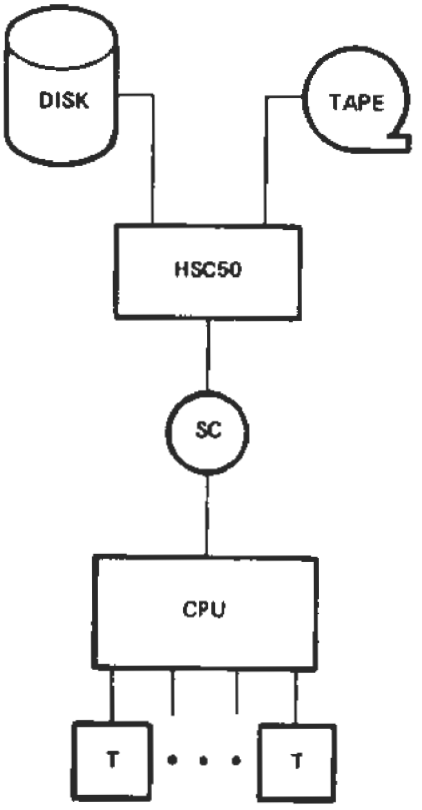
VMS Version 3.4 extends VAXcluster support to the VAX-11/750; a local disk is required on each VAX-11/750 for booting VAX/VMS. VAX-11/782s, VAX-11/780s and VAX-11/750s can be mixed with HSC50s on the same CI. For VMS distribution purposes, there must be either an RA60 present on the HSC50 or an appropriate local tape or disk drive.

Future VMS Releases

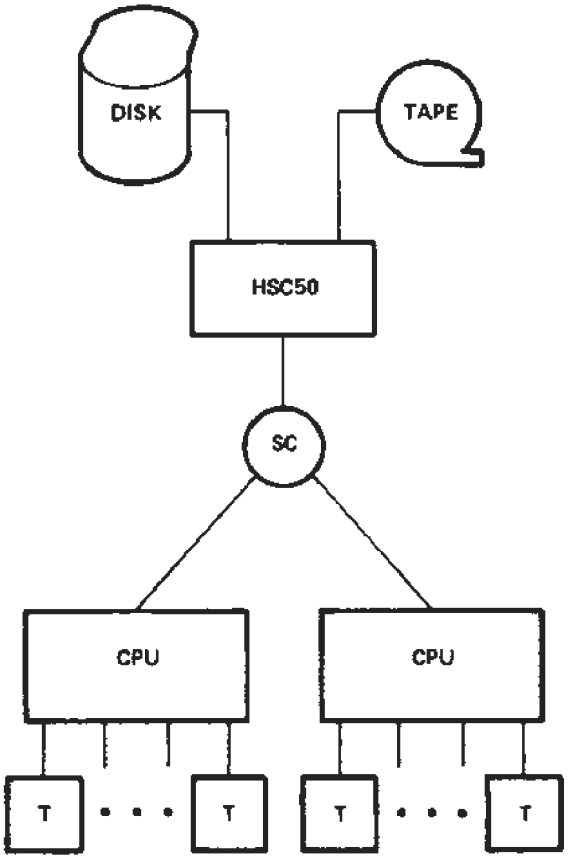
Future VMS releases will include the capabilities described in more detail in the next article.

TYPES OF CONFIGURATIONS

1. Single CPU, single HSC - this is viewed as a replacement for the current MASSBUS based systems offering increased performance and availability. These systems should be sold into applications where VAX-11/780 systems are currently sold today.



2. Multiple CPU systems with single HSC. More CPUs add processing power. These may be added in incremental fashion for either increased compute power or added number of interactive users.



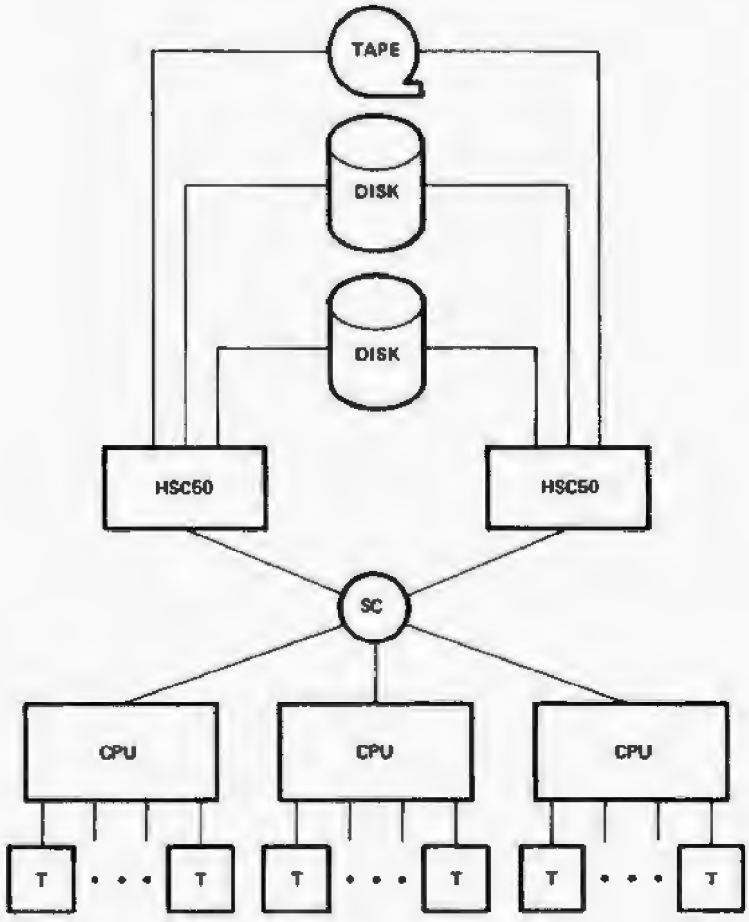
3. Multiple CPUs with Multiple HSCs

This type of configuration could be either used for added capacity, added performance or added availability. Here we anticipate that the customer would fit into one of two categories:

1. The customer who is availability conscious, but has learned to tolerate some down time will adopt an n+1 strategy (have one additional unit of each of the components-- CPU, HSC, and disk storage or tape devices).

2. The customer who is extremely concerned about availability-- one who would want to recover very rapidly-- will adopt the n+2 strategy. Here two components of any type could fail without affecting the overall system availability.

(see configuration example on the next page)


TAPE
DISK
DISK
HSC50
HSC50
SC
CPU
CPU
CPU
T
T
T
T
T
T

SUMMARY

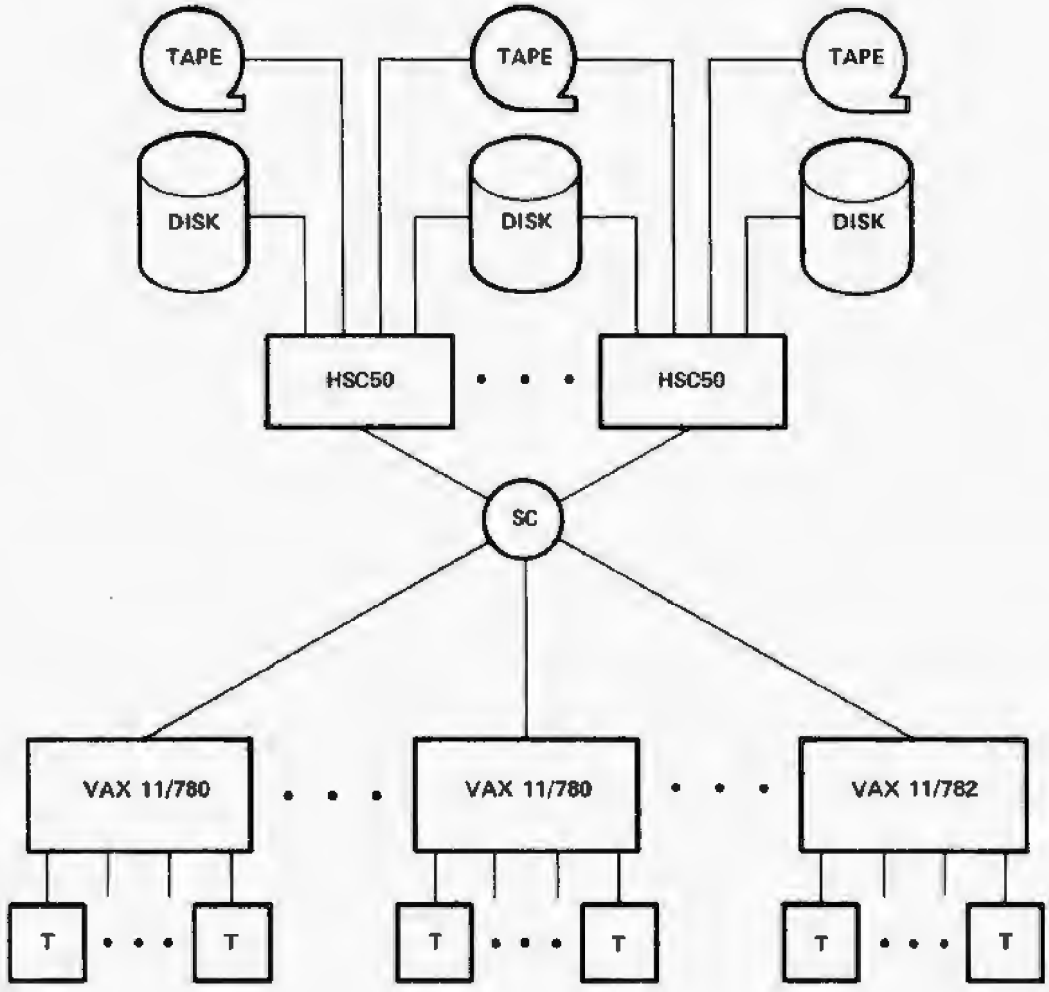
We have shown that the VAXcluster systems approach allows us to meet the goals of incremental growth or systems extendibility, high system availability, and data integrity.

We have introduced you to all the necessary hardware components to support the VAXcluster system approach and the software to support this hardware. The advanced software functionality to support the VAXcluster environment will be introduced in the future. More details about the future software functionality are contained in the next article. See Appendix A for a pricing and availability summary.

In taking the VAXcluster system approach we are on a sound foundation to meet the needs of our customers, and the market in general for the 1980s.

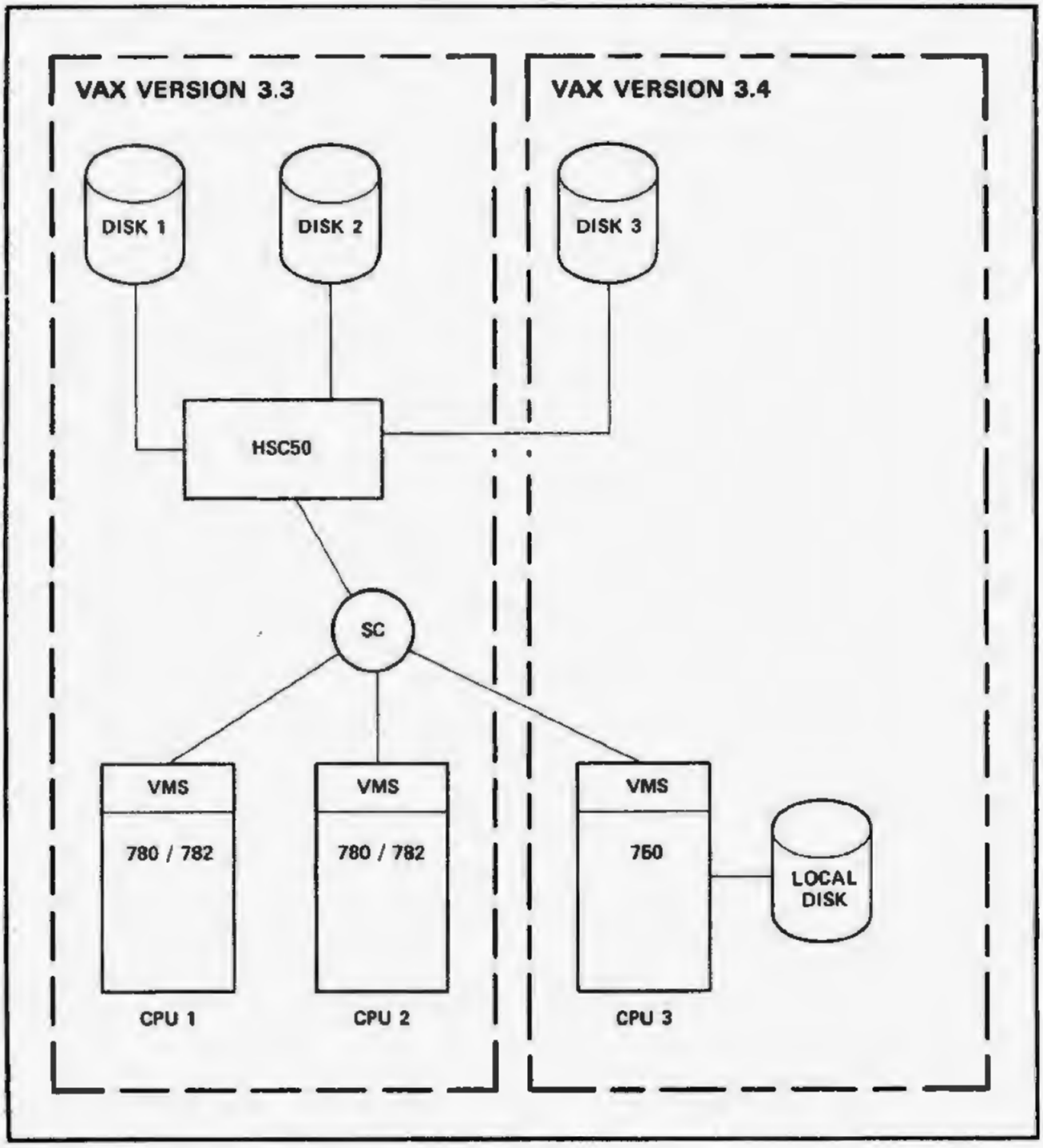
VAXcluster systems --- The new way to grow systems.

Figure 1



Up to 16 VAX Processors and/or HSC50 Controllers
(Up to 4 Processors/HSC50)

Figure 2


VAX VERSION 3.3
VAX VERSION 3.4
DISK 1
DISK 2
DISK 3
HSC50
SC
VMS
780 / 782
VMS
780 / 782
VMS
750
LOCAL DISK
CPU 1
CPU 2
CPU 3

PRODUCT DETAILS

VMS BASE PRODUCT MARKETING

VAXcluster SOFTWARE

Trevor Kempsell
DTN: 264-8325
ZK01-1/C27

VAX/VMS operating system software is integral to the operation of the new VAXcluster System, and is a key component of the cluster architecture. VAX/VMS provides the software intelligence and logic which runs the VAXcluster. By design, VAX/VMS implements facilities which provide high system availability, multi-system file/record sharing, (with the ability to accept new CPUs) and mass storage into the VAXcluster as needs arise. The following article describes the VAX/VMS software features relative to the VAXcluster environment. This software will be introduced in phases. Please consult the table at the end of this article.

VAXcluster Systems provide global data availability through the use of:

- A distributed file system that allows all VAXcluster mass storage to appear as if it were local to any CPU in the cluster to the file level
- A distributed lock management facility to synchronize access to the files/records
- The Mass-Storage Control Protocol server (MSCP server) facility that gives cluster-access to locally connected mass storage

The VMS tools used to achieve data integrity are backup, volume shadowing, journaling, and recovery unit journaling. Backup, which is a fairly traditional solution, is still important as a supplemental means of providing a minimum amount of data integrity. Volume shadowing provides immediate access to data at all times. It creates multiple copies of data as a safety mechanism in the event that data becomes inaccessible or destroyed. Journaling is a method of maintaining a record of the changes made to data. In addition, recovery unit journaling provides the mechanism for maintaining a record of a specified set of operations; this record can be used immediately to restore data to its original state.

Process integrity is provided with checkpointing. Checkpointing gives the user the ability to restart a long-running process from user-specified checkpoints.

VAX/VMS Data Integrity Capabilities

VAX/VMS has several components to provide data integrity capabilities. The following table summarizes these components that are available in VMS VAXcluster systems.

Data Integrity Components

Component	Description
Common Journaling Facility	Provides capability to write and read journals
Recovery Unit Facility	Guarantees that a specified set of multiple operations will be atomic. Initiates automatic recovery.
Recovery Control Process	Performs rollback and rollforward operations. It reads the journals and invokes the recovery routines for all recoverable facilities.
Record Management System Journaling (RMS Journaling)	Provides a simple to use interface for journaling RMS operations.

These components work together to provide VAXcluster data availability and data integrity. These features, which are of obvious value in cluster environments, can also be used with nonclustered VAX/VMS systems.

The Common Journaling Facility is the central component for data integrity. It can be used to provide four types of data integrity functions:

- Reversibility of a series of operations: allows the database to be rolled back
- Availability of data despite corruption of files or disks: allows the database to be rolled forward.
- Audit trail capabilities: allows activities or changes to be logged for audit purposes.
- Integrity of a set of file operations: allows atomic transactions to be implemented.

Reversibility of a Series of Operations

A computer system might correctly perform all operations asked of it, but it can be given incorrect data. A user can discover that the wrong job or transactions were run owing to errors external to the computer. Common examples are operational and data entry errors. A user could discover after running a series of jobs or transactions that a number of incorrect database operations had been performed because of errors in the application software.

In such cases, the user will want to restore files to a previous state known to be correct. Restoring objects to a previous state is known as a rollback operation. The Common Journaling Facility uses before image journals to provide rollback recovery.

VAX-11 RMS uses this capability to provide reversibility of RMS file operations.

Availability of Data Despite Corruption of Files or Disks

A database can become corrupted for any number of reasons-- for instance, incorrect recording of data due to disk subsystem errors, programming errors in the application software, or system software errors. In such cases the user will want to reconstruct corrupted files from their most recent noncorrupted state, using the data that had been entered. Reconstructing objects using data that was entered is known as a rollforward operation. The Common Journaling Facility uses after image journals to provide rollforward recovery.

The user first creates a copy of the object before any updates are made. This backup copy is used with the changes that were recorded in the after image journal to create a reconstructed version, up to a specified point in time.

VAX-11 RMS can use this after image journal capability to maintain a record of changes to RMS files. It can use the recovery utility to apply the changes in the after image journal to the base copy of a file, and thus create a reconstructed file. The recovery utility can be used to restore specific files or all journaled files on a volume.

Audit Trail Capabilities

An application environment that has certain security requirements might want to maintain a continuous log of specific system events. An obvious example is keeping an audit trail of all users who access a particular file and whether they perform read or write operations on the file.

The Common Journaling Facility provides audit trail journals and mechanisms to extract the information from the journals. VAX-11 RMS supports audit trail journals for maintaining information about operations on RMS files.

Integrity of a Set of File Operations

Database users commonly want to perform a series of file operations with the guarantee that they will be performed as an atomic action, that is, all of them will be done completely, and correctly or none of them will be performed at all. If only some of the operations are completed, the database is considered to have lost data integrity and can be considered corrupted.

The Recovery Unit Facility provides a mechanism for identifying a set of operations to be performed as a single action. The Recovery Unit Facility establishes the framework within which a recoverable facility will write information into a recovery unit journal. In addition, the Recovery Unit Facility initiates recovery when a failure occurs.

The Common Journaling Facility provides the recovery unit journals, which are used to store information about the changes. When a failure such as a system failure or a head crash occurs, the Recovery Control Process reads the journal and invokes the appropriate recovery operations.

VAX-11 RMS provides recovery unit protection for certain operations performed on RMS files. If a file is specified as requiring recovery unit protection, VAX-11 RMS automatically writes recovery information to the recovery unit

journal. If a failure occurs before a recovery unit completes, the file is automatically rolled back to the start of the recovery unit.

Facilities and Data Integrity

The VAX/VMS data integrity features have been described in terms of RMS file operations. These features can also be used by user-written software facilities, such as database management or transaction processing packages. These are called recoverable facilities. A recoverable facility is a privileged software layer that performs tasks for applications and has been modified to provide recovery capabilities using these VAX/VMS data integrity components.

VAX-11 RMS is an example of a recoverable facility that is supplied by Digital. VAX-11 RMS provides all forms of recovery that are available: rollforward, rollback, audit trails, and recovery unit protection.

The Common Journaling Facility and the Recovery Unit Facility are the two components that work together to provide the basis for the data integrity features for RMS files. By enhancing VAX-11 RMS, we have provided this level of data integrity into RMS file operations. In addition, knowledgeable users can write their own recoverable facilities to work with the Common Journaling Facility and the Recovery Unit Facility.

Protection of Computation Investments

Hardware or software failures are measured in terms of the cost of computation lost and jobs not completed on schedule. The Checkpoint/Restart Facility is designed to protect long-running processes. The Checkpoint/Restart Facility permits an image to be restarted in a well-defined state, following a system failure, thereby preserving investment in computation time. To use this facility, checkpoint calls are inserted at strategic points in a program. A checkpoint call saves the state of a process. The most recently executed checkpoint call becomes the point at which a process can be restarted after system failure or abnormal process termination. The saved process state includes information on address space, I/O channels, and miscellaneous data for restoring logical names.

Device Transparency and File/Record Access

There are several software components that work together to produce the high availability and the transparent sharing of data on all disk mass-storage in the VAXcluster environment. Of these software components, the three major ones are VAX-11 RMS, the Distributed File System, and the Distributed Lock Manager. All of these components exist on each VAX processor in the cluster to avoid any single point of failure.

VAX-11 RMS uses the Distributed Lock Manager, giving the user record level, cluster-wide access to any disk mass-storage. The necessary interlocking is handled by the Distributed Lock Manager.

The Distributed File System

The Distributed File System allows all VMS processors in a VAXcluster to share disk mass-storage in a transparent way. A disk volume shared in this way appears to each CPU as a local disk. All access to such a disk from any level above the disk driver works transparently, as if the disk were local.

The shared disk can be directly connected to one of the cluster processors as with a MASSBUS-connected disk. In this case, the Mass Storage Control Protocol (MSCP) Server on the processor with the local disk, transparently provides the equivalent MSCP services. VAX-11 RMS uses the Distributed File System and the Distributed Lock Manager to provide file and record-level access to disk storage throughout the cluster.

A shared file system with the MSCP server permits incremental growth by allowing additional CPUs, with their own local disk storage to be added to the cluster at a later time. A shared file system will then allow users on the existing VAXcluster CPUs to share the new, local-processor-owned file system.

The Distributed Lock Manager

The VAX/VMS Distributed Lock Manager is a tool for synchronizing access among resources for processes on a single CPU or in a cluster. The Lock Manager provides a namespace in which processes can lock and unlock resource names. In addition, it provides a queuing mechanism so that processes can be put into a wait state until a particular resource is available. In this way, cooperating processes can synchronize their access to shared objects, such as files or records.

In order to increase concurrency-- to allow as much sharing as possible-- the Lock Manager provides two features. First, the namespace provided by the Lock Manager is tree-structured. This allows applications to lock objects at varying degrees of granularity, for example, a specific record or an entire file. Secondly, the Lock Manager provides a number of different lock modes, in addition to the usual shared and exclusive lock modes. The Lock Manager also includes functions that assist facilities, such as VAX-11 RMS in performing distributed buffer management.

The Distributed Lock Manager is integrated into VMS. The algorithms that actually implement the distributed locking capability are highly cooperative and designed to minimize interprocessor communications thus, optimizing performance.

The Lock Manager also implements deadlock detection. This means that the Lock Manager will not allow a circular list of processes to wait for each other. In the cluster environment, deadlock detection works cluster-wide in that processes forming a deadlock can be located on different cluster nodes. On failure of a processor node in a cluster, the Lock Managers in the remaining nodes release locks held by the failed processor node after allowing the recovery unit facility to perform any necessary operations.

The MSCP Server

MSCP (Mass Storage Control Protocol) is a protocol for logical access to disks and tapes. The VMS MSCP Server implements the disk-only portion of this protocol. This permits any VAX/VMS processor in the cluster to access UNIBUS or MASSBUS disks that are locally connected to another VAX processor cluster node. The MSCP Server also includes volume shadowing capability for UDA-type (UNIBUS Disk Adapter) disk drives. These UDA disks appear as if they were logically error-free. Incoming I/O requests from other processors in the cluster are received by the MSCP Server. The MSCP Server uses the standard VAX/VMS device-driver interface to communicate with the local disks and passes the data back over the CI to the requesting CPU.

Job Queuing in a Cluster

Normally, a VAXcluster system operates with a common set of batch and print queues for the entire cluster. The user can submit jobs to any queue within the cluster, provided that the necessary mass-storage volumes are accessible to the system on which the job executes. An operator can perform queue management functions on any queue.

The user, operator, and system manager have a great deal of freedom in configuring, managing, and using the batch and print queues within the cluster. For example, the system manager can define generic print queues that schedule jobs only on the local printers for each system, or jobs can be sent to any available printer, or any desired subset of the printers in the cluster.

Generic batch queues, coupled with batch execution queues that execute on specific systems within the cluster, enable the batch workload to be shared. The default scheduling policy balances the shared batch workload across the cluster by keeping the ratio of active batch jobs to available batch slots as equal as possible on each processor node.

Cluster-Wide Communications

Transfers across the CI do not involve DECnet protocols. Instead they use new VMS system level protocols. DECnet-VAX, however, uses these protocols to utilize the CI as the physical link between VAX processor nodes in the cluster for DECnet communication within the cluster.

DECnet protocols are used to allow VAXcluster processors to participate as nodes in any DECnet network. Such a VAXcluster DECnet node has all of the functionality of a non-clustered VAX/VMS system.

High Availability

The high availability, or constant accessibility, of correct data is another benefit provided by the VAXcluster. The following table summarizes possible system failures and the features of VAXcluster systems that maintain system availability.

VAXcluster/VMS High Availability Features

Failure Type	Applicable High Availability VAXcluster Features
CPU Failure	Multiple CPUs, Recovery Unit Facility, Common Journaling Facility, Checkpoint/Restart Facility
CI Bus Failure	Dual path, Dual CIs
HSC50 Failure	Multiple HSC50s with dual-ported disks/tapes
Disk Drive Failure	Redundant drives, Volume Shadowing (automatic switchover to shadowed volume, transparent to user), Common Journaling Facility, Recovery Unit Facility, Backup

Dual Porting of DSA Disks/Tapes

Digital's DSA (Digital Storage Architecture) disk drives, the RA80, RA60, and RA81, can be dual-ported between two HSC50 controllers to enhance system availability. On failure of one HSC50 controller in a dual-controller configuration, there is an alternate path to the disk through the second HSC50. Only one path to a disk is active at a time. This path remains active until either a controller fails or until the system manager specifically takes action to redirect operations to the alternate path.

All CPUs in a VAXcluster system have the capability of accessing any HSC50 disk by means of any active path. A DSA disk drive could be dual-ported between a UDA and an HSC50, but no automatic failover to the alternate path is supported. The same applies to disks dual-ported between two UDAs.

Tape drives can be dual-ported in a similar manner, except that by nature, a tape drive is logically allocated to a CPU while in use and does not have multiple users or shared usage.

A Highly Available System

VAXcluster systems can be configured to withstand the failure of various components in the cluster and yet still maintain a level of system availability. The level of availability is determined by the amount of hardware redundancy in the cluster and the VMS features used, and is matched to satisfy the particular application's availability requirements. A fully redundant cluster exists when there are two identical processors and both have two CI interfaces, two HSC50s-- one connected to each CI, and disks and tapes dual-ported between the HSC50s. In this configuration there is always a path to the HSC50 mass storage no matter which single component fails. Volume shadowing makes the failure of a specific volume transparent to the application.

If a CPU fails, local mass storage on that CPU is no longer accessible to the cluster. The integrity of data and investment in CPU time can be maintained despite such a failure by using the data integrity and checkpointing tools provided by VMS. As long as there is still a path to the journal and/or the checkpoint data on mass storage, the recovery and/or restart can take place on another node in the cluster. In order to restart from a checkpoint, the restart must occur on an identical CPU. An identical CPU is defined to be one that is the same processor model number, has the same processor options, and is at the same VMS and hardware revision level. The VMS software components, specified in this chapter, can identify the failure of a component and can take the necessary action to keep the rest of the cluster operating despite such a failure.

Redundancy at this level is an extreme example, but might be appropriate for certain applications. A single CI with two HSC50s and dual-ported mass storage would be more typical of a VAXcluster system configured for redundancy.

When a hard CPU failure, as opposed to a transient power failure has been identified by VMS, another CPU in the cluster will automatically invoke data-base recovery, if applicable, and roll-back files to the start of a recovery unit. When a disk is logically mounted by VMS, a check is made for outstanding recovery-unit data on that volume and recovery takes place. For

checkpointed images on the failed processor, the image can be restarted on another, identical CPU in the cluster. Operator action is required to invoke the restart on the identical processor.

In summary, tools are provided to the user for implementing a variety of data integrity levels and process auto-restart/failover.

VAX/VMS V3.3, V3.4 AND FUTURE RELEASES FEATURE SUMMARY

The following is a description of the software and hardware functionality as supported with the VAX/VMS V3.3 and V3.4 releases. The last section is a chart showing the levels of cluster functionality supported by the various releases. This chart is useful for planning the growth of applications and VAXcluster systems.

VAX/VMS V3.3 Cluster Support

The VAXcluster functionality provided in the V3.3 release supports a mix of VAX-11/780s, VAX-11/782s, and HSC50 mass-storage controllers with shared disk support only. These cluster nodes are connected to a single CI, one CI780 on each processor. Each VAX-11/780 on the CI must have its own system disk, although no local mass storage is required. For VMS distribution purposes, there must be either a RA60 disk present on the HSC50, or an appropriate local tape or disk drive.

Dual-porting of disks is possible where the active path from the HSC50 to the disk is defined when the disk is logically mounted. Sharing of disk volumes on the HSC50 is possible with access from any VAX-11/780 or VAX-11/782 processor on the CI, with the constraint that only one CPU can write, and all the others can read only.

If an HSC50 volume is used in this way, new files and extensions to old files are not accessible at the read-only CPUs until the directory caches have been written to the disk volume. VMS file system caching has to be disabled for all the readers and the writer on the specific volume to ensure "old data" is not passed.

The HSC50 at VAX/VMS V3.3 release does not provide the ability to do volume shadowing or to have diagnostic capability from the host. DECnet-VAX is implemented over the CI for cluster management purposes and normal communication (throughput comparable to DMR). If the customer has a DECnet-VAX license, local disks on other VAX processors can be accessed using DECnet-VAX over the CI. Likewise, communication between a VAXcluster node and other remote network nodes is also possible using the DECnet-VAX interface. The Distributed Lock Manager, MSCP server, Distributed File System, and the data integrity/checkpointing facilities are not included in the VAX/VMS V3.3 release.

VAX/VMS V3.4 Cluster Support

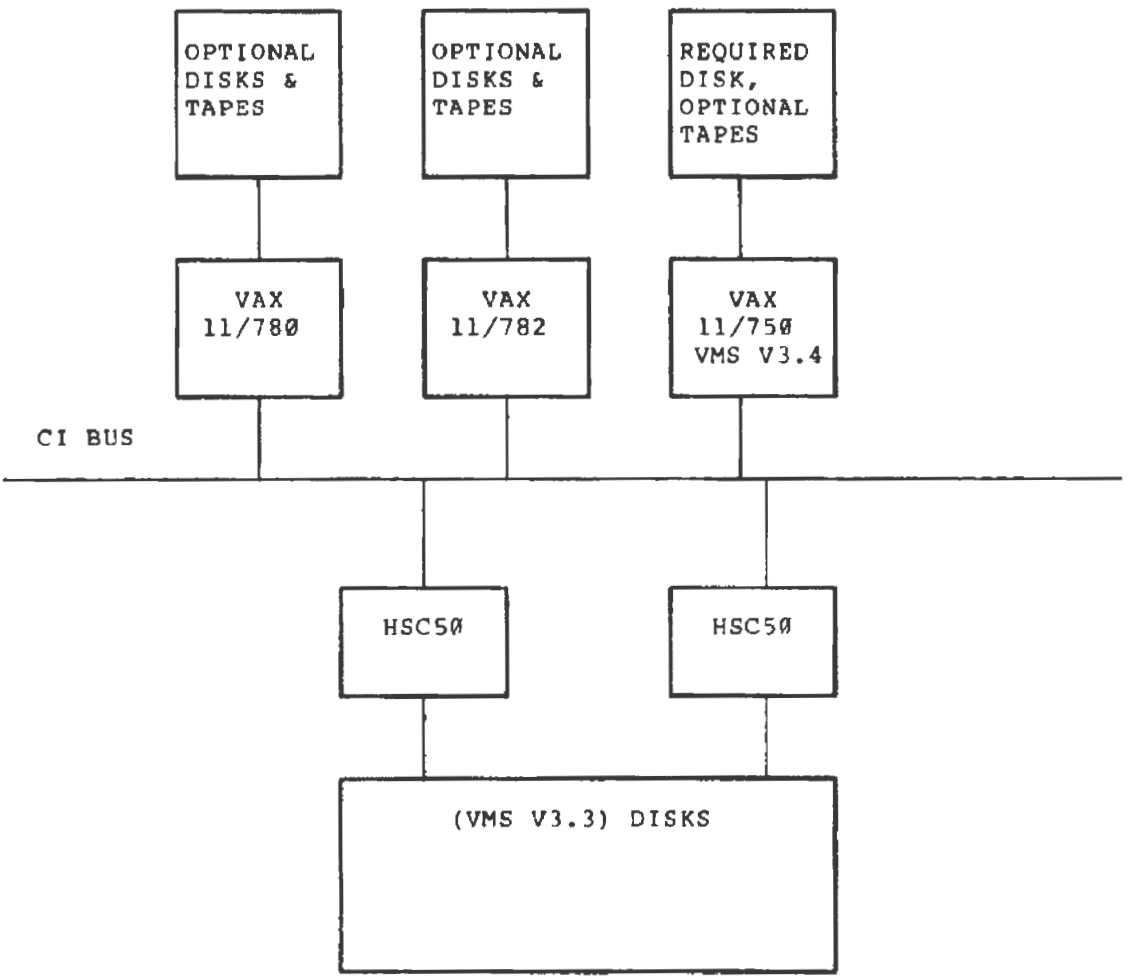
The VAXcluster functionality provided in the V3.4 release extends the V3.3 features to the VAX-11/750. A local disk is required on the VAX-11/750 for booting the VAX/VMS operating system. VAX-11/782, VAX-11/780, and VAX-11/750 processors can be mixed with HSC50s on the same CI. For VMS distribution purposes, there must be either an RA60 disk present on the HSC50 or an appropriate local tape or disk drive. The Distributed Lock Manager, MSCP server, Distributed File System, and the data integrity/checkpointing

facilities are not included in the VAX/VMS V3.4 release.

The following figure shows a typical VAXcluster configuration that is supported with the VMS V3.3 and V3.4 releases.

NOTE: The cluster announcement in no way affects Digital's present software licensing arrangements. Therefore, although the actual software code might reside on a disk off the HSC, the license for the software is CPU specific in the same manner as today's existing software contracts.

TYPICAL VAX/VMS V3.3/3.4 CLUSTER CONFIGURATION



VAXcluster Feature Support Table

The following table depicts the level of VAXcluster support that is applicable to VAX/VMS V3.3, V3.4, and future operating system version releases. By using this table, proper near-term VAXcluster expectations can be set and longer term, application growth and cluster development plans can be formulated.

VAXcluster Feature Support

Key to table: X = Facility Introduction * = Restriction no longer applicable

FEATURE	VMS V3.3	VMS V3.4	FUTURE
CI - 11/780, 11/782 DECnet-VAX support	X (V3.1)		
CI - 11/750, 11/780, 11/782, DECnet-VAX support		X	
HSC50 support	X		
DISKS	X		
A CPU HAS ACCESS TO A SPECIFIC VOLUME	X		*
VOLUME SHARING (One Writer, Muliple Readers)	X		*
SEEK OPTIMIZATION	X		
ROTATIONAL POSITION OPTIMIZATION	X		
DISK DUAL PORTING (Static active path)	X		
AUTO FAIL/OVER (Static active path)			X
TAPES			X
A CPU HAS ACCESS TO A SPECIFIC DRIVE			X
TAPE DUAL-PORTING (Static active path)			X
VOLUME SHADOWING			X
HOST DIAGNOSTIC CAPABILITY			X
VAX-11/780s, VAX-11/782s & HSC50s on the CI	X		
Each CPU requires its own system disk (local or HSC50)	X		*
No local mass-storage required	X		
VAX-11/750s become part of the VAXcluster		X	
Local disk mass-storage required (for booting)		X	*
Data Integrity Facilities			X
Common Journaling Facility			X
Recovery Unit Facility			X
VAX-11 RMS Journaling & Recovery			X
CHECKPOINT/RESTART FACILITY			X
DISTRIBUTED LOCK MANAGER			X
CLUSTER-WIDE TRANSPARENT DISK ACCESS (Local or HSC)			X
DISTRIBUTED FILE SYSTEM			X
File/Record level sharing on all disks			X
MSCP SERVER			X
Cluster-wide access to local disk storage			X
CLUSTER-WIDE BATCH/PRINT QUEUES			X

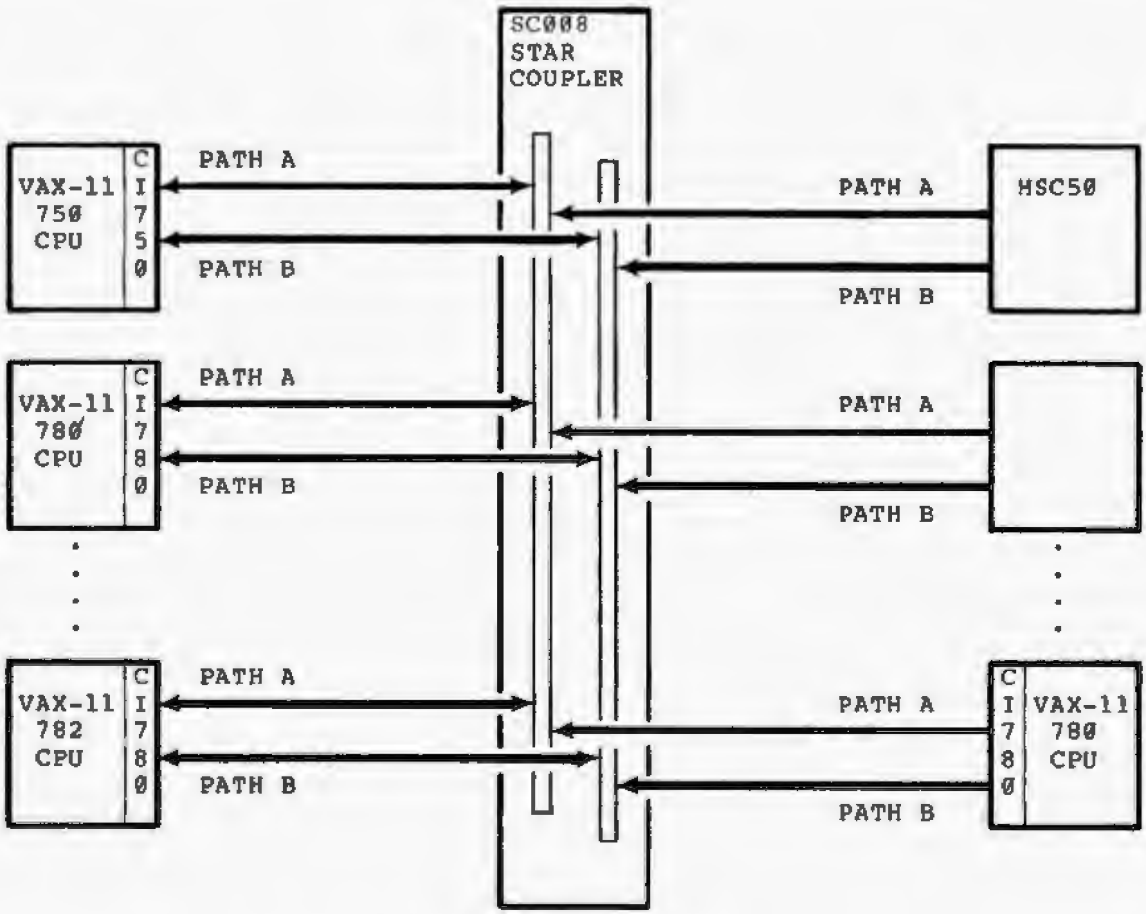
COMPUTER INTERCONNECT (CI) SUPPORT EXTENDS TO VAX-11/750

Scott Garren
DTN: 247-2417
TWO/B02 RCS:TWSK

Joining the previously announced CI780, the CI750 allows your customers to include the high level of price/performance and low cost-per-user of the VAX-11/750 in a VAXcluster system.

PRODUCT DESCRIPTION

The Computer Interconnect (CI) provides high-speed, fault-tolerant communications for up to 16 processor and HSC50 nodes in a VAXcluster system. While often referred to as the CI or CI bus, there are three classes of components involved -- CI adapters, CI cables, and a Star Coupler. CI adapters, such as the microprocessor-controlled CI750 and CI780, interface the VAX to the CI cables. CI cables connect the adapters to a star coupler which, in turn, serves as the common connecting point for the nodes in a VAXcluster system.



FOR INTERNAL USE ONLY

Performance

The nature of a VAXcluster environment dictates a special need for high-performance internode communications. Hence:

- Microcode-controlled CI adapters provide data buffering, address translation, and encode/decode functions. This adapter "intelligence" minimizes the overhead required to complete typical high-level, intercomputer communications.

- The communication path between the VAXcluster nodes has been designed to support a data transfer rate of up to 70 megabits per second.

Availability and Reliability

The CI has been designed to meet the high availability and reliability aspects of a VAXcluster environment.

- All communication paths between the nodes in a VAXcluster system are dual path-- from the CI adapters, to the cables, to the star coupler. If a failure occurs on one path, the surviving cable will automatically pick up the retransmission. This built in redundancy prevents a single point of failure from interrupting operation.

- Since no single node is bus-master, a node may be removed from, or added to the VAXcluster system. Maintenance can be performed on individual nodes, and a new node can be added-- all without interrupting the operation of the rest of the cluster.

- On a less grand, but still important scale, you'll find high reliability features such as shielded coaxial cables, signal isolation, and noise rejection.

CONFIGURATION GUIDELINES

General

- The maximum cable length from a CI750 or CI780 to the Star Coupler is 45 meters.

- A minimum of 2MB of memory is required by VAX/VMS in a VAXcluster environment.

VAX-11/750 Configuration

- The CI750 requires one CMI option slot in the CPU backplane.

- In addition, an expander cabinet is required to house the 10 1/2" CI750 box. This cabinet must be bolted to the right side of the VAX-11/750 CPU cabinet.

- A VAX-11/750 or VAX-11/751 can support only one CI750.

- A VAX-11/750 or VAX-11/751 can support a CI750 or a DR750, but not both.

- The CI750 will be supported by VAX/VMS Version 3.4. This initial release requires that a disk be locally attached through a MASSBUS, UDA50, or

other UNIBUS disk interface in order to boot VAX/VMS. Once VMS is booted, the VAX-11/750 becomes part of the cluster.

For this reason, we are offering the CI750 as an add-on device for use with packaged systems. A future release of VAX/VMS will eliminate the need for the local disk. At that time we will announce a CI-based VAX-11/750 System Building Block.

VAX-11/780 Configuration

- The CI780 requires one optional panel space in either the CPU cabinet or an SBI expander cabinet H9602-HA(HB).

ANNOUNCING VAX/VMS SUPPORT FOR THE HSC50

Ron Brown
DTN: 522-2251
CX01-1/P27 RCS:CSGV

The HSC50 is the I/O subsystem component of VAXcluster systems. It was announced on November 22, 1982. This article summarizes the key features of the HSC50. Your customers will find it a cost effective, high-performance I/O subsystem for both single and multi-processor VAX systems.

PRODUCT DESCRIPTION

The HSC50 is a Computer Interconnect (CI) based intelligent disk/tape server that offers full architectural support and performance optimization for DSA disks--RA80, RA81, and RA60--and tapes as they become available, particularly in multiple VAX CI configurations.

The first implementation of the VAXcluster architecture is based on the Computer Interconnect which supports up to 16 nodes. Each node may be a VAX-11/782, VAX-11/780, VAX-11/750 processor or an HSC50 controller. Digital is currently offering VAXcluster configurations in which each HSC50 controller can provide storage access for as many as four processors. Later, we will offer VAXclusters where each HSC50 will provide storage access to more processors.

Performance Optimizations

The HSC50s disk data channels are shared by up to four drives per channel. Each channel has realtime visibility into the exact sector position of each drive. With this information, the HSC50 dynamically allocates data transfer channels to the first drive to reach a designated sector. The result is significantly higher I/O throughput rates.

The HSC50 has a large pool of high-speed data buffers to accommodate data transfer bursts from concurrent disk and tape operations. It's completely independent of the need for realtime CPU memory access.

When multiple requests are queued for processing, the HSC50 "sees" into this request queue and organizes the most efficient seek sequence for each drive.

The HSC50 converts long transfer requests into a set of smaller fragments. It then uses its realtime visibility into the drive's rotational position to transfer data as soon as any fragment arrives under the diskhead.

Because the HSC50 provides parallel processing, many operations can be executed in parallel--simultaneous data transfers from different disks and/or tapes, seeking on some disks while reading from others, performing error recovery functions in conjunction with normal I/O services.

Data Integrity and Availability

To validate the integrity of stored data and provide for recovery in the event of a malfunction, the HSC50 uses not only generic mechanisms like parity, checksums, and autocorrelation patterns, but also special protection features:

- Before the execution of any disk transfer, the HSC50 reads four redundant copies of the sector header to ensure diskhead positioning.

- The HSC50 uses the industry's most powerful error correcton code to protect data. With it, the HSC50 can detect and correct up to eight independent error bursts, each up to ten bits in length, anywhere within a sector.

- Immediately upon receipt of user data, the HSC50 appends an error detection code that travels with the data to the disk surface. As the data is read, the HSC50 repeatedly checks the code for continuous verification.

- The HSC50 maps around media defects. If a sector is illegible, the server replaces it from a pool of good sectors. Thus the hosts always see "perfect" media; this is guaranteed through several levels of redundancy.

- All DSA drives are dual ported, so when they detect a "dead" controller on one port, they automatically switch to the alternate port. The HSC50 fully supports this fail-over feature; it declares to the host systems the specific drives to which it has access paths and dynamically reports to the hosts any drive that becomes accessible.

- The HSC50 internal architecture is designed to "firewall" resources so that failing components can be isolated and retired from service, while the rest of the subsystem continues operation.

- The HSC50 can diagnose many of its resources inline through periodic minidiagnostics--review critical subsystem elements, automatic removal of a malfunctioning device from service, and the running of inline diagnostics to pinpoint the problem.

- The Field Engineer can access the HSC50; take a specific resource, like a drive, temporarily out of service; and run an HSC50-based diagnostic--while the HSC50 continues I/O services to the hosts.

Maintainability

The HSC50 is designed for quick and easy maintenance. This design is implemented at several levels. Besides onboard microdiagnostics, the server has extensive loadable macrodiagnostics. Inserted into the subsystem via a TU58 cassette, they function within the HSC50 to test internal conditions.

Some diagnostics are automatic when the HSC50 is powered up, or when device malfunctions are encountered. Also, whenever special timers expire, diagnostic sanity checks automatically occur. When the HSC50 detects a failure, it reports it and removes the failing device from service.

More diagnostics can be run by operator request. Many can be run inline while the HSC50 is operating, while another even more comprehensive set can be run offline.

- The HSC50 uses airflow and thermal sensors to shut itself down when any failure(s) or environmental condition(s) could mean subsystem damage.

- Each HSC50 logic board has a set of LED's to indicate operational status.

- The front panel contains indicators that report both normal subsystem status and fault information.
- When the HSC5Ø is online, it reports errors it detects to the host error-logging facility.
- The HSC5Ø supports an ASCII serial line, so a Field Engineer can connect a maintenance terminal for direct communications with HSC5Ø software.

VAXcluster SYSTEM BUILDING BLOCKS: A MORE FLEXIBLE WAY OF CONFIGURING VAX SYSTEMS

Terry Retford
DTN: 247-2930
TWO/B02 RCS:TWSK

With the introduction of VAXclusters, we are taking the opportunity to respond to customers' needs for more flexibility in configuring systems. This new method will enable you to order systems which are tailored precisely to your customers' needs. The initial implementation is for VAXcluster systems. We plan to extend this approach to the complete VAX family in the near future.

We have chosen a method which we call System Building Blocks (SBBs). This approach consists of a number of kernel CPUs together with a number of menus from which the basic supported options are selected. Customers select the kernel which best suits their need, and through menu selections configure a fully supported system.

The menus contain only the option choices necessary to configure a supported system. Customer orders must include selections from each of the menus below:

- Load Device (for distributing VMS)
- System Device
- Console Terminal
- Communications Interface
- VMS Warranty Services

Additional add-on options are selected from the VAX Systems and Options Catalog in the normal manner.

VAX-11/780 SYSTEM BUILDING BLOCK FOR VAXcluster SYSTEMS

This building block contains a VAX 11/780 CPU configured with 2MB of 64K memory, a CI780 interface port, an HSC50 with a disk and tape controller, a Star Coupler, and a UNIBUS cabinet.

Note that the 780CI-AE and 780CI-AP SBB kernels that follow have 2MB of the new 64K RAM memory. Refer to the March 28, 1983 issue of Sales Update for details.

SBB KERNEL

MODEL NUMBER	DESCRIPTION
780CI-AE	VAX 11/780 CPU + 2MB OF MEMORY + UNIBUS CAB + CI780 + SC008 + HSC50 + HSC5X-B + HSC5X-C + 2 sets BNCIA-20 + VMS DZ LICENSE 120V 60Hz
780CI-AJ	SAME AS ABOVE EXCEPT 240V 50Hz

continued on next page

MANDATORY SELECTION OF 1 DEVICE FROM EACH OF THE FOLLOWING MENUS.

LOAD DEVICE MENU

MODEL NUMBER	DESCRIPTION
TEU78-AB/AD	TU78 TAPE DRIVE
RA60-CA/CD	RA60 DISK + CAB

SYSTEM DEVICE MENU

MODEL NUMBER	DESCRIPTION
RA60-AA	RA60 DISK
RA60-CA/CD*	RA60 + CAB
RA81-AA/AD	RA81 DISK
RA81-CA/CD*	RA81 + CAB
RA80-AA/AD	RA80 DISK
RA80-CA/CD*	RA80 CAB

SOFTWARE WARRANTY MENU**

MODEL NUMBER	DESCRIPTION
QE001-HM	VMS 9 TRACK MT DOCUMENTATION
QE001-HJ	VMS RA60 PACK DOCUMENTATION
QE001-AZ	INSTALLATION 90 DAYS WARRANTY AND 5 TRAINING CREDITS

TERMINAL MENU

MODEL NUMBER	DESCRIPTION
LA120-DA	CONSOLE

COMMUNICATION INTERFACE MENU

MODEL NUMBER	DESCRIPTION
DMF32-AB	8 LINE DMA CTRL
DZ11-A	8 LINE MUX
DZ32-A	8 LINE MUX

* Use this option if load device is a TEU78.

** Warranty required for the first VAX System purchased by a customer. The QE001-AZ option must be selected and one H Kit.

VAX-11/780 SYSTEM BUILDING BLOCK FOR VAXcluster UPGRADES

This System Building Block is used to add a VAX-11/780 CPU to a cluster which already has the components in 780CI-AE/AJ or 782CI-AE/AJ.

SBB KERNEL

MODEL NUMBER	DESCRIPTION
780CI-AP	VAX-11/780 CPU + 2MB OF MEMORY + UNIBUS CAB + CI780 + 1 set BNCIA-20 + VMS DZ LICENSE 120V 60Hz
780CI-AT	SAME AS ABOVE EXCEPT 240V 50Hz

MANDATORY SELECTION OF 1 OPTION FROM EACH MENU SHOWN BELOW

COMMUNICATION MENU

MODEL NUMBER	DESCRIPTION
DMF32-AB	8 LINE DMA CTRL
DZ11-A	8 LINE MUX
DZ32-A	8 LINE MUX

TERMINAL MENU

MODEL NUMBER	DESCRIPTION
LA120-DA	CONSOLE

OPTIONAL SYSTEM DEVICE MENU*

MODEL NUMBER	DESCRIPTION
RA60-AA	RA60 DISK
RA60-CA/CD	RA60 + CAB
RA81-AA/AD	RA81 DISK
RA81-CA/CD	RA81 + CAB
RA80-AA/AD	RA80 DISK
RA80-CA/CD	RA80 + CAB

- Note that the system device menu is optional, but the customer must have a Digital system disk (RA81/RA60/RA80) already on-site which will be allocated permanently to this CPU. VAX/VMS Versions 3.3 & 3.4 require that a system disk be owned permanently by each CPU within the cluster.

VAX-11/782 SYSTEM BUILDING BLOCK FOR VAXcluster SYSTEMS

This Building Block contains a VAX-11/782 Dual CPU configured with 4MB of 16K memory, a CI780 interface port, an HSC50 with a disk and tape controller, a Star Coupler, and a UNIBUS cabinet.

SBB KERNEL

MODEL NUMBER	DESCRIPTION
782CI-AE	VAX 11/782 DUAL CPU + LA120-DA + 4MB OF 16K MEMORY + UNIBUS CAB + CI780 + SC008 + HSC50 ● HSC5X-B + HSC5X-C + VMS DZ LICENSE 120V 60Hz + 2 SETS BNCIA-20
782CI-AJ	SAME AS ABOVE EXCEPT 240V 50Hz

MANDATORY SELECTION OF 1 DEVICE FROM EACH OF THE FOLLOWING MENUS.

LOAD DEVICE MENU

MODEL NUMBER	DESCRIPTION
TEU78-AB/AD	TU78 TAPE DRIVE
RA60-CA/CD	RA60 DISK & CAB

SYSTEM DEVICE MENU

MODEL NUMBER	DESCRIPTION
RA60-AA	RA60 DISK
RA60-CA/CD*	RA60 + CAB
RA81-AA/AD	RA81 DISK
RA81-CA/CD*	RA81 + CAB
RA80-AA/AD	RA80 DISK
RA80-CA/CD*	RA80 + CAB

* Use this option if load device is a TEU78.

** Warranty required for the first VAX system purchased by customer. The QE001-AZ and one H Kit must be selected.

SOFTWARE WARRANTY MENU**

MODEL NUMBER	DESCRIPTION
QE001-HM	VMS 9 TRACK MT DOCUMENTATION
QE001-HJ	VMS RA60 PACK DOCUMENTATION
QE001-AZ	INSTALLATION 90 DAYS WARRANTY AND 5 TRAINING CREDITS

TERMINAL MENU

MODEL NUMBER	DESCRIPTION
LA120-DA	CONSOLE

COMMUNICATIONS INTERFACE MENU

MODEL NUMBER	DESCRIPTION
DMF32-AB	8 LINE DMA CTRL
DZ11-A	8 LINE MUX
DZ32-A	8 LINE MUX

VAX-11/782 SYSTEM BUILDING BLOCK FOR VAXcluster UPGRADES

This system is used to add a VAX-11/782 CPU to a Cluster which already has the components in 782CI-AE/AJ or 780CI-AE/AJ above.

SBB KERNEL

MODEL NUMBER	DESCRIPTION
782CI-AP	VAX-11/782 DUAL CPU + 4MB OF 16K MEMORY + UNIBUS CAB + CI780 + VMS DZ LICENSE 120V 60 Hz
782CI-AT	AS ABOVE EXCEPT 240V 50Hz

MANDATORY SELECTION OF 1 OPTION FROM EACH MENU SHOWN BELOW

COMMUNICATION MENU

MODEL NUMBER	DESCRIPTION
DMF32-AB	8 LINE DMA CTRL
DZ-11	8 LINE MUX
DZ32-A	8 LINE MUX

TERMINAL MENU

MODEL NUMBER	DESCRIPTION
LA120-DA	CONSOLE

OPTIONAL SYSTEM DEVICE MENU*

MODEL NUMBER	DESCRIPTION
RA60-AA	RA60 DISK
RA60-CA/CD	RA60 + CAB
RA81-AA/AD	RA81 DISK
RA81-CA/CD	RA81 + CAB
RA80-AA/AD	RA80 DISK
RA80-CA/CD	RA80 + CAB

* Note that the system device menu is optional, but the customer must have a Digital system disk (RA81/RA60/RA80) already on-site which will be allocated permanently to this CPU. VAX/VMS Version 3.3 & 3.4 require that a system disk be owned permanently by each CPU within the cluster.

DIGITAL STANDARD PRICE LIST

The DSPL has the above breakdown in the preface to the Systems section which contains the SBB Kernels 780CI-AE/AJ/AP/AT and 782CI-AE/AJ/AP/AT. An example of a configuration order is shown below.

Assume a customer requires a VAXcluster system with two VAX-11/780s with 4MB of memory and 16 terminal ports on each, one tape, and at least 1.3 GigaBytes of disk storage.

QTY	PART NO.	DESCRIPTION
1	780CI-AE	VAX-11/780 CPU, 2MB, HSC50, 4 tape and disk channels, UNIBUS Cabinet.
1	TEU78-AB	Magnetic tape plus interface to MASSBUS.
1	RA81-CA	RA81 456MB disk and cabinet.
1	QE001-HM	Documentation and 9 track magtape.
1	QE001-AZ	Installation, warranty and training credits.
1	LA120-DA	Console terminal.
1	DMF32-AB	Eight lines for terminal communications.
		(The above completes selection from the mandatory menus; the following disk is optional.)
2	RA81-AA	912MB of RA81 disk mounts in RA81-CA cabinet.
		(Add the second processor from the upgrade building block.)
1	780CI-AP	Second VAX-11/780 CPU with 2MB plus UNIBUS Cabinet.
1	LA120-DA	Console terminal.
1	DMF32-AB	Eight lines for terminal communications.
		(This completes mandatory menu selection. The following items are optional.)
1	MS780-FB	4MB of 64K memory. 2MB for each processor.
2	DMF32-AB	Sixteen lines for terminal communications.

The customer could add other items such as terminals or other devices according to the guidelines in the VAX System and Options Catalog.

If you have any problems configuring VAXcluster systems with building blocks, please contact your technical support specialist.

VAXclusters SELLING

A P P L I C A T I O N M A R K E T G R O U P S

VAXcluster SYSTEMS SELLING TIPS

Dave Walker
DTN: 264-4390
MK02-1/B2

VAXclusters will have an appeal to today's VAX customer and we urge you to take the VAXcluster message to your customers. Also, VAXclusters give you the opportunity to sell in new application environments and to new customers as well.

With this announcement, Digital introduces the most flexible, cost-effective way for our customers to buy and grow computer systems. Our customers can now start with the industry's most interactive, efficient, general-purpose system -- VAX, and continue to grow and tailor this basic system to respond to the application needs and continual growth of the user community. The modularity and continuity of the VAX/VMS product-offering will provide the ability to manage the system without loss of investment in existing programs or need for reprogramming, retraining, or replanning. The management of a VAXcluster will require much less operations overhead than multiple independent systems.

Benefits Summary

The following is a summary of the key benefits provided by the VAXcluster system.

Common Information Access - an expandable multiprocessor system that appears to be a single system to an expanding user community and provides easy responsive access to the shared information stored throughout the system.

Modular Growth - The system can grow from a single system and meet the demands of mixed and new applications or the continual addition of new users. Processors, memory, storage devices, and terminals can be added without interruption or deterioration of the integrity of the system and its existing programs.

Processors within the Cluster can be dedicated to specific applications or sized for a specific number of users which provides system management flexibility.

Higher Availability - The hardware redundancy features enable systems to be configured without a single point of failure. Sophisticated system software protects the integrity of the information, and provides a tunable, high-uptime system. The productivity of a large user base can be enhanced through the continuous access and availability of a VAXcluster System.

Increased Capacity - VAXcluster Systems provide for increased capacity with up to 10 Gigabytes of disk storage on each HSC50. Configurations with over 100 Gigabytes of disk storage are possible for customers with large mass-storage requirements. Each VAX-11/780 can now be expanded up to 32 megabytes of main memory with the new 64K RAM memory. This will allow a Cluster configuration with large amounts of main memory and compute power which is many, many times that of a single VAX-11/780.

Preserved Investment - Your customer's current investment in VAX-11/750, VAX-11/780, or VAX-11/782 CPU and MASSBUS and UNIBUS disk and tape technology is preserved. All of these can be configured into a VAXcluster System. A new customer who purchases a system can see the upward growth potential for extending the limits of any single CPU.

Lower Costs - It should also be noted that cost to customers associated with configuring a multi-CPU cluster is substantially lower than the cost of configuring an equivalent number of independent standalone systems. There are significant savings for your customer in configuring the actual number of disks and tapes required for a Cluster. This is especially true for tapes, because a tape can be shared by the entire Cluster.

The following matrix identifies the needs of prospects and presents a VAXcluster selling strategy.

NEW VAX PROSPECT

Need	Sell Today's Deliverables	Sell VAXcluster Program
New System	VAX and VAXcluster Systems, VAX/VMS, Layered Products, DNA, DSA	VAXcluster and VAX Family Commitment
Manage Multiple Management Centers	VAXcluster hardware & VAX/VMS as the production standard	Modularity of processors, but sharing of data resources
Higher System Availability	VAXcluster Topology & redundant data paths & VAX/VMS V3.3 and V3.4	VAXcluster Topology & future VAX/VMS releases
Mainframe Alternative	VAXcluster Systems and leadership in multi-processing and networking	Commitment to VAX/VMS, hardware compatibility, and continued leadership in VAXcluster components (e.g. CPU, mass storage, and controllers)

INSTALLED VAX CUSTOMER

Need	Sell Today's Deliverables	Sell VAXcluster Program
More Capacity	More Memory-32MB on VAX-11/780 More Mass Storage-10G/HSC VAX-11/782* VAXcluster Systems*	VAXcluster and VAX Family Commitment
Access to Common Data	VAXcluster Systems and HSC50 features	VAXcluster and Disk Storage Strategies
Higher System Availability	VAXcluster Topology and VAX/VMS V3.3 & V3.4	VAXcluster Topology and future VAX/VMS Releases
Manage Shared Resources or Multiple Management Centers	VAXcluster hardware and VAX/VMS as the Production Standard	Modularity of Processors but sharing of data resources

* The decision to sell the tightly coupled VAX-11/782 solution or the VAXcluster solution should be based on the applicability of the attached processor to provide a solution to the near-term problem. The workload can

be evaluated with the QUALIFY software tool from Digital that will indicate the performance improvement possible with the VAX-11/782. The budget available for upgrading is also a significant factor in the decision.

The VAX-11/782 solution does not preclude VAXcluster implementation at a later date, because the VAX-11/782 is a supportable host on the CI.

VAXcluster APPLICATION SCENARIOS

The following section highlights several applications based on information provided by the following individuals:

Technical OEM	Gillian Scholes	DTN 225-6710
Advanced Technology Company	Michael T. Peterson	DTN 231-7118
National Service Corporation	Dave Chen	DTN 264-2935
Agricultural Combine	Martyn Lewis	DTN 633-2210
Bank	Gary Eckroth	DTN 264-3804
National Energy Labs	Gail Taplin	DTN 231-5951
University	Bill Clark	DTN 231-5617

The information presented in the application scenarios is representative of typical situations in which VAXclusters are being considered by our customers. The "customer" names, however, are fictitious. Questions about any of the application scenarios should be directed to the individual contributors.

To facilitate the reading of these scenarios, they have been organized according to the following format:

Introduction:	What exists today
Current:	The existing system
Need:	Customer requirements
Description:	The VAXcluster System
Benefits:	Customer view
Conclusion:	Summary

THE TECHNICAL OEM MARKET

Introduction

Modular expandibility for incremental system power, database resource sharing, and higher availability are the key attractions of VAXcluster Systems for the Technical OEM marketplace.

This extension of the VAX/VMS Architecture into the VAXcluster offering will be immediately recognized by Digital's OEM customers as a major opportunity to offer the full benefits of VAX computers in advanced multiprocessor configurations. The flexibility of building large systems in smaller increments, the cost-effectiveness of the HSC50 controller, high speed interprocessor communications via a new 70Mb Computer Interconnect, plus the new extensions being added to the VAX/VMS Operating System will attract a considerable amount of interest among OEMs who are not now using VAX.

Data Reduction and Analysis Applications

Introduction

VAXcluster Systems are of particular interest in OEM applications where large amounts of data are to be reduced and analyzed. For example, the oil industry depends upon seismic exploration in their quest for new sources for oil and minerals. Land and marine crews actually measure the time it takes for seismic wave signals to travel from a known surface point to one or more reflection points and back again. Highly sophisticated controllers are used to collect the data, recording it on streamer tapes which are then delivered to seismic data processing centers for initial reduction and analysis.

Current

Seismic data processing is a massive computational exercise. In the past it has been accomplished on large mainframes using batch processing techniques. Increasingly, however, customers are demanding higher performance 32-bit processors enhanced by array processors. As a preliminary step, the data is first "demultiplexed" through a matrix inversion process. Working with geophysicists, seismic analysts apply computerized data synthesis and correlation techniques to produce seismic sections using electrostatic or photographic plotters. The initial data may be processed several times before a final interpretation is made.

Need

Typically, the OEM will be looking for one or more 32-bit processors with the performance capability of a VAX-11/780. The precise need is:

- fast computational ability
- high input/output throughput
- large amounts of storage and memory
- access to information that must be shared
- the means to expand a system in modular increments
- higher availability for increased system uptime

Description

The VAXcluster approach using VAX-11/780s for computational units and HSC50 based mass-storage subsystems represents an ideal system for seismic data processing. Many processors and shared files operate as a single entity.

Benefits

The OEM can have up to 10 gigabytes of mass storage for each controller and up to 32MB of main memory on each VAX-11/780.

Not only can a VAXcluster System meet the large data requirements of the application but it can also handle the data quickly through the high speed computer interconnect. The interconnect is a dual path link which operates at a maximum data transfer rate of 70Mb/second over each path. The dual path is designed to provide automatic "failover" to the surviving path in the event of failure of a single path.

New features in VAXcluster Operating System Software include the VMS Lock Manager which allows either systems or user processes to have the ability to share resources within the cluster. This will increase productivity for the user and contribute to the reduction of overall system costs.

Control and Monitoring Applications

Introduction

VAXcluster Systems also open up new opportunities for OEMs who need higher availability systems, particularly in the monitoring of continuous processes. One example would be the OEM who is designing a system for a hydro-electric project. The supervisory control system must control the huge turbines. It must also monitor the overall operation of the hydro-electric complex at thousands of remote points, and then display current information on wall sized maps. It must supply information interactively to terminals in several control centers as well as to the headquarters of the operation which may be hundreds of miles away. It may also have to exchange information with other computers that are controlling electrical grid systems.

Current

In the past a pair of VAX-11/780s, linked by a DMC11 controller, would form the basis for a high availability system. One processor would act as a "hot standby" for the other to ensure the continuous operation of the system. Typically, DECnet communications software would link the VAX-11/780s to other smaller systems, and a variety of telecommunications or "special" interfaces would support the Control Center terminals and a wall size mimic display.

Need

The most important considerations in the design of control and monitoring systems are:

- reliability of the system
- high throughput
- large address space for handling database subsystems
- high I/O capacity to attach and make efficient use of perhipherals attached to the system

- communications capabilities to link individual processors to other portions of the "control network"
- an extensive library of software packages to support various types of applications

Description

The OEM previously would have had to design around the inherent limitations of a network based system in order to achieve the speed and recovery features necessary for a system like this.

Now, with the VAXcluster product set, the OEM is able to use the CI780 to interconnect the two VAX-11/780s. The interface between the VAX processors and the Computer Interconnect (CI) is an intelligent port controller which performs error checking, arbitrates bus contention, provides path failover and optimizes communication. An HSC50 provides access to as many as 24 disk drives which can be dual ported to provide an alternate path for accessing critical data in the event of failure in one of the processors or in the path itself.

The HSC50 will also support volume shadowing in future software releases of VAX/VMS in order to protect data from corruption or a failure in the disk drive itself. Future versions of VAX/VMS will provide the OEM with the necessary tools for securing the integrity and availability of critical data or processes--software tools such as the Common Journaling Facility, Recovery Units Facility and the Checkpoint Facility--discussed in the software section of this Sales Update, Special Edition.

Benefits

VAX processors are widely used today by OEMs for many control and monitoring applications throughout a variety of industries. With the introduction of VAXcluster Systems, Digital is able to supply more flexible tools to provide the OEM with a means of developing unique solutions for specific markets. The VAXcluster approach to computing allows an OEM to focus a particular expertise on the overall project design.

Conclusion

VAXcluster Systems are expected to generate new business in the Technical OEM marketplace. OEMs in other areas such as process control, communications control and message switching, material or inventory management, and medical instrumentation will be attracted by the VAXcluster approach to building more powerful, higher capacity and more reliable systems.

Digital has a unique offering among its competitors--one that outclasses all other minicomputer system vendors and allows us to compete more effectively for applications that were once captive to mainframe vendors.

Our OEMs service a multitude of market areas that will benefit from the capabilities offered by VAXcluster Systems. Modular expandibility has a universal appeal. Resource sharing and higher availability meet more specific needs.

ADVANCED TECHNOLOGY COMPANY

Introduction

Industry research departments and large government research laboratories such as Advanced Technology Company (ATC), are basing their overall computational procurements around an interconnection strategy that incorporates the VAXclusters concept in expanding the processing and information handling capabilities at each node in their distributed network.

Current

At ATC, a network allow scientists and engineers to conduct a wide variety of experiments and simulations at different locations throughout the research campus.

Need

In these very sophisticated environments performance, high availability and instantaneous access to large quantities of information are critical to the success of individual researchers and to the mission of ATC. The application of a loosely coupled, multiprocessing system with a distributed file structure is an ideal solution for their needs.

Your customers can also enjoy the benefits of a distributed file structure with high availability features, increased capacity and performance to a much greater extent than can be offered by any other single vendor.

Description

Digital's concept of loosely coupled VAX processors, interconnected via a high speed 70Mb bus is the next significant step in the evolution of computer systems on the ATC network.

There are various applications supported by the Advanced Technology Company network, and VAXcluster Systems are important to each.

One application is dedicated to acquiring real-time data from an ocean environment facility. The data from these experiments must be instantly and reliably available to the investigators for their research. It is the availability of a transparent, high speed interconnect between processors in a VAXcluster System that offers the performance and high availability that researchers require.

Another application serves an Engineering Division that relies heavily on CAD support. This application often has to be executed on dedicated computers that cannot afford to be slowed down by processing overhead. Following a presentation of the VAXcluster concept by Digital, the Engineering Division decided to fund a major portion of the procurement when it became clear that this approach would enable Engineering to substantially increase the network's responsiveness, even in the face of increasing workloads.

Still another application serves a Simulation Group with a special set of applications needs. The Numerical Oceanographic Simulator Project is funding the major portion of this procurement which involves up to ten or more large VAX Processors interconnected and front-ending to an Amdahl Supercomputer. Only a VAXcluster System offers the availability, performance, and capacity to

conduct the kind of extensive, and computationally intensive applications their needs demand for this application.

Benefits

Overall, ATC is one of the most extensive, sophisticated, and functional networks to be found in a research environment. What appeals to the users of this network is the existence of a transparent, high speed interconnect between VAX processors. This provides the level of performance the users on the network demand. It also provides the high availability their research needs require.

Both of these attributes led ATC to develop their long-term computing strategies around an architecture which encompassed the VAXcluster concept.

Conclusion

It is a fact that, in major research laboratories, capacity, high availability and performance (all featured in the VAXcluster product set) are extremely important. This is why research oriented customers are among the largest buyers of Digital's processing and distributed systems products. Indeed, some of Digital's largest distributed processing systems are found in research laboratories such as Advanced Technology Company.

All laboratories, large as well as small, have the same basic requirements. The difference is only quantitive. One of the fastest growing and most lucrative markets for these exciting products is, and will continue to be the research environment.

NATIONAL SERVICE CORPORATION

Introduction

The National Service Corporation (NSC) recently announced a proprietary (internal) Network Service Offering which reflects the recent advances in telecommunications and data processing technology. As a result of government deregulation, the data processing and communications industries are converging as they evolve applications for a new era in information processing. NSC is among the industry leaders in exploiting this convergence to better service its internal network requirements.

Current

Multiple VAX-11/780 systems are currently operating under Digital's VMS Operating System to provide a Network Service Offering for NSC. This service, not yet fully implemented, already provides NSC users with programmable, distributed processing capabilities that allow communications between computing equipment provided by different vendors.

Need

Prior to the installation of their Network Service Offering, NSC computer users encountered several major problems.

- single system applications incapable of communicating with other applications
- inflexible applications that were unable to adapt to changing business environments
- difficulty in network management
- high start-up costs associated with development and implementation of information systems.

Some of these problems continue to persist in their current environment.

Description

Using an extended concept of distributed processing, however, NSC is continuing to evolve the Network Service Offering with the following design goals:

- installation at all major access/egress points in the corporation
- superiority in price and performance of the central processing unit(s)
- the ability to support customer programming options with a greater variety of languages than before
- a stable environment so that Divisional users will be able to protect their programming investment
- higher availability of network service
- capability to add processing capacity and service functionality as needed

to maintain a responsive Network Service Offering within the company

- a secure environment for their users' data

Benefits

With these goals in mind, NSC has chosen Digital's VAXcluster approach for their Network Service Offering. The VAXcluster product set meets their internal needs by providing upward migration compatibility with VMS, with ever improving price/performance ratios in the system configuration.

The VAX/VMS Architecture offers the most stable environment in the industry so that NSC is able to protect its investment in software and user training.

Also, through the features inherent in the VAXcluster approach, National Service Corporation is able to achieve the reliability and higher availability of service they must provide to the largest clients in various Divisions.

Conclusion

The complex applications software required to solve their internal users needs was introduced without the need to develop unique systems hardware due to the strength and stability of Digital products.

VAXcluster Systems offer the solution to National Service Corporation. Growth. Availability. Stability.

to maintain a responsive Network Service Offering within the company

- a secure environment for their users' data

Benefits

With these goals in mind, NSC has chosen Digital's VAXcluster approach for their Network Service Offering. The VAXcluster product set meets their internal needs by providing upward migration compatibility with VMS, with ever improving price/performance ratios in the system configuration.

The VAX/VMS Architecture offers the most stable environment in the industry so that NSC is able to protect its investment in software and user training.

Also, through the features inherent in the VAXcluster approach, National Service Corporation is able to achieve the reliability and higher availability of service they must provide to the largest clients in various Divisions.

Conclusion

The complex applications software required to solve their internal users needs was introduced without the need to develop unique systems hardware due to the strength and stability of Digital products.

VAXcluster Systems offer the solution to National Service Corporation. Growth. Availability. Stability.

AGRICULTURAL COMBINE INCORPORATED

Introduction

Initially, Agricultural Combine Inc. (ACI) chose Digital Equipment Corporation. to provide an architectural basis for implementing their networking strategy. Today, this same customer is equally enthusiastic about our VAXcluster offering.

Current

ACI provides researchers across the United States with a variety of computing facilities, many of which are located in remote areas. In addition to their network of interconnected computing facilities, researchers have access to more powerful computational systems comprised of multiple VAX-11/780 processors within the headquarters complex. Applications presently include data collection, data reduction, statistical analysis and information management using Digital's ALL-IN-ONE and VAX-11 BASIC.

Need

In considering their expansion plans, Agricultural Combine has placed special emphasis on the need for higher availability, larger mass-storage devices and common file structures. Also, there is an increasing pressure from within to bring about topological changes in their systems without major re-structuring and re-systematizing.

Description

Included in the customer's plans is the phased replacement of older Digital systems with VAX-11/730 and VAX-11/750 systems to give them the benefits of the VAX/VMS Family of products and the VMS Operating System in particular.

Included in ACI's strategy is the plan to build loosely coupled multiprocessing systems using the VAXcluster product set.

Benefits

Digital's VAXcluster offering provides a wide range of computing facilities to the scientists working at Agricultural Combine. It further demonstrates our strategy of delivering a broad range of solutions across a range of applications. And it underscores our ability to support much more than a single application or a departmental system.

Conclusion

This customer's appreciation of the VAXcluster approach to expanding the computing power of a single system through the use of multiple systems, operated as though they were one, has become a major factor in their strategic plan for future computer applications within their organization. Digital's VAXcluster product set is an integral part of their strategy as they plan for the future.

BANK

Introduction

The requirements of todays banks extend far beyond on-line teller systems to a wide range of computing facilities that enable information to be shared both within and outside the boundaries of their existing organizations.

Current

A large bank in New York City has installed ALL-IN-ONE with DECmail electronic mail. The system selected for the initial pilot phase is a VAX-11/780 with an HSC50 intelligent disk controller.

Need

Frequent and often unnecessary meetings, delays in reaching people by telephone, and numerous interoffice memoranda were beginning to cut into the managerial productivity in the bank. Time was being wasted.

Description

The use of electronic mail eliminated some of the meetings, but more importantly the bank began to cut down on the number of missed telephone calls and handwritten messages. Dramatic improvements in internal communication helped to justify additional investment in office automation capabilities.

The second of a three phase approach to office automation in the bank has been developed. Other office functions are to be made available such as work processing, calendar management and the means to tie into the corporate data processing center. A VAXcluster System will be utilized to support these applications.

A second VAX-11/780 and additional storage will be added to provide the additional computational and storage facilities required by the application. Digital's SNA Gateway will also be added to support communications between Digital's VAXcluster Systems and the mainframe computer facilities at the corporate data center.

The third phase of the bank's plan will extend the accessibility of the VAXcluster System corporate wide. Additional VAX processors and disk storage facilities will be added as required.

Benefits

While it is difficult to place a dollar figure on improved productivity in the bank, the improvements in the bank's internal communications have helped them to justify their investment in office automation. Perhaps the most important criterion of all has been the ease of use associated with Digital's systems. It has been very easy for managers and data processing professionals alike to learn how to use Digital's office products.

Furthermore, the modular growth provided by the VAXcluster System has made it possible to make the necessary transitions. No hardware has been replaced. Only additional components have been added. The potential for extended system downtime will be eliminated since files will be duplicated through the use of volume shadowing capabilities offered by the VAX/VMS software extensions.

Conclusion

The VAXcluster approach will increasingly become an integral part of the bank. Electronic mail is a critical application and has been favorably received by the bank's professionals. It is anticipated that Digital's additional office automation products will be equally well received as the bank extends its computational facilities using the VAXcluster approach to building its system.

NATIONAL ENERGY LABORATORY

Introduction

The world's demand for energy is continually increasing. As researchers realize the limit of the earth's resources (fossil fuels such as gas and oil), they look for alternative approaches to develop safe and plentiful sources of energy, sources that will also be safe for the environment. At National Energy Labs (NEL), and other energy laboratories across the country, the researchers are studying ways to make fusion energy practical and efficient.

Current

With over 100 Digital Equipment Corporation computers at the NEL facility, the VAXcluster concept of loosely coupled multiprocessor systems is of great interest to their efforts at solving the world's energy puzzle.

Need

The paramount need at the Fusion Energy Division at NEL is to be able to share large amounts of common data, at high speeds. The capability and capacity of a computer system for one of their experiments is their biggest concern. This is closely followed by the system's performance and then availability.

Description

Originally, NEL envisioned a tightly coupled system of four VAX-11/780's sharing memory. When partially implemented, the scope and technology of these experiments changed, making it apparent that this type of configuration would not handle the enormous growth in data handling which they expect to encounter over the next few years.

A VAXcluster System is now being considered to handle the increased CPU power and to enable massive amounts of storage to be added as the project grows.

Benefits

The VAXcluster approach allows NEL to take advantage of new VAX processors as they are introduced, i.e. they can be incorporated into a system without disrupting their experiments or obsoleting their investment in Digital computers.

Their need for high speed data acquisition, common file access and massive storage requirements is enhanced through the VAXcluster product set and offers a new dimension of capability to an already impressive network of distributed VAX processors by offering the capability for more users to access a common file.

Conclusion

The potential demand for VAXcluster Systems will be great at all laboratory facilities. Most, in fact, already have two or more VAXs. Grouping VAX processors in loosely coupled configurations with intelligent mass-storage subsystems and high speed computer interconnection will prove to be very attractive to NEL's research needs. It will also be attractive to other energy laboratories and to laboratories in all scientific communities.

UNIVERSITY

Introduction

A large eastern university in the United States is facing a computer crisis. The current computer center resources are inadequate to meet the demands of the various departments in the university. Normally, it is impossible to satisfy this type of demand on short notice.

Current

Hundreds of students from the Computer Science, Engineering and Business Departments are receiving computer instruction on Digital VAX computer systems. They are learning how to use text editors, a wide variety of computer languages, and working with a range of educational applications.

Need

The university is now planning to integrate computer training into other areas for the next semester. This will involve the Physical and Social Sciences Departments, the Medical School and the School of Education. This means there will be hundreds of new student users with a need for substantially more computer power and information storage capacity to accommodate their demands.

Description

Digital VAXcluster Systems provide the immediate answer this institution is seeking. It provides an architecture that accommodates change in existing systems without disrupting system operations. The additional computing and storage can be added to the system without the trauma of most system upgrades or expansions.

A VAXcluster System will be installed at this university through the modular addition of additional VAX systems and the required storage. Any file, including compiler files, can be shared by all the processors in the VAXcluster. Although the system is comprised of multiple processors, VMS software extensions will enable these resources to operate as a single entity. The VAXcluster system operates as a single system.

Benefits

The VAXcluster System approach will enable this institution of higher education to meet all its needs for the foreseeable future and beyond. The customer need not buy unused capacity -- only what is presently needed. Best of all, there are no costly conversions in relation to file conversions, program recompilations or modification of operating systems.

Conclusion

VAXcluster Systems represent an up to date and very cost effective solution for the rapidly changing computing requirements in today's colleges and universities.

COMPETITION

V A X B A S E P R O D U C T M A R K E T I N G

VAXcluster SYSTEMS: COMPETITION

Dave Chanoux
DTN: 247-2580
TWO/B02 RCS:TWSK

The March 14 issue of Competitive Update categorized VAX competition and identified leading competitors in each category.

General Purpose	IBM
Workstation	Hewlett Packard
High Availability	Tandem
High Performance	System Engineering Labs

Throughout this Sales Update special issue you have read that VAXcluster systems take VAX to new applications in high availability systems and high performance general purpose systems. This article positions VAXcluster systems within the framework developed in Competitive Update and presents product strengths and weaknesses.

VAXcluster systems extend VAX against competitors in two categories; general purpose systems and high availability systems.

General Purpose Systems

VAXcluster systems compete against large systems in the general purpose category. Over the past several years, we have positioned the VAX-11/780 as a direct competitor to mid and large IBM 4300 series products. Competition against larger IBM mainframe products (303X, 308X) has been through distributed networks of VAX. We have competed in our approach to computing, information management, and problem solving by positioning VAX as an alternative to mainframes. VAXcluster systems provide growth to performance levels of 303X and 308X while retaining the characteristics which made it an attractive alternative to these products.

The Selling Tips article in this issue identifies customer needs where VAXclusters excel. To each application, the VAXcluster brings a unique blend of benefits from distributed processing and centralized mainframes. These are strengths which are our key to success in the new VAX range.

VAXcluster Features	Benefits	Usually Found
1. Large database capacity; Cost-effective mass storage; Rich information management products	Supports large applications; High customer value; Easy and fast implementation; No need to duplicate data	Mainframe
2. High compute performance	Supports large applications; Good price/performance metric	Mainframe
3. Multiple processors in a	Clusters mirror	Distributed

cluster can be allocated to applications, departments, users, or functions	organizations.	System
4. Many application types supported	Select the best hardware and software combination at each cluster node for cost effective performance.	Distributed System
5. Large applications can be segmented to multiple processors	Simultaneous processing achieves higher performance	Distributed System

Features 1 and 2 allow you to compete in the mainframe environment. Features 3, 4, and 5 are distributed computing characteristics which allow you to win with VAXclusters. To be successful, understand your customers' application and sell against large IBM mainframe systems where VAXclusters have unique benefits.

High Availability Systems

VAXcluster systems change our competitive position with Tandem Computers. The following five points draw a comparison between VAXcluster systems and the Tandem approach. They highlight product strengths and weaknesses.

1. Configuration Architecture

Tandem's approach to achieve high-availability systems is through multiple paths to system resources. In the event of a failure, there is an alternate path to a critical system resource. Critical system resources are always configured in pairs. Tandem's architecture achieves high availability through dual paths using a 2n redundancy. VAXclusters achieve the same system availability and better efficiency through dual paths using an n + 1 redundancy. (n is the number of a particular system resource required.) More system resources means more "excess baggage" representing standby resources in a Tandem system and more VAXcluster advantage.

2. Processor Performance

VAX characteristics including 32 bit system architecture, a range of processors, expansion up to 32 MB of main memory, and up to sixteen nodes in a VAXcluster system will give VAX a clear product advantage in this category. Tandem processors use a 16 bit architecture, are in the 11/34-11/70 performance range, support up to 2 MB of main memory, and are configured in pairs with up to eight pairs in a Tandem system. These product comparisons let you emphasize ease of use, ease of implementation, systems matched to customers' needs, and superior performance.

3. Multi-Access Peripherals

VAXcluster disk and tape controllers are shared by all processors, while the Tandem system is constructed with dual ported controllers attached to processor pairs. Other Tandem processors (not one of the pair) can access information, but information requests have to be processed through the owner of the controller. VAXcluster systems do it better. If disks and tapes are attached to the HSC50, every processor in the cluster has direct access to disks and tapes for data requests.

Our RA disk family is a competitive advantage in capacity and cost per megabyte. To improve our competitive advantage, configure as much disk storage as possible on systems. Point out to prospects the direct access to disk from every processor in the cluster.

4. Terminal Controllers

Tandem terminals are attached through dual ported multiplexers and are dedicated to a processor pair. VAXcluster terminals are attached through any VAX supported communications device either directly as through a DZ11 or DMF32, through another system in a DECnet network or through the Ethernet attachments. Terminals can be dedicated to processors or switched between processors using bus switches (DT07) at the processors or line switches (CS11) at the terminals provided by CSS. Digital's distributed system capability provides customers with a great deal of flexibility in accessing VAXcluster systems.

5. Operating System

The Tandem Guardian operating system has been designed specifically for transaction processing. In that application, Guardian gives Tandem a product advantage. Digital's VMS is a general purpose operating system combining interactive time sharing, computation, and batch processing. In the multi-application environment, the product advantage is with VAXcluster systems.

The following table summarizes the product advantages for VAXcluster systems and Tandem systems.

VAXcluster ADVANTAGES	TANDEM ADVANTAGES
General purpose system	Transaction processing operating system
Growth matched to performance needs	Non-stop operations
Mixed performance processors on same communication path	
High performance as well as high availability	
Redundancy is across cluster	
Gives shared files to all CPUs	
Files always available to any user	

The end result: VAXclusters have a competitive advantage in several, but not all application and customer situations. To be successful in selling VAXclusters, understand your customer's application using the product comparisons described above. Direct your selling activities to points where VAXclusters have product advantage.

SOFTWARE SERVICES FOR VAXcluster SYSTEMS

Mo Bakr
DTN: 276-8062
OG01-2/V08 RCS:OGOX

Software Services recognizes the opportunities of VAXcluster and plans to provide customers with a full range of services to enable them to take full advantage of the added capability.

As new functionality is introduced to the marketplace, starting with VAX/VMS V3.3/3.4, corresponding software services will be introduced to provide the customer with a choice of service offerings that meet their needs.

In support of the current offering (VAX/VMS V3.3/3.4) we will continue to offer cost-effective Digital solutions to answer the customer's life-cycle software support needs. These include:

- Pre-Sale Assistance
- Software Installation
- Warranty Support
- Professional Services
- On-Going Post Warranty Service Options (Self-Maintenance, Basic Service, and DECsupport)

Marketing Guidelines for Service

In support of the System Building Block concept, we have created the QE001-AZ option priced at $7360 for the VAX-11/780 and VAX-11/782 clustered building blocks. This option, which is required for the customer's initial VAX/VMS system, provides installation services, 90 Day Warranty and Training Credits per CPU. Combined with the appropriate media and documentation kit (QE001-HM,HJ), the customer will have a fully supported system for the 90 day warranty period. Note that with the Building Block concept, the VAX/VMS DZ license is included with each CPU.

The customer's initial system, which must be supported, can be a current A Kit (for non-building block systems) or a combination of DZ, AZ and H Kit.

Subsequent systems require a minimum of a right-to-copy license (DZ) for each system. Other offerings that are available for all or some of the systems are:

- Fully supported A-Kits
- AZ Kits which provide the full range of non-media service (Warranty, Installation, Training Credits)
- H-Kits which contain the appropriate type of media and documentation for each system
- Add-on installation
- Combination of the above

On-Going Service Offerings:

VAX/VMS can be covered for each system under one of the standard offerings: Self-Maintenance, Basic or DECsupport.

For VAXcluster Systems, where each CPU retains its VMS copy, the minimum recommended support package is a full service coverage of SMS, Basic or DECsupport on the first system and the right-to-copy (HZ service option) on the remaining systems.

The customer should also be encouraged to purchase additional telephone support contacts (TSC) to support the large number of users on the cluster.

At the current time, customers will purchase layered products under existing licensing guidelines, as specified in each SPD.

It is important to understand that cluster functionalities will be introduced in a phased approach. Software Services will coordinate the appropriate service announcement with each phase.

It is also important to stress the degree to which cluster technology will underline and enhance the role that software plays in a customer system solution in the pre-sale phase, in the start-up phase, and in the steady-state operation phase. Software Services should be involved early in the pre-sales cycle to help the customer find not only the appropriate solution to his present problem, but a solution today that can grow with the changing needs of tomorrow.

SOFTWARE SERVICES SUPPORT IN A 3 VAX CLUSTER

Initial State (Warranty Period):

Option	VAX #1	VAX#2	VAX #3
VAX/VMS Option 1	A-Kit or Equivalent (DZ, AZ, H-kit)	A-Kit or Equivalent	A-Kit or Equivalent
VAX/VMS Option 2	A-Kit or equivalent	DZ+H-Kit + A/O installation	DZ
VAX/VMS Minimum	A-Kit or Equivalent	DZ	DZ
Layered Product Option 1	A-Kit	A-Kit	
Layered Product Option 2	A-Kit	DZ	

On-Going Service State:

VAX/VMS Option 1	Service-Contract	Service-Contract	Service-Contract
VAX/VMS Option 2	Service-Contract	Service-R.T. Copy/ Additional TSC	Service-R.T. Copy
Layered Product Option	Service-Contract	Service-Contract	
Layered Product Option 2	Service-Contract	Service-R.T. Copy	

NOTE: The DZ license is included in the 11/780 and 11/782 building blocks.

FIELD SERVICE SUPPORT OF VAXcluster SYSTEMS

Bill Freeman
DTN: 285-6124
VW01-1/C05

Digital Customer Services will continue its commitment to its customers by offering a spectrum of services from which a customer may chose to match his service needs.

Field Service Offerings

Field Service will provide hardware maintenance for the VAXcluster via standard service contract agreements.

DECservice Agreement

Digital's most comprehensive on-site service includes a written commitment to respond to the customer's service need within a specified time. This service offering may be tailored to meet the needs of the customer with coverage up to 24 hours per day and seven days per week. When problems are extensive, remedial service will continue until the cluster is once more operational.

Remote Diagnosis is an integral part of DECservice which provides fast and accurate diagnosis from Digital's Remote Diagnostic Centers.

Problem escalation, scheduled preventative maintenance, parts and labor are provided under the DECservice agreement, plus Field Service will install the latest engineering modifications to keep the clustered systems up to date.

Basic Service Agreement

Field Service offers the Basic Service Agreement to customers who do not require the fixed response time and continuous remedial service. Basic Service provides next day response, eight hours per day, five days per week.

Remote diagnosis, problem escalation, preventative maintenance, parts, labor, material and installation of engineeirng modifications are included in this service agreement.

Guaranteed Uptime Agreement

Customers with a need for high availability and who are willing to meet certain site and system requirements are able to take advantage of Digital's Guaranteed Uptime Agreement and receive a commitment from Digital of 96%, 98%, and 99% uptime, depending on their needs.

Remote Diagnosis is a requirement of the Guaranteed Uptime Agreement.

APPENDICES

APPENDIX A

PRICING AND AVAILABILITY SUMMARY

The following charts summarize the pricing and availability information for the products described in this Special Issue of Sales Update.

For specific product prerequisities, please refer to the appropriate product SPD and/or VAX Systems and Options Catalog. If there are any questions concerning this information, please contact your appropriate product line representative.

Key

MLP - Purchase price stated in U.S. dollars, FOB Digital plant, and apply only in the continental United States. Federal, state and local taxes are not included. All prices and specifications are subject to change.

BMC - Field Service 8-hour/5-day monthly maintenance charges for BASIC, OEM and Terminals Service Agreements coverage in U.S. dollars.

VAX-11/780 & 11/782
VAXcluster SYSTEM BUILDING BLOCKS
PRICING AND AVAILABILITY SUMMARY

Model Number	Description	MLP	BMC	OEM Discount	Availability*
	VAX-11/780 VAXcluster System Building Blocks:				
780CI-AE	VAX-11/780 2MB 64K ECC MOS Memory, CI780 CI Adapter, SC008 Star Coupler, HSC50 with one Disk Data 2-sets CI cables (BNCIA-20) Channel and one Tape Data Channel, VAX/VMS V3.3 or later (QE001-DZ) license only, 120V/60Hz	217,000	724	2	Q1FY84
780CI-AJ	Same as above except 240V/50Hz	217,000	724	2	Q2FY84
780CI-AP	VAX-11/780 2MB 64K ECC MOS Memory, CI780 CI Adapter, VAX/VMS V3.3 or later (QE001-DZ) license only, 1-set CI cable (BNCIA-20), 120V/60Hz	173,000	557	2	Q1FY84
780CI-AT	Same as above except 240V/50Hz	173,000	557	2	Q2FY84
	VAX-11/782 VAXcluster System Building Blocks:				
782CI-AE	VAX-11/782 4MB 16K ECC MOS Memory, CI780 CI Adapter, SC008 Star Coupler, HSC50 with one Disk Data 2-sets CI cables (BNCIA-20) Channel and one Tape Data Channel, VAX/VMS V3.3 or later (QE001-DZ) license only, 120V/60Hz	370,000	2,130	2	Q1FY84
782CI-AJ	Same as above except 240V/50Hz	370,000	2,130	2	Q2FY84
782CI-AP	VAX-11/782 4MB 16K ECC MOS Memory, CI780 CI Adapter, VAX/VMS V3.3 or later (QE001-DZ)license only, 1-set CI cables (BNCIA-20) 120V/60Hz	320,000	1,963	2	Q1FY84
782CI-AT	Same as above except 240V/50Hz	320,000	1,963	2	Q2FY84

* The VAXcluster program consists of a number of phased product announcements. The first phase includes VAX/VMS Version 3.3 which will be available in May 1983 and VAX/VMS Version 3.4 which will be available in August 1983.

The other future VAX/VMS features described in this Special Sales Update will be available in future releases of VMS. The announcement of the next phase of the VAXcluster program will occur in six to nine months and will include the content definition and availability date for the next major release of VMS.

VAX COMPUTER INTERCONNECT (CI)
PRICING AND AVAILABILITY SUMMARY

Option Number	Description	MLP	BMC	OEM Discount	Availability
CI750-BA	CI750 w/cabinet & power control, 120V/60Hz	18,500	150*	2	Limited ships September 1983 Volume Q2FY84
CI750-BB	CI750 w/cabinet & power control, 220V/50Hz	18,500	150*	2	Same as above
CI750-AA	CI750 10-1/2" box, Mounts in CI750-BA, 120V/60Hz	17,500	150*	2	Same as above
CI750-AB	CI750 10-1/2" box	17,500	150*	2	Same as above
CI750-SA	2 Node Starter Kit 2-CI750-BA, SC008, 2-BNCIA-10, 120V/60Hz	39,000	322*	2	Same as above
CI750-SB	2 Node Starter Kit 2-CI750-BA, SC008, 2-BNCIA-10, 220V/50Hz	39,000	322*	2	Same as above
CI780-AA	VAX-11/780 or VAX-11/782 CI Adapter 120V/60Hz	19,500	150	2	NOW
CI780-AB	Same as above except 240V/50Hz	19,500	150	2	NOW
CI780-SA	CI780 Starter Kit 2-CI780-AA, SC008-AC 2-BNCIA-10, 120V/60Hz	40,000	322	2	NOW
CI780-SB	CI780 Starter Kit 2-CI780-AB, SC008-AC 2-BNCIA-10, 240V/50Hz	40,000	322	2	NOW
SC008-AC	Star Coupler 8 node with cabinet	7,500	22	2	NOW
SC008-AD	Upgrade to Star Coupler for 9-16 Nodes	5,500	22	2	NOW
BNCIA-10	CI cable set, 10 meters	600	N/C	2	NOW
BNCIA-20	CI cable set, 20 meters	830	N/C	2	NOW
BNCIA-45	CI cable set, 45 meters	1,460	N/C	2	NOW

* Field Service pricing will appear in the 2-May-83 Addendum Number 12ZQ4DSPL.

HSC50 PRICING AND AVAILABILITY SUMMARY

Option Number	Description	MLP	BMC	OEM Discount	Availability
HSC50-AA	Standalone Pkg w/processor, memory CI interconnect, 60Hz	32,500	95	2	Q4FY83
HSC50-AB	Standalone Pkg same as above except 50Hz	32,500	95	2	Q2FY84
HSC5X-BA	Disk Data Channel	7,100	25	2	Q4FY83
HSC5X-EA	Supplemental Power Supply	2,600	25	2	Q4FY83

VAX/VMS PRICING AND AVAILABILITY SUMMARY

Product Q-Number	Description	MLP	DPMC	BSMC	SMMC	Add On	Availability*
QE001-AZ	VAX/VMS Installation**, 90 Day Warranty, and 5 Training Credits for the VAX-11/780 or VAX-11/782	7,360	***	***	***	990	Q1FY84
QE001-HM	Documentation & Media - Magtape	2,640	N/A	N/A	N/A	990	NOW
QE001-AM	VAX/VMS, Installation Documentation & Media - Magtape	20,000	610	340	175	990	NOW

* The VAXcluster Program consists of a number of phased product announcements. The first phase includes VAX/VMS Version 3.3 which will be available in May 1983 and VAX/VMS Version 3.4 which will be available in August 1983.

The other future VAX/VMS features described in this Special Sales Update will be available in future releases of VMS.

The announcement of the next phase of the VAXcluster program will occur in six to nine months and will include the content definition and availability date for the next major release of VMS.

** Post Warranty SPS Services are available and listed under fully supported line (AM, AJ).

*** See SPS prices for applicable media.

NEW PROMOTIONAL/SALES MATERIALS

Frank Seery
DTN: 251-1578
CF01-2/M38

Karen Kilday
DTN: 251-1670
CF01-2/M38

APPENDIX B

"Announcement Packages" are being mailed to each sales person via the local Literature Contact/Sales Communications Center (SCC), and will be distributed upon receipt. The package will include the following:

1. VAXcluster System Announcement Brochure
2. CI 750/CI780 Product Brochure
3. VAXcluster Technical Summary
4. HSC50 Storage Systems Product Brochure*
5. VAXcluster Systems Building Block (Excerpt from Q4 VAX Systems & Options Catalog)

Based on your Literature Contact's response to Literature Flash #123, bulk quantities of items 1-4 will be dispatched after the "Announcement Packages". Please see your Literature Contact to receive additional copies.

Advertising Campaign

An extensive worldwide campaign is planned to support the VAXcluster announcement. Ads will begin in the spring for the U.S.

Printed Materials

Below is a list of additional literature designed to support you in your VAXcluster sales activities. These materials are available through your local Literature Contact.

- VAX Family Brochure
- VAX-11/750 Computer System Product Brochure
- VAX-11/780 Computer System Product Brochure
- VAX-11/782 Attached Processor Product Brochure
- VAX/VMS Operating System Product Brochure

Newsletter

INSIGHT, Digital's monthly customer newsletter, is automatically sent to each sales office in the U.S. and GIA**. Any of your prospects in the U.S. or GIA** may be added to the mailing list by sending their names and addresses to Sheila Chayes, CF01-2/M88. Additional copies are available from your Literature Contact.

* Discusses specific TA78 support. Tapes will be supported with future releases of VMS.

** except in Canada and Australia

VAXcluster SYSTEM: QUESTIONS AND ANSWERS

APPENDIX C

Does a VAXcluster System offer 100% non stop computing?

Through future releases of VAX/VMS many features that give a user high system availability without the subsequent performance degredation necessary to implement non-stop processing will be provided. Although this approach does not make 100% non stop computing possible, it does give a user the ability to configure a system that can withstand most system-wide failures.

Can DSA disks be dual ported between an HSC and a UDA?

All DSA devices are designed to allow the drives to be dual ported. It is possible to dual port drives between HSC and UDA with the following constraint - only one path is active at any time. A path remains active until operator intervention changes the control of the DSA device to another path. This is in contrast to Massbus dual porting which allows paths to alternate on a per I/O basis.

Is failover to an alternate HSC50 with dual ported disks automatic?

It will be in a future release. If the path to a disk is lost, either by failure of the HSC50 or the SDI, a new path is established automatically by VAX/VMS.

Will all existing VAX/VMS functionality operate cluster-wide?

It is a goal to provide all functionality and services that are available on a single VAX/VMS system across the VAXcluster. However, some process communication capabilities as they exist today may not be available to cluster-wide communications, but will remain available on a single processor within the cluster.

Why is a local disk required on a VAX-11/750?

There is currently no VAX-11/750 boot-ROM for the HSC50. Therefore, booting the VAX-11/750 requires a local system disk. Additional capabilities will be available in the next future release of VAX/VMS that will remove this restriction on the VAX-11/750.

Will there be specific tools to aid the management of a VAXcluster?

Tools will be made available with future releases of VAX/VMS that will aid in the management of the cluster and assist in monitoring cluster usage.

How is UNIX(tm) involved in VAXcluster systems?

Since Digital has recently announced that we will distribute and service UC/Berkeley 4.1 BSD UNIX for VAX, it is important to position this with respect to the Cluster environment. The UNIX(tm) system does not implement the VAXcluster architecture, and therefore cannot coexist in a VAXcluster. The cluster environment is based on VAX/VMS. VAXclusters support the UNIX(tm)

environment with VNX which currently consists of C language, CMS which is equivalent to UNIX SCCS(tm), and VMS MMS which is equivalent to UNIX MAKE(tm).

UNIX(tm) is a trademark of Bell Labs.

If a processor fails, what happens to the locks on the files of the local disks?

If a host processor fails, locks are released except for those specified for a recovery process. These locks remain in place until recovery is completed.

Why must a user restart from a checkpoint on an identical processor?

A checkpoint takes a snapshot of a process state. In order to restart from that snapshot, identical processors are required (780 to 780, 750 to 750, same ECO levels, and same processor options). If the environment differs, the information taken at the checkpoint does not apply and cannot be used to restart the process.

How many CI interfaces are supported by VAX/VMS per processor?

With VAX/VMS V3.3 and V3.4 a single CI interface is supported on a processor. More than one CI will be supported in future releases of VAX/VMS.

How are mass storage devices seen in a cluster?

All disk storage devices (DSA, MASSBUS, UNIBUS) appear as local devices through a naming convention that is known cluster-wide. Local tapes are only accessible by the local CPU. HSC tapes, however, are seen cluster-wide, and allocated to a single processor at a time.

Is there cluster performance information available?

Information regarding the HSC50 performance capabilities of the VAXcluster will become available soon after the announcement. Additional information will be provided as new functionality is implemented and will be published at a later date.

Are the VAX-11/782, 780, and 750 the only processors available in a VAXcluster?

These are the only processors that are available at this time. There are currently no plans to support the VAX-11/730 in a VAXcluster System.

On what devices can a user do volume shadowing?

Volume shadowing with future releases, can be done on any DSA disk. With devices on a UDA, the shadowing is initiated and controlled by VAX/VMS. With devices on the HSC, once volume shadowing is initiated, it is automatically performed by the HSC without VMS intervention.

How are files copied between mass storage (disks and tapes) on an HSC?

The HSC has a local backup capability (disk-to-disk, disk-to-tape) with no host involvement. This is a volume level copy and is a subset of the VAX/VMS backup format. For copying at the file level between mass storage on the HSC, the transfer flow is HSC-host-HSC.

How does a user application find a file on a local disk? an HSC50? a remote disk physically attached to another node?

All disk mass-storage in a VAXcluster appears as a local device on each processor. The user, therefore, sees the device as if it were physically connected to his machine. VAX-11 RMS (the VAX/VMS Record Management System) uses the distributed lock manager. The distributed lock manager synchronizes cluster-wide FILE/RECORD level access for all programs that use RMS (e.g. all VAX/VMS utilities). If the user does I/O directly to the device (logical I/O priviledge required) then again the device is seen as if it were local but no locking or synchronization takes place.

What is the throughput of DECnet-VAX using the CI as the physical link?

The intended use of DECnet over the CI is for cluster management and low volume data exchange and not for high speed task-to-task communication. DECnet-VAX uses the CI as the physical link through software which replaces the DDCMP layer. The other layers of DECnet-VAX are unchanged, and thus the same user interface and the associated overhead is present. Aggregate throughput is approximately 1 megabit at each node. Device transparency and disk mass-storage device access over the CI is achieved using new VAX/VMS system level protocols, and does not use DECnet.

How do VAXclusters compare to LAN?

The processors in VAXclusters cooperate much more closely than nodes in LANs (Local Area Networks). In a cluster, the Distributed Lock Manager synchronizes access to resources within the cluster allowing this closer cooperation among the cluster nodes. The MSCP facilities and data flow between HSC50s and processors within a cluster happens at a rate that cannot be achieved on a network. These facilities allow clustered processors to directly access files and records in the cluster, without any need to transmit the data in any intermediate format; for example, a network data packet.

The effect of these capabilities is that individual processors in a VAXcluster can share data at high speeds without special programming or procedures in the application. This contrasts with the operation of a LAN, in which applications might need to specify nodenames, access control information, and tolerate the transmission delays caused by a network protocol.

In VAXcluster systems, DECnet-VAX is a useful tool for managing a cluster by means of virtual terminal capabilities, and communication to systems, networks, and devices beyond the physical cluster site.

Security benefits of a cluster are achieved by the fact that all of the VAXcluster nodes are typically within a single computer room. Networks are closely or widely distributed and only as secure as their least secure node. VAXcluster systems have a single system manager, while distributed networks typically have a system manager for each network node.

A VAXcluster is also a homogeneous entity, consisting of just VAX/VMS processors, all having identical user-access controls. A network usually is heterogenous -- a mix of various systems -- and therefore has a mix of various access controls, some being less secure than others.

Can DECSYSTEM 20 share the same CI or be a member of the VAXcluster?

No, cooperation of the operating systems (i.e. TOPS20, VMS) would be required to implement any form of coexistence. In addition, the 32-Bit and 36-Bit architectures are different (in data format, word lengths, byte sizes, file structures, disk sector lengths and disk formats). Extensive software would be required to overcome these differences. There are no software engineering plans to do this.

Title: VAX-11 Software Engineering Manual -- Rev 3

Specification Status: draft

Architectural Status: under ECO control

File: SEPRR3.RNO

PDM #: not used

Date: 19-Feb-77

Superseded Specs: none

Author(s): F. Bernaby, P. Conklin, S. Gault, T. Hastings, P. Marks, R. Murray, I. Nassi, M. Spier

Typist: P. Conklin

Reviewer(s): R. Brender, D. Cutler, R. Gourd, T. Hastings, I. Nassi, S. Poulsen, D. Tolman

Abstract: This manual presents the VAX-11 programming conventions and software engineering practices as developed for, and adopted by the Central Engineering Group. These conventions and practices are standard within Central Engineering; we hope that they be used by other corporate groups as well. Designed to be the "programmer's helper", the manual contains the coding conventions as well as practical data of technical, procedural, administrative and conceptual nature that would be useful to the software engineer.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Original	M. Spier	14-Apr-76
Rev 2	Revised from Review	P. Marks	21-Jun-76
Rev 3	After 6 months experience	P. Conklin	19-Feb-77

Rev 2 to Rev 3:

1. Convert to standard RUNOFF manual chapter format.

2. Remove unwritten sections.

3. Move introductory note to preface.

4. Remove request for review from preface.

5. Note relationship to BLISS conventions.

6. Split chapter 6 into 6 and 7. Add chapters 8-11, especially BLISS. Add the BLISS transportability guidelines. Add Chapter 15 for diagnostics.

[End of SEPRR3.RNO]

VAX - 11

SOFTWARE ENGINEERING MANUAL

19 FEBRUARY 1977

Revision 3

Digital Equipment Corporation, Maynard, Massachusetts

Revision 1, April, 1976
Revision 2, June, 1976
Revision 3, February, 1977

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

This draft standard does not describe any program or product which is currently available from Digital Equipment Corporation. Nor does Digital Equipment Corporation commit to implement this standard in any program or product. Digital Equipment Corporation makes no commitment that this document accurately describes any product it might ever make.

Digital Equipment Corporation's software is furnished under a license and may only be used or copied in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital or its affiliated companies.

The following are trademarks of Digital Equipment Corporation:

CDP	DIBOL	KA10	RAD-8
COMPUTER LAB	DIGITAL	KI10	RSTS
COMSYST	DNC	LAB-8	RSX
COMTEX	EDGRIN	LAB-K	RT-11
DDT	EDUSYSTEM	MASSBUS	RTM
DEC	FLIP CHIP	OMNIBUS	SABR
DECCOMM	FOCAL	OS/8	TYPESET-8
DECUS	GLC-8	PDP	TYPESET-10
DECsystem-10	IDAC	PHA	TYPESET-11
DECsystem-20	IDACS	PS/8	UNIBUS
DECtape	INDAC	QUICKPOINT	

PREFACE

Over the years, much ado has been made about coding standards and conventions. Everyone believed that conventions are good, so long as they are not the other guy's conventions! Committees were formed, and reformed, and left to die for lack of consensus. We repeatedly refused to follow conventions that we deemed "imperfect" and consequently we followed none at all.

A great deal of this has been foolish nit-picking on the part of our vast multitude of entrepreneurs. The time has come to stop the foolishness and to recognize the reasons for which code uniformity is mandated.

Standards, conventions and uniform practices all aid us in producing reasonably professional, maintainable products of consistent quality. Any individual can always have a private opinion as to what is "good", or "right", or "efficient" or "aesthetic". Any collection of individuals invariably comes up with as many divergent opinions on the subject as there are individuals. We should all be sufficiently mature and sufficiently professional to be willing to compromise with both our egos and our fellow peers; to compromise just enough to accept objectively a set of reasonable conventions that will establish the uniformity and consistency of all of our software products.

The Methodology group has compiled the conventions and practices presented in this manual. They apply to all VAX-11 programming. They are based on existing PDP-11 coding practices. This manual was reviewed by the Coding Conventions Committee consisting of Peter Conklin, Dave Cutler, Roger Gourd, Steve Poulsen and Mike Spier. These conventions have been broadened to the BLISS environment by review with Ron Brender, Rich Grove, and Dave Tolman. Transportability issues have been addressed in concert with Peter Marks and Ike Nassi.

We want these conventions to be adopted willingly, not forced upon people through arbitrary managerial edict. This is best accomplished by having you formulate to yourself exactly WHY you find some convention to be objectionable; then try and propose --to yourself-- an alternate one, and reflect on whether or not the new one is really that much superior, and why. All that we ask of you is to convince yourself that these conventions are no less reasonable than any other set of conventions. Then, we hope, you will be willing to show sufficient professional maturity to adopt and follow these conventions.

This document is the result of integrating and reorganizing the BLISS Software Engineering Manual and the VAX Assembler Software Engineering Manual published during the summer of 1976. New chapters have been incorporated, covering transportability, naming conventions, and external interface specifications. We solicit constructive criticism and recommendations for enhancement. In particular, the last chapter contains a list of topics we would like to address in future editions. Please feel free to contribute toward these topics. This is completely a home grown document. If you feel this is a desirable way to proceed, you should feel a responsibility to review this carefully and to contribute material you feel appropriate. The value of the document depends directly on the quality and applicability of the submitted material.

CONTENTS

[End of Prefix]

Title: VAX-11 Software Engineering Introduction -- Rev 3

Specification Status: draft

Architectural Status: under ECO control

File: SE1R3.RNO

PDM #: not used

Date: 23-Feb-77

Superseded Specs: none

Author: P. Conklin, P. Marks, M. Spier

Typist: P. Conklin

Reviewer(s): R. Brender, D. Cutler, R. Gourd, T. Hastings, I. Nassi, S. Poulsen, D. Tolman

Abstract: The introduction gives a chapter by chapter overview of the manual and how it is organized.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Original	M. Spier	14-Apr-76
Rev 2	Revised from Review	P. Marks	21-Jun-76
Rev 3	After 6 months experience	P. Conklin	23-Feb-77

Rev 2 to Rev 3:

1. Change to be the general contents and guide to the chapters.
2. Add Chapter 7.
3. Add Chapter 8.
4. Add purpose of manual.
5. Split chapter 6 into 6 through 11.
6. Add chapters 14 and 15.
7. Collect loose ends as last chapter (# 99).

[End of SE1R3.RNO]

CHAPTER 1

INTRODUCTION

23-Feb-77 -- Rev 3

This manual is concerned with software engineering practices in the VAX-11 environment. It does not discuss or define the differences between VAX-11 and other environments. Designed to be the "programmer's helper", the manual contains the coding conventions as well as practical data of technical, procedural, administrative and conceptual nature that would be useful to the software engineer.

This manual has two purposes:

- to provide the Software Engineer with information not normally found in language reference manuals such as usage notes and symbol construction rules.
- to present recommended standards, conventions, and practices such as commenting, formatting, and documentation.

Conventions, standards and practices can assure good, professional, maintainable products of consistent quality. They need not encroach on the programmer's "right" to be creative in his or her expression of a program.

Chapter 1 is the introduction and gives a guide to the manual's organization. It includes a chapter by chapter overview.

Chapter 2 tells how to use this manual. It tells how to find the exact information needed. It also gives the notations used in the manual.

Chapter 3 is the methodological policy statements. These are the policies which lead to the specifics of the format. They also outline the basic structure of programs into modules. The policy statements include the goals to be attained by following them. These policies include the choice of language, the layout of the source text, the

separation into modules, and the sharing of code.

Chapter 4 is a program structure overview. It lists the source module's textual elements, and gives examples of the parts of the program. This pulls together in one place the details documented later in chapter 6.

Chapter 5 gives the standard module template files and the instructions for using them. The standard template contains all of the standard boilerplate as a convenience to save excessive retyping.

Chapter 6 details the commenting conventions. These are consistent across all source languages. The entries are arranged alphabetically for ease of reference. There is extensive cross-referencing to aid retrieval. For each item, it gives the background and the rules, and then gives templates and examples.

Chapters 7 through 11 give usage and formatting conventions for each of our programming languages. The languages covered are assembler, BASIC, BLISS, COBOL, and Fortran. Although there is occasional redundancy between these chapters, we felt it better to minimize retrieval difficulty at the expense of some duplication. The chapters are layed out in the same style as Chapter 6. When a topic deserves more than a page to describe, an outline is given here and a cross reference is made to a fuller presentation in some other chapter.

Chapter 12 is the naming conventions. These include the formation of symbols reserved to Digital and the list of facility prefixes.

Chapter 13 gives details on forming external and interface documentation. In particular, it includes details on the notation for specifying procedure arguments.

Chapter 14 contains guidelines for the transportation of BLISS programs across architectures.

Chapter 15 contains additional information and guidelines for writing diagnostics programs.

The last chapter is a collection of loose ends and future sections.

The appendices give full sample programs written to this standard.

[End of Chapter 1]

Title: VAX-11 Software Engineering How to Use -- Rev 3

Specification Status: draft

Architectural Status: under ECO control

File: SE2R3.RNO

PDM #: not used

Date: 26-Feb-77

Superseded Specs: none

Author: P. Conklin, P. Marks, M. Spier

Typist: P. Conklin

Reviewer(s): R. Brender, D. Cutler, R. Gourd, T. Hastings, I. Nassi, S. Poulsen, D. Tolman

Abstract: Chapter 2 gives a guide to the use of the manual and gives its notations. It suggests ways of looking up information in it.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Original	M. Spier	14-Apr-76
Rev 2	Revised from Review	P. Marks	21-Jun-76
Rev 3	After 6 months experience	P. Conklin	26-Feb-77

Rev 2 to Rev 3:

1. Replace usage cross reference notation.

2. Note split of commenting and usage chapters.

[End of SE2R3.RNO]

CHAPTER 2

HOW TO USE THIS MANUAL

26-Feb-77 -- Rev 3

This manual assumes familiarity with the VAX-11 languages. Its purpose is to serve as a guide to the precise use to which certain language features may be put.

The introduction (chapter 1) indicates the chapters of the manual, explaining what each chapter contains. The sts table of contents lists individual sections within the chapters. The index is organized by keywords (e.g., COMMENT, ROUTINE, STATEMENT, etc.)

Suppose that you were told that your program needs better comments. You should typically look up the concept under "C" in the chapter on commenting. Similarly, if you were told that your useage or formatting of some source statement was poor, you could look it up under the statement's name in the chapter on formatting and usage for your language.

This will enable you immediately to retrieve the information required, and have the exact amount of information that is pertinent to your immediate needs. You may then get additional information about the keyworded item in the other chapters. The important point is that such additional information is not confused with the information needed for some specific reason. The manual is deliberately not organized for front- to back-cover sequential reading.

Keyworded data is cross referenced. The rules pertaining to keyword "A" may require knowledge or use of keywords "B" and "C".

- Knowledge of "B" and "C": a "SEE ALSO" pointer indicates the related item(s) which you should also understand.

- Use of "B" or "C": the first occurrence of "B" and of "C" within "A" is prefixed with the word "see" serving as a reference pointer to indicate the possible need to consult those keywords in turn.

There may be variants of a single keyworded concept. For example LABEL and LOCAL LABEL. In this case, the keywords are ordered by the main concept (e.g., LABEL), and any variant is to be retrieved by suffixing that keyword with the qualifying key word. We use the colon ":" as a qualification delimiter within the manual (e.g., LABEL: LOCAL).

Finally, whenever this manual is reissued, all changes relative to the immediately preceding version of the manual will be indicated by means of a left margin change bar, as illustrated to the left of this entire paragraph.

[End of Chapter 2]

Title: VAX-11 Software Engineering Policy -- Rev 3

Specification Status: draft

Architectural Status: under ECO control

File: SE3R3.RNO

PDM #: not used

Date: 26-Feb-77

Superseded Specs: none

Author: P. Conklin, P. Marks, M. Spier

Typist: P. Conklin

Reviewer(s): R. Brender, D. Cutler, R. Gourd, T. Hastings, I. Nassi, S. Poulsen, D. Tolman

Abstract: Chapter 3 gives the methodological policy statements. These include the choice of language, the layout of the source text, the separation into modules, and the sharing of code.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Original	M. Spier	14-Apr-76
Rev 2	Revised from Review	P. Marks	21-Jun-76
Rev 3	After 6 months experience	P. Conklin	26-Feb-77

Rev 2 to Rev 3:

1. Remove reference to page boundaries.
2. Allow code in application languages.
3. Document reasons for structure and for transportability.
4. Limit interface data types to call standard.
5. Remove references to self-initializing.

[End of SE3R3.RNO]

CHAPTER 3

METHODOLOGICAL POLICY

26-Feb-77 -- Rev 3

1. All system programs for the VAX-11 family are written in an application language or one of the two official system implementation languages:

   - The VAX-11 Macro Assembler, or
   - BLISS-32

   Of these, BLISS is the default choice for a language. BLISS is intended to replace as much assembly code as possible. The assembler will be used as a system implementation language only for:

   - Hardware dependent routines, such as interrupt handlers or I/O drivers, where extreme machine dependency coupled with high performance requirements rule out the use of BLISS.
   - Cases where functionality is needed that is not supplied by BLISS; for example, routines which are to be invoked in a non-standard way.
   - Routines which cannot be written in BLISS because of compilation difficulty (as distinct from functional impossibility, or undesirability). This category includes all routines which would have been coded in BLISS had there been available a BLISS compiler that supports the required technicalities (e.g., special relocation or addressing features). All these routines are, in principle, candidates for future recoding in BLISS, conditions permitting.

2. All code will be written uniformly, according to these conventions, in order to:

   - Make system code meaningfully readable. If source code is not properly structured, organized, and indented according to these conventions, you have obscured the algorithm from the reader. The code should be structured into blocks with a limited amount of branching. This allows a graphical reflection of the control flow. If the code is unstructured, you have lost the ability to understand and modify it.

   - Enable all programmers to read, understand and be able to modify one another's code, regardless of source language. Note that the documenting conventions are identical across all our languages.

   - Enable field support personnel (both software specialists and hardware engineers) to read and understand VAX-11 system code. To lower field support costs by eliminating, as much as possible, the need for software specialists who are knowledgeable of certain routines only, and to further the software specialist's ability to master any system code.

   - Make our software well documented: make programs both readable and comprehensible by being able to extract technical documentation from the source code itself. Facilitate the work of technical writers by providing them with uniform, well documented source code.

   - Reduce the bug rate and enhance the quality and stability of our software products. Maintain the product's initial high quality throughout its lifetime: through cycles of bug fixes, modifications and functional evolution.

3. All major bodies of code, or distinct logical sub-systems (with the exception of speed/size sensitive executive or diagnostic modules), will be coded as independent routines using the standard call/return interface, to:

   - Encourage and facilitate the use of BLISS in non-critical sections of system software, and to

   - Encourage the future recoding of assembly language routines in BLISS, conditions permitting.

   - Enhance the ability to transport non-assembly language code.

   - Limit the interface data types to those specified as part of the calling standard.

4. All user-level system products (language processors, utilities, library subroutines, etc.) should be designed and implemented so that they may be transportable between systems and/or family architectures. Keep in mind that:

   - o Transportability is a major goal to which Central Engineering is firmly committed.

   - o Transportability has to be designed carefully into the product, and carefully realized by following the transportability guidelines.

   - o All machine-dependent features are to be avoided as a rule. If necessary, they should be localized to a clearly-identifiable, non-transportable module.

5. The sharing of code is encouraged as much as possible. Whenever possible, use a library service routine instead of coding your own version of that same function. If such a library routine does not yet exist, code one that is of general nature, and submit it to the library.

6. All programs are to be written modularly, in small self-contained modules that are maintained as individual source files. These modules will be assembled separately. The object code files will be linked to form the larger software product. Modularity will benefit us by:

   - o Enhancing quality: each module can be tested and debugged separately; small modules are more easily controllable than large bulky programs.

   - o Isolating functionality: it becomes easier to custom tailor a system through selective linking of exactly those modules that are needed.

   - o Enhancing modifiability: the modification of a given module will be less likely to have an undesirable side effect on some other module's functionality.

   - o Working set control: the ability to rearrange the linking order of modules is a most powerful tool in optimizing program behavior within a paged runtime environment.

7. All modules (with the possible exception of certain core executive or diagnostic programs) are to be written as pure, non-self modifying and well localized code.

   - Self initializing: With the exception of system startup or bootstrap code, all routines should be self initializing. If they depend on an initial value of some permanent allocation (OWN) variable, initialize that variable dynamically rather than relying on compile time or link time value settings.

   - Well localized: VAX-11 is a virtual memory machine. Any piece of code may --whether originally intended to, or not-- possibly run in a demand paging environment. You should make the greatest efforts possible to design and structure your code in such a way that the locality of reference is kept to a minimum. Don't promiscuously branch over a large absolute address span. Don't make reference to widely (and wildly) fragmented database elements within a single sequence of instructions, and especially within the scope of a tight loop.

[End of Chapter 3]

Title: VAX-11 Software Engineering Program Structure -- Rev 3

Specification Status: draft

Architectural Status: under ECO control

File: SE4R3.RNO

PDM #: not used

Date: 28-Feb-77

Superseded Specs: none

Author: P. Conklin, P. Marks, M. Spier

Typist: P. Conklin

Reviewer(s): R. Brender, D. Cutler, R. Gourd, T. Hastings, I. Nassi, S. Poulsen, D. Tolman

Abstract: Chapter 4 overviews and then details the layout of a module. It includes examples of the module and routine prefaces.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Original	M. Spier	14-Apr-76
Rev 2	Revised from Review	P. Marks	21-Jun-76
Rev 3	After 6 months experience	P. Conklin	28-Feb-77

Rev 2 to Rev 3:

1. Remove the VERSION NUMBER statement.
2. Explain why routine owns are discouraged.
3. Update to use template in the example.
4. Define calling sequence vs. input and output parameters.
5. Add references to $FORMAL, etc., macros.
6. Change CONFIGURATION to ENVIRONMENT.
7. Combine abbreviated and detailed edit history.
8. Add weak and validation section.
9. Add BLISS to show similarity.
10. Add critical algorithms to functional description.
11. Use NONE for inapplicable sections, do not delete them.
12. Title and ident are first two lines.
13. Legal notices are fully capitalized.
14. Edits have initials if several editors per version.
15. Ident examples include edit.
16. BLISS module head includes other module switches.
17. BLISS structure defs are together.
18. Add blank line after legal notices.

[End of SE4R3.RNO]

CHAPTER 4

PROGRAM STRUCTURE

28-Feb-77 -- Rev 3

Programs are written in modules. The module is the source text that is assembled or compiled as a unit. Each module can be coded in any language. The program structure and commenting conventions are consistent across all languages to allow the reader to learn one pattern independent of the writer's choice of language. Also, for reader ease, every section and subsection must appear in its standard position. If a section or comment is not applicable, enter the word NONE as a separate line. This is done to make the reader's job as simple and clear as possible. Each module exists as a separate source text file, and is structured as follows:

4.1 THE MODULE PREFACE

It provides the necessary documentation to explain the module's functionality, use and history. It consists of the following items in the exact given order. All items must be included.

- A title statement specifying the module's name. The title is a symbol of up to 15 characters in length. This statement has a comment indicating the module's functionality. The title statement, together with its comment, are reproduced as page headers in the listing. The title statement is always the first line of the file.

- An IDENT statement indicating the module's current version number. The ident statement is always the second line of the file.

- The standard DEC legal notices fully capitalized for emphasis.

- A FACILITY statement. A module may be a dedicated part of a larger linked facility, or part of several facilities, or a general purpose library function. This statement identifies the larger whole of which the module is part.

- A short functional description of the module (a documenting comment) including the design basis for any critical algorithms. If the module requires an extensive functional description, then this item is an abstract of the description, and is identified as such by the keyword ABSTRACT. The extensive functional description will then be provided on the following page.

- ENVIRONMENT statement. Give any special environmental assumptions such as access modes, OTS, etc. If the module's assembly is governed by a system wide configuration file, then state the file(s)'s name(s). Otherwise if the module has special conditional assembly parameters, then specify very explicitly what they are and what values they assume under all given conditions.

- The author and date on which the module was coded.

- The detailed current edit history. This item specifies the versions, the modifier, and the last date of each version. This item also lists the specific changes made between base levels (during production) or releases, providing a short functional description of each problem and its solution, as well as appropriate reference information such as SPR number(s), etc. The comments include the full name of the person responsible for each version. If several people modify the module, the initials of the others appear in each edit line.

4.2 THE MODULE'S DECLARATIVE PART

It contains:

- For BLISS, specification of the table of contents.

- Specification of INCLUDE files or library definitions.

- Definition of local macros.

- Declaration of local equated symbols

- Declaration of own storage allocations.

- Specification of externals. For assembly language, only WEAK or VALIDATION externals need be listed.

4.3 THE MODULE'S ACTUAL CODE

This is in the form of zero or more ROUTINE(s). The module may have no routines in it (i.e., no executable code) if it is a DATA SEGMENT MODULE. Each routine consists of the following sequence of items:

4.3.1 The ROUTINE PREFACE

- A routine statement specifying the routine's name. This statement has a comment indicating the routine's functionality. The routine statement, together with its comment, are reproduced as page headers in the listing.
- A detailed functional description of the routine.
- A list of the routine's calling sequence, input and output parameters.
- A list of the implicit inputs and outputs, and functional side effects, if any, of the routine's code.

4.3.2 The Routine's Declarative Part

- Specification of local INCLUDE file(s), if appropriate. Normally, such use is not recommended.
- Declaration of local (stack frame resident) variables.
- Declaration of optional equated symbols, own storage allocation variables and macros, all of which are local to this routine. In general, use of these local items is not recommended unless it adds significant clarity. Usually, these are better declared at the module level.

4.3.3 The Routine's Code

- For assembly language, the routine's entry point(s).
- The routine's body.
- The routine's return instruction.

4.4 MODULE TERMINATION

An end module statement terminates the module.

4.5 ANNOTATED SAMPLE LAYOUTS

The above are explained in detail in the commenting and formatting chapters of this manual. In the following sections a sample layout of the module format is presented. Samples are given for both assembler and BLISS coding to show the similarity.

The following notations are used to designate source listing formatting:

- <new page> indicates an inserted form feed "CTRL/L" character or an assembler .PAGE directive, to force the listing onto a new page.
- <separator> indicates either several (normally=4) <skip>s or a <new page>. A <separator> is indicated wherever it would be desirable to force a new page, if the present page is sufficiently full. If the last section only marginally fills the present page, and the following item of text would remain on the page, then they can both appear on the same page separated by several blank lines.
- <skip> indicates a blank line.
- <space> indicates a single blank character.
- <tab> indicates a horizontal tab character.

4.6 SAMPLE LAYOUT OF THE MODULE PREFACE

4.6.1 Example Of The Assembler Module Preface

```
        .TITLE  EXAMPLE - <terse functional description>
        .IDENT  /03-05/

;
; COPYRIGHT (C) 1977
; DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASSACHUSETTS 01754
;
; THIS SOFTWARE IS FURNISHED  UNDER A LICENSE FOR USE ONLY ON A SINGLE
; COMPUTER  SYSTEM AND  MAY BE  COPIED ONLY WITH  THE INCLUSION OF THE
; ABOVE COPYRIGHT NOTICE.  THIS SOFTWARE, OR ANY OTHER COPIES THEREOF,
; MAY NOT BE PROVIDED OR  OTHERWISE MADE AVAILABLE TO ANY OTHER PERSON
; EXCEPT FOR USE ON SUCH SYSTEM AND TO ONE WHO AGREES TO THESE LICENSE
; TERMS.  TITLE TO AND  OWNERSHIP OF THE  SOFTWARE  SHALL AT ALL TIMES
; REMAIN IN DEC.
;
; THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT NOTICE
; AND SHOULD  NOT BE CONSTRUED  AS A COMMITMENT  BY DIGITAL  EQUIPMENT
; CORPORATION.
;
; DEC ASSUMES  NO  RESPONSIBILITY  FOR  THE USE OR  RELIABILITY OF ITS
; SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DEC.
;

;++                                      <this is a DOCUMENTING COMMENT>
; FACILITY: General Library
;
; FUNCTIONAL DESCRIPTION: (or ABSTRACT:)
;
;       A short 3-6 line functional description of the module.
;       If an extensive functional description is called for,
;       then this should be a short abstract.
;
; ENVIRONMENT: User Mode with OTS
;
; AUTHOR: Charlie Brown, CREATION DATE: 4-Jul-76
;
; MODIFIED BY:
;
;       Lucy vanPest, 17-Aug-76: VERSION 02
; 01    -  Program Crashes if Disk Error
; 02    -  SPR #4711: reads incorrect block after error.
;
;       Snoopy Beagle Brown, 19-Dec-76: VERSION 03
; 03    -  SPR #5391: reads blocks backward if 50 hertz.
; 04    -  Power fail recovery not reliable
; 05    -  (LVP) SPR #5432: recover if ECC recoverable.
;--                                      <end of DOCUMENTING COMMENT>
<new page>
```

4.6.2 Example Of The BLISS Module Preface

```
MODULE EXAMPLE ( ! <terse functional description>
        IDENT='03-05'
        <other module switches>
        ) =

!
! COPYRIGHT (C) 1977
! DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASSACHUSETTS 01754
!
! THIS SOFTWARE IS FURNISHED UNDER A LICENSE FOR USE ONLY ON A SINGLE
! COMPUTER SYSTEM AND MAY BE COPIED ONLY WITH THE INCLUSION OF THE
! ABOVE COPYRIGHT NOTICE. THIS SOFTWARE, OR ANY OTHER COPIES THEREOF,
! MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY OTHER PERSON
! EXCEPT FOR USE ON SUCH SYSTEM AND TO ONE WHO AGREES TO THESE LICENSE
! TERMS. TITLE TO AND OWNERSHIP OF THE SOFTWARE SHALL AT ALL TIMES
! REMAIN IN DEC.
!
! THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT NOTICE
! AND SHOULD NOT BE CONSTRUED AS A COMMITMENT BY DIGITAL EQUIPMENT
! CORPORATION.
!
! DEC ASSUMES NO RESPONSIBILITY FOR THE USE OR RELIABILITY OF ITS
! SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DEC.
!

!++                                      <this is a DOCUMENTING COMMENT>
! FACILITY: General Library
!
! FUNCTIONAL DESCRIPTION: (or ABSTRACT:)
!
!       A short 5-6 line functional description of the module.
!       If an extensive functional description is called for,
!       then this should be a short abstract.
!
! ENVIRONMENT: User Mode with OTS
!
! AUTHOR: Charlie Brown, CREATION DATE: 4-Jul-76
!
! MODIFIED BY:
!
!       Lucy vanPest, 17-Aug-76: VERSION 02
! 01    -  Program Crashes if Disk Error
! 02    -  SPR #4711: reads incorrect block after error.
!
!       Snoopy Beagle Brown, 19-Dec-76: VERSION 03
! 03    -  SPR #5391: reads blocks backward if 50 hertz.
! 04    -  Power fail recovery not reliable
! 05    -  (LVP) SPR #5432: recover if ECC recoverable.
!--                                     <end of DOCUMENTING COMMENT>
<new page>
```

4.7 SAMPLE LAYOUT OF THE MODULE DECLARATIONS

4.7.1 Example Of The Assembler Module Declarations

```
        .SBTTL  DECLARATIONS
;
; INCLUDE FILES:
;

        <library INCLUDE files and library macros which define:
             MACROs, assembly parameters, systemwide equated
             symbols, table definitions>

;
; MACROS:
;

        <local macro definitions>

;
; EQUATED SYMBOLS:
;

        <equated symbol definitions>

;
; OWN STORAGE:
;

        <declaration of permanent storage allocations>
        <also local storage structures, etc.>
        <if many structures, give each a heading>
        <see $OWN and structure macros>

;
; WEAK AND VALIDATION DECLARATIONS:
;

        <only include section if any declared>

<new page>
```

4.7.2 Example Of The BLISS Module Declarations

```
!
! TABLE OF CONTENTS:
!

        <forward routine declarations in order with
             a summary description of each>

!
! INCLUDE FILES:
!

        <library REQUIRE files and library macros which define:
             MACROs, assembly parameters, systemwide equated
             symbols, table definitions>

!
! MACROS:
!

        <local macro definitions other than structure definitions>

!
! EQUATED SYMBOLS:
!

        <LITERAL and BIND declarations>
        <when a group of structure, macro, and literal declarations
         define a structure they should be grouped together here>

!
! OWN STORAGE:
!

        <declaration of permanent storage allocations>
        <also local storage structures, etc.>
        <if many structures, give each a heading>

!
! EXTERNAL REFERENCES:
!

        <externals with short description>

<new page>
```

4.8 SAMPLE LAYOUT OF THE ROUTINE PREFACE

4.8.1 Example Of The Assembler Routine Preface

```
|
|       .SBTTL  EXAMPLE - <short one-line description>
|  ;++                              <this is a DOCUMENTING COMMENT>
|  ; FUNCTIONAL DESCRIPTION:
|  ;
|  ;       <detailed functional description of the routine>
|  ;
|  ; CALLING SEQUENCE:
|  ;
|  ;       <instruction for calling this routine>
|  ;       <include AP-list if applicable>
|  ;       <see $FORMAL macro>
|  ;
|  ; INPUT PARAMETERS:
|  ;
|  ;       <list of explicit input parameters other than AP-list>
|  ;       <typically registers or stacked arguments>
|  ;
|  ; IMPLICIT INPUTS:
|  ;
|  ;       <list of inputs from global or own storage>
|  ;
|  ; OUTPUT PARAMETERS:
|  ;
|  ;       <list of explicit output parameters other than AP-list>
|  ;       <typically registers or stacked results>
|  ;
|  ; IMPLICIT OUTPUTS:
|  ;
|  ;       <list of outputs in global or own storage>
|  ;
|  ; COMPLETION CODES:
|  ;
|  ;       <list of R0 completion codes>
|  ;       <if standard function, change heading to FUNCTION VALUE>
|  ;       <if the hardware condition codes are set,
|  ;               change the heading to CONDITION CODES>
|  ;
|  ; SIDE EFFECTS:
|  ;
|  ;       <list of functional side effects including environmental changes>
|  ;       <exclude implicit outputs of global or own storage>
|  ;       <list all SIGNALs generated if any>
|  ;--                              <end of DOCUMENTING COMMENT>
|  <separator>
```

4.8.2 Example Of The BLISS Routine Preface

```
ROUTINE EXAMPLE (arguments) =   !<short one-line description>
!++                             <this is a DOCUMENTING COMMENT>
! FUNCTIONAL DESCRIPTION:
!
!       <detailed functional description of the routine>
!
! FORMAL PARAMETERS:
!
!       <list formal parameters and give documentation of them>
!
! IMPLICIT INPUTS:
!
!       <list of inputs from global or own storage>
!
! IMPLICIT OUTPUTS:
!
!       <list of outputs in global or own storage>
!
! COMPLETION CODES:
!
!       <list of function value completion codes>
!       <if standard function, change heading to FUNCTION VALUE>
!
! SIDE EFFECTS:
!
!       <list of functional side effects including environmental changes>
;       <exclude implicit outputs of global or own storage>
!       <list all SIGNALs generated if any>
!--                             <end of DOCUMENTING COMMENT>
<separator>
```

[End of Chapter 4]

Title: VAX-11 Software Engineering Template -- Rev 3

Specification Status: draft

Architectural Status: under ECO control

File: SE5R3.RNO

PDM #: not used

Date: 28-Feb-77

Superseded Specs: MARS template by R. Gourd

Author: P. Conklin, P. Marks, M. Spier

Typist: P. Conklin

Reviewer(s): R. Brender, D. Cutler, R. Gourd, T. Hastings, I. Nassi, S. Poulsen, D. Tolman

Abstract: Chapter 5 presents the standard template files. It also includes step by step instructions for editing them to form a module in standard format.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Original	M. Spier	14-Apr-76
Rev 2	Revised from Review	P. Marks	21-Jun-76
Rev 3	After 6 months experience	P. Conklin	28-Feb-77

Rev 2 to Rev 3:

1. Add instructions from Gourd memo RSG028 Rev 2.
2. Update to latest MODULE.MAR punctuation.
3. Abstract is in one space, not one tab.
4. Add instructions for editting modifications.
5. Add configuration to the environment section.
6. Add instructions to include $FORMAL macro.
7. Add weak/validation section.
8. Add instructions for .ENTRY.
9. Document using intials in maintenance history.
10. Max source line should be 80 columns.
11. Add BLISS template.
12. Add blank after legal notices; add blank after abstract.

[End of SE5R3.RNO]

CHAPTER 5

TEMPLATE

28-Feb-77 -- Rev 3

Included here are instructions for commencing a module of coding, a copy of the template file which is the basis of a new module, and instructions for filling in the template.

5.1 MAKING A NEW ASSEMBLY LANGUAGE MODULE

When you commence the writing of a program in VAX-11 assembler language, you should work from a copy of the template file MODULE.MAR, which contains the proper formatting for assembler programs.

\ MODULE.MAR is on the PDP-11 MIAS system under [202,1]. To commence creation of your own module, simply type

```
PIP filename.MAR=DB0:[202,1]MODULE.MAR
```

where "filename" is your designated file name (nine characters or less). \

MODULE.MAR is normally available under the VAX-11 system by copying from the system assembler library directory

```
$COPY MARS_LIB:MODULE.MAR filename.MAR
```

where "filename" is your designated file name (nine characters or less).

Once your copy of the module template exists you must fill in and/or alter certain information prior to writing code.

A copy of MODULE.MAR is shown at the end of this section. The line numbers in the left margin are for reference in this tutorial; they are not part of the file. Refer to Chapter 4, the Program Structure Overview, for an overview of the various sections. Refer to Chapters 6 and 7 for details on each section. Refer to the Appendix for a sample program.

line 001 Replace "TEMPLATE" with your module name and put a terse (half line) description to the right of the hyphen (-).

line 002 Enter the version number between the two slashes.

line 025 After the colon, enter the name of the facility within which the module resides (e.g., system library, math library, etc.).

line 028 After this line, enter a terse (3 to 6 lines) summary of the functionality of the module, starting each successive line with ";<tab>".

line 030 After the colon, describe the environment within which this module (code) will run, e.g., at what access mode, whether it has interrupts disabled, interrupt level, etc. Include any conditional assembly instructions here.

line 032 Following the first colon and <space>, enter your name; follow the second colon and <space> with the creation date of the module.

line 036 As versions are released, copy this line after the replicated line 037. After the <tab> which is before the comma enter the modifier's name. After the space after the comma enter the modification date. Update this date everytime the file is editted. At the end of the line enter the version number.

line 037 As edits are made after first release, copy this line changing the edit number. At the end of this line describe the edit. If the individual making the change is different from the one responsible for this version, then put the changer's initials in parentheses at the start of the description of each edit.

lines 043 047 051 055 Make appropriate entries in each defined section (reference Chapter 4 if you don't understand the section titles named on template lines 041, 045, 049, and 053).

line 055 Follow this line with a section of weak and validation declarations if any.

line 056 Replace "TEMPLATE_EXAMPLE" with your routine's name and follow the hyphen with a half line description.

line 059 Enter a sufficient description of the function(s) of this routine, starting each successive line with ";<tab>".

line 063 If this module is "called", replace "NONE" with the calling sequence (AP-list). Otherwise give the instruction for invoking this routine.

lines 067 071 075 079 083 087 When applicable, replace "NONE" with the information required by the section titles named on template lines numberred 065, 069, 073, 077, 081, and 085.

line 091 If the routine is CALLed, define its formals by including a $FORMAL macro here.

line 093 Replace "TEMP_EXAMPLE" with your. routine's name

line 092 Preceed the semi-colon with the entry mask or first instruction and adjust the comment appropriately. Or merge lines 093 and 094 into a .ENTRY statement.

line 095 Commence the body of your routine/module, commenting appropriately thoughout. Keep source lines to 80 columns maximum.

line 096 Replace "TEMP_XMPL_EXIT" with your routine's exit location label.

line 097 Replace and/or delete the inappropriate return instruction from this and the succeeding line.

```
001             .TITLE  TEMPLATE -
002             .IDENT  / /
003
004     ;
005     ; COPYRIGHT (C) 1977
006     ; DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASSACHUSETTS 01754
007     ;
008     ; THIS SOFTWARE IS FURNISHED  UNDER A LICENSE FOR USE ONLY ON A SINGLE
009     ; COMPUTER  SYSTEM AND  MAY BE  COPIED ONLY WITH  THE INCLUSION OF THE
010     ; ABOVE COPYRIGHT NOTICE.  THIS SOFTWARE, OR ANY OTHER COPIES THEREOF,
011     ; MAY NOT BE PROVIDED OR  OTHERWISE MADE AVAILABLE TO ANY OTHER PERSON
012     ; EXCEPT FOR USE ON SUCH SYSTEM AND TO ONE WHO AGREES TO THESE LICENSE
013     ; TERMS.  TITLE TO AND  OWNERSHIP OF THE  SOFTWARE  SHALL AT ALL TIMES
014     ; REMAIN IN DEC.
015     ;
016     ; THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT NOTICE
017     ; AND SHOULD  NOT BE CONSTRUED  AS A COMMITMENT  BY DIGITAL  EQUIPMENT
018     ; CORPORATION.
019     ;
020     ; DEC ASSUMES  NO  RESPONSIBILITY  FOR  THE USE OR  RELIABILITY OF ITS
021     ; SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DEC.
022     ;
023
024     ;++
025     ; FACILITY:
026     ;
027     ; ABSTRACT:
028     ;
029     ;
030     ; ENVIRONMENT:
031     ;
032     ; AUTHOR:        , CREATION DATE:
033     ;
034     ; MODIFIED BY:
035     ;
036     ;        , : VERSION
037     ; 01     -
038     ;--
```

```
  <page>
039             .SBTTL  DECLARATIONS
040     ;
041     ; INCLUDE FILES:
042     ;
043
044     ;
045     ; MACROS:
046     ;
047
048     ;
049     ; EQUATED SYMBOLS:
050     ;
051
052     ;
053     ; OWN STORAGE:
054     ;
055
```

```
  <page>
056             .SBTTL  TEMPLATE_EXAMPLE -
057     ;++
058     ; FUNCTIONAL DESCRIPTION:
059     ;
060     ;
061     ; CALLING SEQUENCE:
062     ;
063     ;       NONE
064     ;
065     ; INPUT PARAMETERS:
066     ;
067     ;       NONE
068     ;
069     ; IMPLICIT INPUTS:
070     ;
071     ;       NONE
072     ;
073     ; OUTPUT PARAMETERS:
074     ;
075     ;       NONE
076     ;
077     ; IMPLICIT OUTPUTS:
078     ;
079     ;       NONE
080     ;
081     ; COMPLETION CODES:
082     ;
083     ;       NONE
084     ;
085     ; SIDE EFFECTS:
086     ;
087     ;       NONE
088     ;
089     ;--
090
091
092
093     TEMP_EXAMPLE:
094             ; ENTRY POINT (OR MASK)
095
096     TEMP_XMPL_EXIT:
097             RET
098             RSB
099
100             .END
```

5.2 MAKING A NEW BASIC LANGUAGE MODULE

Details to be supplied.

5.3 MAKING A NEW BLISS LANGUAGE MODULE

When you commence the writing of a program in BLISS, you should work from a copy of the template file MODULE.BLI, which contains the proper formatting for BLISS programs.

\ MODULE.BLI is on the IPC PDP-10 System-F under BLI:. To commence creation of your own module, simply type

```
COPY filename.BLI=BLI:MODULE.BLI
```

where "filename is your designated file name (six characters or less). If the module is not transportable use output file type .B32 to indicate this. \

MODULE.BLI is normally available under the VAX-11 system by copying from the system BLISS directory

```
$COPY BLISS_LIB:MODULE.BLI filename.BLI
```

where "filename" is your designated file name (nine characters or less). If the module is not transportable use output file type .B32 to indicate this.

Once your copy of the module template exists you must fill in and alter certain information prior to writing code.

A copy of MODULE.BLI is shown at the end of this section. The line numbers in the left margin are for reference in this tutuorial; they are not part of the file. Refer to Chapter 4, the Program Structure Overview, for an overview of the various sections. Refer to Chapters 6 and 9 for details on each section. Refer to the Appendix for a sample program.

line 001 Replace "TEMPLATE" with your module name and put a terse (half line) description to the right of the exclamation (!)

line 002 Enter the version number between the two apostrophes. Add any other module switches one per line after line 002.

line 027 After the colon, enter the name of the facility within which the module resides (e.g., system library, math library, etc.).

line 030 After this line, enter a terse (3 to 6 lines) summary of the functionality of the module, starting each successive line with "!<tab>".

line 032 After the colon, describe the environment within which this module (code) will run, e.g., at what access mode, whether it has interrupts disabled, interrupt level, etc. Include any conditional compilation instructions here.

line 034 Following the first colon and <space>, enter your name; follow the second colon and <space> with the creation date of the module.

line 038 As versions are released, copy this line after the replicated line 039. After the <tab> which is before the comma enter the modifier's name. After the space after the comma enter the modification date. Update this date everytime the file is editted. At the end of the line enter the version number.

line 039 As edits are made after first release, copy this line changing the edit number. At the end of this line describe the edit. If the individual making the change is different from the one responsible for this version, then put the changer's initials in parentheses at the start of the description of each edit.

line 046 Enter all routine names defined in this module one per line. Terminate each except the last with a comma. Follow each with a short summary comment (half line). Keep the routines in the order of occurence in the module. Include any routine attributes needed by BLISS.

lines 051 055 059 063 Make appropriate entries in each defined section (reference Chapter 4 if you don't understand the section titles named on template lines 049, 053, 057, and 061).

line 069 Enter all external references made by your routine here one per line. Terminate each except the last with a comma. Include any necessary attributes. Follow each with a terse summary comment of its purpose (half line).

line 070 Replace "TEMP_EXAMPLE ()" with your routine's name and its formal parameter list. Put a terse description of the routine to the right of the exclamation (!). If the above two edits will not fit on this line, keep the comment on this line and place the formal parameter list on the next line. If your routine returns a value, delete the ":NOVALUE" and enter the routine value(s) in the section entitled "ROUTINE VALUE:" (line 091).

line 074 Enter a sufficient description of the function(s) of this routine, starting each successive line with "!<tab>".

line 078 If this module has parameters, replace "NONE" with the list of all parameters in order one per line. For each give a complete description including the passing mechanism in formal notation.

lines 082 086 091 095 When applicable, replace "NONE" with the information required by the section titles named on template lines numberred 080, 084, 088, 089, and 093. Delete whichever of lines 088 and 089 is not applicable.

line 102 List the routine's locals one per line. Follow each with its attributes and a descriptive comment.

line 103 Commence the body of your routine/module, commenting appropriately thoughout. Keep source lines to 80 columns maximum.

line 104 Replace "TEMP_EXAMPLE" with your routine's name.

```
001     MODULE TEMPLATE (       !
002                     IDENT = ' '
003                     ) =
004     BEGIN
005
006     !
007     ! COPYRIGHT (C) 1977
008     ! DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASSACHUSETTS 01754
009     !
010     ! THIS SOFTWARE IS FURNISHED  UNDER A LICENSE FOR USE ONLY ON A SINGLE
011     ! COMPUTER  SYSTEM AND  MAY BE  COPIED ONLY WITH  THE INCLUSION OF THE
012     ! ABOVE COPYRIGHT NOTICE.  THIS SOFTWARE, OR ANY OTHER COPIES THEREOF,
013     ! MAY NOT BE PROVIDED OR  OTHERWISE MADE AVAILABLE TO ANY OTHER PERSON
014     ! EXCEPT FOR USE ON SUCH SYSTEM AND TO ONE WHO AGREES TO THESE LICENSE
015     ! TERMS.  TITLE TO AND  OWNERSHIP OF THE  SOFTWARE  SHALL AT ALL TIMES
016     ! REMAIN IN DEC.
017     !
018     ! THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT NOTICE
019     ! AND SHOULD  NOT BE CONSTRUED  AS A COMMITMENT  BY DIGITAL  EQUIPMENT
020     ! CORPORATION.
021     !
022     ! DEC ASSUMES  NO  RESPONSIBILITY  FOR  THE USE OR  RELIABILITY OF ITS
023     ! SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DEC.
024     !
025
026     !++
027     ! FACILITY:
028     !
029     ! ABSTRACT:
030     !
031     !
032     ! ENVIRONMENT:
033     !
034     ! AUTHOR:        , CREATION DATE:
035     !
036     ! MODIFIED BY:
037     !
038     !       , : VERSION
039     ! 01    -
040     !--
```

```
|    <page>
|  041     !
|  042     ! TABLE OF CONTENTS:
|  043     !
|  044
|  045     FORWARD ROUTINE
|  046             ;               !
|  047
|  048     !
|  049     ! INCLUDE FILES:
|  050     !
|  051
|  052     !
|  053     ! MACROS:
|  054     !
|  055
|  056     !
|  057     ! EQUATED SYMBOLS:
|  058     !
|  059
|  060     !
|  061     ! OWN STORAGE:
|  062     !
|  063
|  064     !
|  065     ! EXTERNAL REFERENCES:
|  066     !
|  067
|  068     EXTERNAL ROUTINE
|  069             ;               !
```

```
    <page>
070     ROUTINE TEMP_EXAMPLE () :NOVALUE =      !
071
072     !++
073     ! FUNCTIONAL DESCRIPTION:
074     !
075     !
076     ! FORMAL PARAMETERS:
077     !
078     !       NONE
079     !
080     ! IMPLICIT INPUTS:
081     !
082     !       NONE
083     !
084     ! IMPLICIT OUTPUTS:
085     !
086     !       NONE
087     !
088     ! ROUTINE VALUE:
089     ! COMPLETION CODES:
090     !
091     !       NONE
092     !
093     ! SIDE EFFECTS:
094     !
095     !       NONE
096     !
097     !--
098
099         BEGIN
100
101         LOCAL
102             ;               !
103
104         END;                            !End of TEMP EXAMPLE
    <page>
105     END                                 !End of module
106     ELUDOM
```

5.4 MAKING A NEW COBOL LANGUAGE MODULE

Details to be supplied.

5.5 MAKING A NEW FORTRAN LANGUAGE MODULE

Details to be supplied.

[End of Chapter 5]

Title: VAX-11 Assembler Software Eng. Commenting -- Rev 3

Specification Status: draft

Architectural Status: under ECO control

File: SE6R3.RNO

PDM #: not used

Date: 28-Feb-77

Superseded Specs: none

Author: P. Conklin, P. Marks, M. Spier

Typist: P. Conklin

Reviewer(s): R. Brender, D. Cutler, R. Gourd, T. Hastings, I. Nassi, S. Poulsen, D. Tolman

Abstract: Chapter 6 gives each piece of the commenting conventions in detail. The items are in alphabetical order. Each item includes references to related topics, gives the background and the rules, and then gives templates and examples.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Original	M. Spier	14-Apr-76
Rev 2	Revised from Review	P. Marks	21-Jun-76
Rev 3	After 6 months experience	P. Conklin	28-Feb-77

Rev 2 to Rev 3:

1. Change column numbers to start with 1 instead of 0.

2. Change CF to FP.

3. Use lowercase English for character names instead of bracketting them.

4. Change example comments to not waste a leading space.

5. Add sections for Author, Calling Sequence, CASE instructions, Comment: group, Completion codes, Condition Handler, Conditional Assembly, Environment statement, Facility statement, Function Value, Functional Description, Inplicit Inputs and Outputs, Interlocked Instructions, Libraries, Listing Control, $LOCAL Macro, Macros, $OWN Macro, Parameters: Input and Output, Program, Queue Instructions, Routine: Order, Side Effects, Signals, String Instructions, Structures, Synchronization: Process, UNWIND, .VALIDATE Declaration, .WEAK Declaration. Add many cross references and sections which are there only to cross reference to another section.

6. Combine abbreviated and detailed history.

7. Add ;++ format.

8. Change symbol definition mechanism from Spier to STARLET.

9. State when dual names might be justified.

10. Clarify when to renumber local labels.

11. Add Call by descriptor.

12. Clarify when <separator> can be four blank lines.

13. Change terminology:
    Routine to Procedure (where appropriate)
    Subroutine to Routine: non-standard
    Definition to Declaration
    Copyright to Legal Notices

14. Move symbol naming rules to Chapter 7. Add references to chapter 8.

15. Change examples to use template formats and text.

16. Put configuration Statement in Environment statement.

17. Document that entry mask must include registers on non-standard subroutines.

18. Change to use .ENTRY.

19. Change [VALUE] to Chapter 8 notation.

20. Move Comment to Comment: Line.

21. Document maintenance numbers. Don't reset them on release.

22. Omit license paragraph for unlicensed software.

23. Add that labels should be meaningful.

24. Give rules for file name.

25. Never use lower case in symbols. Freely use underline. Choose names to suggest attributes.

26. Add that functional description should include critical algorithms.

27. Note that arg list is read only.

28. Include typical .PSECT attributes.

29. Fill in the external symbols section.

30. Split into chapters 6-7.

31. Note that as matter of taste can put a space after the comment delimiter.

32. Make completely language independent.

33. Move contents of completion codes here from naming conventions.

34. Emphasize that legal notices must be on the first page.

35. Emphasize that no keyword is to be omitted; instead use NONE.

36. Emphasize that both the blank comment lines and the blank lines are mandatory.

37. Add support letters to version standard.

38. Add that numbers and letters are not skipped in version, update, or patch.

39. Add update to version standard.

40. Add examples to version standard.

41. Remove attention grabber outdenting.

42. Allow for edit numbers to be facility wide if appropriate.

43. Completion codes have <2:0> of symbol non-zero. Test with CMPV.

44. Add customer version numbers.

45. Move procedure to chapter 7.

46. Move maintenance comments to end of line.

[End of SE6R3.RNO]

CHAPTER 6

COMMENTING CONVENTIONS

28-Feb-77 -- Rev 3

This chapter contains detailed information on commenting conventions. For ease of reference, it is organized alphabetically by topic. Each topic includes references to related topics. Most entries also include examples or sample templates illustrating the specific topic.

The notation <comment delimiter> is used to represent the comment delimiter of the source language. For example, this is a semicolon (";") in assembly language and an exclamation mark ("!") in BLISS and Fortran.

6.1 ABSTRACT

SEE ALSO:
Functional Description

A short three to six line functional description.

6.2 AUTHOR

This is the full name of the initial coder of the module. The full name of each maintainer appears in the modification history. Both appear in the module preface.

6.3 CALLING SEQUENCE

SEE ALSO:
Parameters: Formal
Parameters: Input and Output
Procedure

If this routine follows the procedure CALL standard then the calling sequence is:

```
CALL entry_name (formal parameters)
```

or

```
value = entry_name (formal parameters)
```

The formal parameters should be documented using the notation in the Functional and Interface Specifications chapter.

If this is a non-standard routine, the method of entry should be given as JSB, INTERRUPT, or EXCEPTION. Any parameters passed in registers or on the stack should be given in the input parameters section. Any parameters left on the stack or in registers should be given in the output parameters section.

6.4 COMMENT

SEE ALSO:
Comment: Block
Comment: Documenting
Comment: Group
Comment: Maintenance
Statement: Block

A comment is any text embedded between a <comment delimiter> on the left and the end of the source line on the right.

There is a grey area between the use of too many and the use of too few comments. It is easy to say that there are never enough comments but often there are so many comments that the program text is obscured. In general, comment logically difficult sections of code, structure accesses where it is not clear what is being accessed, and routine invocations, among others. A good rule of thumb is to include a block comment for each block statement.

Above all, strive to comment your program so that anyone can pick it up, read the comments alone and derive a good understanding for what the program does.

In a sense, there are two programs being written; one consisting of code and one consisting of comments. The comment program is written to describe the intent and algorithm of the code. That is, comments are not simply rewordings of the code but are explanations of the overall (gross, if you will) logical meaning of the code.

6.5 COMMENT: BLOCK

SEE ALSO:
Comment: Group
Statement: Block

The block comment precedes a block statement, providing reference documentation for the immediately following sequence of statements. A block comment serves to introduce and describe the functionality of a logical grouping of code. It allows the reader to understand the meaning and effect of the code that follows without having to read the code itself. The following rules apply to block comments:

- The block comment consists of a number of page wide comment lines: The <comment delimiter> is entered, left aligned, in the line's first character position.

- The first line of the block comment is a begin sentinel, of the form "!+" or "!++". The single form should be used for internal documentation such as might appear in a program logic manual. The double form should be used for all functional documentation. If the routine is to be part of a general library, the functional documentation should be in a form suitable for publication, see Functional Description.

- The last line of the block comment is a matching end sentinel, of the form "!-" or "!--".

- The body of the block comment consists of documentary text, separated from the <comment delimiter> by a tab.

- The block comment is immediately followed by a blank line; immediately following the blank line appears the commented block statement.

Example:

```
<skip>
!+
!       This is a block comment.
!-
<skip>
```

6.6 COMMENT: DOCUMENTING

SEE ALSO:
Comment: Block
Module: Preface
Routine: Preface

The documenting comment is a special format block comment that appears in the module preface and in the routine preface. It serves to describe the functionality of the module and/or routine, as that functionality is to be known from the external point of view: what function is performed, what the input and output parameters are, what values are expected, what completion codes returned, and any other relevant functional information.

- The documenting comment consists of a number of page wide comment lines: the <comment delimiter> is entered in the line's first character position.
- The first line of the documenting comment is a begin sentinel, of the form "!++".
- The last line of the documenting comment is an end sentinel, of the form "!--".
- The documenting comment is structured by means of out-dented keywords that are separated from the <comment delimiter> by a single space. These keywords are part of the standard documenting comment's structure and all of them must be included, in the proper sequence.
- If a specified keyword is not applicable, follow it with the word NONE rather than deleting it. This helps the reader by being explicit about the specification.
- For the body of the documenting comment, see Module Preface, or Routine Preface, or the Program Structure Overview chapter.

Example:

```
!++
!       This is an example of a documenting comment.
!
!       It may be either a module preface, or a routine
!       preface: in each case it has a predetermined format,
!       consisting of a sequence of keywords followed by
!       documentation information.
!--
```

6.7 COMMENT: GROUP

SEE ALSO:
Comment: Block

Whenever the attention of the reader should be called to a particular sequence of code, a group comment should be used. This might be in any of the following:

1. When several paths join, note the conditions which cause flow to reach this point.

```
;
; All exceptions converge at this point with:
;
;       ...<register and stack status>
;
```

2. At the top of a loop.

```
;
; Loop looking for a handler to call.
;
```

3. When some data base has been built, such as a complex sequence on the stack.

```
;
; At this point the stack has the following format:
;
;       00(SP) = saved R2
;       04(SP) = number of ...
;        ...
;
```

- The group comment consists of a number of page wide comment lines: the <comment delimiter> is enterred, left aligned, in the line's first character position.

- The first and last lines of the group comment are just a <comment delimiter> and are set off from surrounding code by a blank line before and after the group. Both the blank comment lines and the blank lines are mandatory and help distinguish the comments and code visually.

- The body of the group comment consists of descriptive text, separated from the <comment delimiter> by a space.

- Tabular information is separated from the <comment delimiter> by a tab.

6.8 COMMENT: LINE

A line comment is normally used to explain the meaning of the statement being commented.

A comment is any text following a <comment delimiter>, up to the end of the line. Each and every line of assembly code should be commented.

- The comment is placed on the right hand side of a non-comment line of text.
- All assembly language comments are aligned with the <comment delimiter> in column 41 of the text (5 tabs from left margin).
- The text of the comment is adjacent to the <comment delimiter>.
- If the statement overflows into the comment field, then its comment is preceded by a space, whereas normally it would be preceded by as many tabs as necessary to position the comment starting with column 41.
- If the comment is too long to be contained on a single line, or if the statement was too long to be commented on the same line, then the comment may be placed (or continued) on the following line, placing the <comment delimiter> in the same column as the first line and including a space after it.
- For commenting a multiple-line fragmented statement see statement.

The comment's text should convey the meaning of the associated program text (e.g., instruction MOVAL A,B should be commented "Initialize pointer to first buffer in free area" or such, not "Move the address of A into B".) As a rule of thumb, symbols should not appear in a comment, rather say what the object is or means. If a line of code is totally self evident to the most casual reader then it need not be given redundant commenting text, however it must have a <comment delimiter> (see example). If a comment applies to several successive lines of code, indicate commonality by tagging follow-on lines with comments of the form "!<space> . . .".

As a matter of taste, some coders place a single space after the <comment delimiter>. All modifications to a module should follow the style of the original author. The original source should not be changed to the modifier's style because then a differences listing would be useless.

Example:

```
        STATEMENT                   ;Compute multiple-line function
        STATEMENT                   ; . . .
        STATEMENT                   ; . . .
        OBVIOUS STATEMENT           ;
        STATEMENT                   ;Here we do something new
                                    ; and extend the comment to the
                                    ; next two lines.
        OBVIOUS STATEMENT           ;
        A SOMEWHAT LONG STATEMENT ;And its comment
        A SOMEWHAT LONGER STATEMENT ;And its long comment
                                    ; which continues on
                                    ; additional line(s).
        A VERY VERY VERY VERY VERY VERY LONG STATEMENT
                                    ;And its comment on next line
        A FRAGMENTED            -   ;The statement's comment
             STATEMENT              ; . . .
```

6.9 COMMENT: MAINTENANCE

SEE ALSO:
Author
History: Modification
Version Number

When an existing module is modified (as distinct from "originally coded"), each logical unit of modification is assigned a maintenance number in the detailed current history section of the module preface. Use a new number for each logical unit of modification that is being worked on. The maintenance numbers increase by one, are decimal, and are never reset. It is permissable after a release to bump the number to a round number (such as the next 100s) to make room for SPR fixes to follow the release level. Add a maintenance comment --derived from that number-- to each line of source code that is affected. There are two reasons for having maintenance comments:

1. The modifications may well be distributed all over the module. The maintenance comment enables you to find all the places where a correction of a single functional problem was made. This is especially useful if the correction has to be further corrected by someone other than the original modifier and/or if it has to be understood by the software specialist in the field.

2. All too often it happens that as we correct bug "B", we innocently modify an instruction which was the correction for a previous bug "A". Bug "B" is fixed at the expense of the reappearance of bug "A" (or one of its relatives). If modification of a program leads you to the modification of a line that already has a maintenance comment, then find out (from the detailed current history) who the modifier was, consult that person, and exercise extreme caution in effecting your modification.

In many cases the edit numbers may be assigned consistently across all modules in a facility. In this case, the module defining the facility's version number should have a full maintenance history and the others should include only module specific changes.

The following rules apply to maintenance comments:

- The maintenance comment consists of a <comment delimiter> followed by a code letter, followed by a maintenance number.
- The code letter may be

  A - this line was ADDED to the text

  D - this line was DELETED. In this case, effect the "deletion" by commenting the line out. Place a <comment delimiter> in the first character position of the line, marking it as a candidate for future physical deletion.

  M - This line was MODIFIED.

- The maintenance comment is placed after the line's regular comment at column 80

  ```
  !Regular comment !<maintenance_comment>
  ```

- If the modified line already has an existing maintenance comment, then add the new one in front of the existing one

  ```
  !Regular comment !<new_mc>!<previous_mc>
  ```

Example:

The maintenance number is assigned in the detailed current history section of the module preface, as follows:

```
! 02    -  SPR #4711: describe the SPR problem
```

The number is now used in maintenance comments for all lines of text affected by the modification called for by SPR #4711:

```
        MODIFIED STATEMENT              !Statement's comment    !M02
        ADDED STATEMENT                 !statement's comment    !A02
!       DELETED STATEMENT               !Statement's comment    !D02
```

NOTE: If the statement is a multiple-line one, make sure to place maintenance comments (or effect a "commenting out" deletion) on all component lines of the statement.

6.10 COMPLETION CODES

The most reliable means for indicating a software detected exception condition occurring in a called procedure is for the called procedure to return a condition value as a function value and for the caller to check the return value for TRUE or FALSE. TRUE is bit 0 set and FALSE is bit 0 cleared. TRUE means that the requested operation was performed successfully; FALSE means an error condition occurred; in both cases, the rest of the value is a condition value. Thus, most procedures are written as functions, rather than subroutines. If it is necessary to indicate an exceptional situation without returning a value, then generate a call to LIB$SIGNAL, see Signals.

The low order three bits, taken together, represent the severity of the error. Severity code values are:

0	Warning
1	Success
2	Error
3	Reserved
4	Severe Error
5-7	Reserved

Bits <31:16> indicate the facility, see the Naming Conventions chapter. Bits <15:3> distinguish distinct conditions or system messages within the facility. Bits <2:0> can vary for a given condition depending upon environment, condition handling, etc. Status codes are expressed in symbolic names in the format:

```
fac$ mnemonic
```

Return status values can be tested by testing the low-order bit of R0 and branching to an error checking routine if the low bit is not set, in the assembler as follows:

```
BLBC    R0,errlabel
```

The error checking routine may check for specific values. It must always ignore <2:0> when checking for a particular condition because <2:0> can vary depending upon the severity in the current environment. For example in assembly language, the following instruction checks for an illegal event flag number error condition:

```
CMPV    #3,#29,R0,#<SS$_ILLEFC@-3>
```

Successful codes other than SS$ NORMAL are defined. In some cases, a successful return includes information about the previous status of a resource. For example, the return SS$ WASSET from the Set Event Flag ($SETEF) system service indicates that the requested flag was already set when the service was called.

6.11 CONFIGURATION STATEMENT

SEE ALSO:
INCLUDE Files
Module: Preface

The configuration statement is part of the environment statement in the module preface, and serves to indicate to the programmer how the module is to be assembled. The module may be part of a large system with a system-wide conditional assembly arrangement. It may also have its own peculiar conditional assembly requirements, either alone or in conjunction with system-wide conventions.

State the name(s) of the include file(s) containing conditional assembly parameters (if any). State the conditional assembly variables affecting this module. If the variables are peculiar to this module, state the values that they may assume, and what these values mean.

Example:

```
! ENVIRONMENT:
!
!       This module may be assembled with various parameters
!       changed. This is done by supplying a special copy
!       of the macro $FAC_CHANGE_DEF with the changed symbols in
!       it as a library file. The symbols which can be changed
!       are the default lines per page (DEF_LINES_PPAGE) which
!       is normally 55, and the maximum line width (MAX_LINE_WIDTH)
!       which is normally 132.
!
```

6.12 ENVIRONMENT STATEMENT

SEE ALSO:
Configuration Statement

This paragraph gives any special environmental assumptions which a module may make. These include both compilation assumptions such as configuration files and execution time such as hardware or software environments. For compile time environments, see Configuration Statement.

For execution time environment describe any situations which the module may assume. For example, it may assume that the hardware is a single processor, or that this module is always invoked with interrupts disabled. The module might assume that it runs only in user mode, that ASTs are disabled, or that storage allocation is handled by the standard procedure library. In general, document here anything out of the ordinary which the module assumes about its environment.

6.13 EXCEPTIONS

SEE Signals

6.14 FACILITY STATEMENT

This section of the module preface gives the full name of the facility of which this module is a part. See the Naming Conventions chapter for a list of the facilities.

6.15 FUNCTIONAL DESCRIPTION

The functional description section of the module and routine prefaces should describe the purpose of the module or routine and should document its interfaces precisely and completely

The functional description should also include the basis for any critical algorithms used. This should include literature references when available. For example, specify why a particular numerical algorithm is used in the math library or why a particular way of sorting was chosen.

The functional description appears in one of three places:

- As a self-contained short description on the first page of the module and routine prefaces.
- As the second or more page(s) of the module and routin prefaces. In this case an abstract appears on the first page.
- As a separate functional specification. In this case an abstract appears on the first page of the module and routine prefaces and a reference to the specification is included.

Example:

```
!++
! FUNCTIONAL DESCRIPTION:
!
! EXP(X) is computed using the following approximation technique:
!
!       If X > 88.028 then overflow
!       If X <= -89.416 then EXP(X) = 0.
!       If |X| < 2**-28 then EXP(X) = 1.
!
! Otherwise,
!
!       EXP(X) = 2**Y * 2**Z * 2**W
!
!       where
!               Y = integer(X*log2(E))
!               V = frac(X*log2(E)) * 16
!               Z = integer(V)/16
!               W = frac(V)/16
!
!               2**W = (P + W*Q) / (P - W*Q)
!
!               P and Q are first degree polynomials in W**2. The
!               coefficients of P and Q are drawn from Hart #1121.
!
!       Powers of 2**(1/16) are obtained from a table.  All
!       arithmetic is done in double precision and then rounded
!       to single precision at the end of calculation. The relative
!       error is less than or equal to 10**-16.4.
!
```

6.16 FUNCTION VALUE

SEE ALSO:
Completion codes
Functional and Interface Specification chapter

A function value is returned in register R0 if representable in 32 bits and registers R0 and R1 if representable in 64 bits. If the function value cannot be represented in 64 bits, one of the following mechanisms is used to return the function value:

1. If the maximum length of the function value is known, the calling procedure can allocate the required storage and pass a pointer to the function value storage as the first argument.

   This method is adequate for CHARACTER functions in Fortran and VARYING strings in PL/1.

2. The called procedure can allocate storage for the function value and return in R0 a pointer to a descriptor of the function value.

   This method requires a heap (non-stack) storage management mechanism.

Procedures, such as operating system CALLs, return a success/fail value as a longword function value in R0. Success returns have bit 0 of the returned value set (Boolean true); failure returns have bit 0 clear (Boolean false). The remaining 31 bits of the value are used to encode the particular success or failure status.

6.17 HISTORY: MODIFICATION

SEE ALSO:
Author
Comment: Maintenance
Module: Preface
Version Number

The detailed modification history is a section of the module preface. An entry is logged for each logical functional modification of the module. For example, if the module is a terminal driver, and bug reports state that sometimes interrupt handling is incorrectly masked and also that deleted characters are handled incorrectly, then these will be given TWO separate log entries: one entry for the interrupt problem, one for the delete problem.

Each log entry is assigned a maintenance number. The maintenance numbers begin with "1" and grow by unit increments. The log entry specifies the maintainer's name, and a description of the problem requiring maintenance.

If a problem that was thought fixed is reopened for further fixes, or if a modification changes hands from one programmer to another, a new log entry (having a new maintenance number) is made.

The maintenance numbers are used to affix maintenance comments at all the places that were modified. This way, it becomes possible for anyone to look at a maintained piece of software (especially anyone in the field) and reconstruct what has happened.

Periodically, at the discretion of the appropriate supervisor, old detailed current history log entries may be deleted, together with their corresponding documenting comments (and lines marked for deletion). It is advised that the deletion not be made until the software has proven itself in the field.

6.18 IMPLICIT INPUTS AND OUTPUTS

SEE ALSO:
  Parameters: Formal
  Parameters: Input and Output
  Side Effects

These sections of a routine preface should include all locations in global or own storage which are read or written by the routine. Any locations which are addressed by parameters should not be documented in these sections, see Parameters: Formal, and Parameters: Input and Output.

ADDITIONAL SPECIFICS TO BE SUPPLIED

6.19 LEGAL NOTICES

A standard DEC copyright statement must always appear on the first page of every source file. It is part of the module preface. The legal notices must be part of the original program text, so that they will be plainly stated on any DEC program listing (regardless of whether the listing was produced by a language processor or was directly printed from the source).

- The legal notices may undergo revision. Make sure that you use the proper current version.
- The legal notices are always in upper case to bring emphasis to them.
- When developing a new module, the year stated is the year of the first release, not of the first coding.
- When modifying an existing program that has legal notices,

  (1) Verify the statements' validity, and

  (2) Add the year of modification to the year stated by the existing copyright statement; DO NOT update that existing year: add the current one (if different), separating it from the last date with a comma (",").

The legal notices are of the following form:

```
!
! COPYRIGHT (C) 1977
! DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASSACHUSETTS 01754
!
! THIS SOFTWARE IS FURNISHED UNDER A LICENSE FOR USE ONLY ON A SINGLE
! COMPUTER SYSTEM AND MAY BE COPIED ONLY WITH THE INCLUSION OF THE
! ABOVE COPYRIGHT NOTICE. THIS SOFTWARE, OR ANY OTHER COPIES THEREOF,
! MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY OTHER PERSON
! EXCEPT FOR USE ON SUCH SYSTEM AND TO ONE WHO AGREES TO THESE LICENSE
! TERMS. TITLE TO AND OWNERSHIP OF THE SOFTWARE SHALL AT ALL TIMES
! REMAIN IN DEC.
!
! THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT NOTICE
! AND SHOULD NOT BE CONSTRUED AS A COMMITMENT BY DIGITAL EQUIPMENT
! CORPORATION.
!
! DEC ASSUMES NO RESPONSIBILITY FOR THE USE OR RELIABILITY OF ITS
! SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DEC.
!
```

The license paragraph should be omitted from software which DEC does not license (e.g., distributed through DECUS or not owned by DEC).

6.20 MODULE

The module is a single body of text that is assembled as a unit. The module is normally part of a larger program or facility that is created by linking all of the component modules object code.

There must be some self evident identity justifying the module's existence. That is to say, the module is not just an arbitrary concoction of code, but a self evident unit of code. Typically, the module consists of either:

- A single function or database, or
- A collection of related functions (e.g., all conversion routines) each of which would be too small for an independent module.

The word "module" is used in its hardware sense: a "black box" unit that may be attached or detached, plugged in or out. In order to have this desirable property of a "plug-in module", the module's interface has to be as clean as possible Use formal argument carrying calls for all routines in the module, avoid all functional side effects. In the case of non-standard interfaces, try using a "standard" non-standard interface (i.e., an interface that is uniform within the program of which the module is part.)

The module should contain

THE FUNCTIONALITY,
THE WHOLE FUNCTIONALITY
AND NOTHING BUT THE FUNCTIONALITY!

Then, if it is known that a certain functionality is wholly and exclusively localized to a given module, it becomes possible to replace the module by a more efficient one, or selectively link it into the larger program depending on the runtime requirements. The ability to do this is more useful and important than any local efficiency "hackery' that would jeopardize the module's functional identity. When in doubt, place each routine in a separate module. Combine a few routines primarily when doing so allows own storage to be used rather than global storage. Never combine many routines.

6.21 MODULE: DATA SEGMENT

SPECIFICS TO BE SUPPLIED

6.22 MODULE: FILE NAME

Each module exists as a distinct source text file. The name of the file reflects the module's functionality and also the larger facility of which it may be part.

The module is stored in a filename which is the non-facility part of the name, see the Naming Conventions chapter. The file type is the standard one for the source language. There is no special significance to the file generation version (i.e., it need not match the edit number or increase from release to release). The file is stored in a directory which corresponds to the facility.

6.23 MODULE: PREFACE

The module preface provides uniform documentation of the module. It contains certain control items (TITLE and IDENT) which are needed by the linker, as well as the standard DEC copyright statement needed for the protection of DEC's legal ownership rights. Apart from these items, the module preface contains all of the information that might be needed in order to know what the module is and does, what the module's history is, and how the module relates to the larger software product of which it is a part. This documentation should include the design basis for any critical algorithms.

The module preface is described and illustrated in the Program Structure Overview chapter. All module prefaces should rigorously adhere to the standard format, so that they can be processed mechanically. For example, it should be possible to extract information from the module preface in order to compile technical documentation. This can only be achieved if the module preface is of uniform syntactical construction.

6.24 PARAMETERS: FORMAL

SEE ALSO:
Implicit Inputs and Outputs
Parameters: Input and Output
Procedure
Routine: Preface

The VAX-11 hardware has a built-in advanced call/return mechanism with provision for automatic argument passing. The caller specifies a list of arguments. The called procedure expects parameters which correspond one-to-one to the caller's arguments.

The procedure's parameters will be bound with the arguments of each caller, at the moment of call. They are known as "formal parameters" because they have no identity (i.e., specific memory address) on their own, but assume the identity of whatever arguments the present caller chooses to supply.

The argument list pointer AP always points at the base of the caller-supplied argument list.

6.25 PARAMETERS: INPUT AND OUTPUT

SEE ALSO:
Calling Sequence
Implicit Inputs and Outputs
Parameters: Formal

These sections of a routine preface should include any parameters passed on the stack or in registers. Any parameters whose locations are addressed directly in own or global storage should be documented as implicit inputs and outputs. Any parameters which are passed via the CALL AP-list mechanism should be documented as formal parameters in the calling sequence.

ADDITIONAL SPECIFICS TO BE SUPPLIED

6.26 PROGRAM

SEE ALSO:
Module
Procedure

An executable program consists of one or more object modules which have been combined and formatted in such a way to be interpretable by an operating system and its hardware.

The following general rules govern the division of program information into modules:

- There is exactly one module within the program, termed the main module, where execution of the program begins.
- If need be, any storage that is referenced by more than one module (i.e., global storage) is declared in one or more modules whose sole purpose is to declare/allocate global storage.
- Separate program operations are divided into modules that contain all of the routines related to a single capability. Examples are symbol table management, binary output generation, and so on.
- Module size is kept moderate in order to facilitate incremental modification and to keep the system resources needed for compilation within reasonable limits.
- When in doubt, place each routine in a separate module.
- Even the main routine is CALLed by an outer environment. Typically this environment is the command interpretter.

6.27 ROUTINE: PREFACE

The routine preface provides uniform documentation of the routine, for the following purposes:

- External functional appearance: From the external point of view, the routine is a "large scale" instruction, performing a high-level function. Like any other instruction, it has to be invoked in a precisely predetermined way and be supplied with arguments of a predetermined form and nature. The routine preface provides exact specifications of the anticipated arguments.

- Runtime behavior: The routine's behavior is dependent on both its input parameter value(s) and possible environmental conditions. For example, the routine OPEN_FILE is dependent on being given a valid file name parameter, as well as on the existence and/or protection of the specified file. It may fail for either reason. The routine's preface specifies the behavior of the routine in case of functional failure: specifies the completion codes that may be returned.

- Side effects: The routine's execution may have functional side effects that are not evident from its invocation interface. Such side effects are documented in the routine's preface. This would include changes in storage allocation, process status, file operations, and signals.

- Functional specification: The short functional specification incorporated in the routine preface should be sufficiently logical and lucid to enable the casual reader to get a fairly accurate idea of what the routine does. This specification should NOT describe HOW the algorithm operates; for that one can read the code (an exception being certain esoteric or elusive effects which otherwise would remain unnoticed from reading the code). The functional specification should explain WHAT the routine's execution accomplishes.

The routine preface is described and illustrated in the Program Structure Overview chapter. All routine prefaces should rigorously adhere to the standard format, so that they can be processed to compile technical documentation.

REMEMBER: It is the CALLed routine which specifies how it is to be called! It is the CALLER'S RESPONSIBILITY to invoke the routine in the precise manner in which it expects to be invoked! The routine preface provides all the necessary information needed in order to determine how a routine is to be called.

6.28 SIDE EFFECTS

SEE ALSO:
Implicit Inputs and Outputs
Signals

This section of the routine preface describes any functional side effects that are not evident from its invocation interface. This would include changes in storage allocation, process status, file operations, and signals. In general, document here anything out of the ordinary which the routine does to its environment. If its effect is to modify own or global storage locations, document them as implicit outputs rather than as side effects.

ADDITIONAL SPECIFICS TO BE SUPPLIED

6.29 SIGNALS

SEE ALSO:
Completion Codes
Condition Handler
Side Effects
UNWIND

The most reliable means for indicating a software detected exception condition occurring in a called procedure is for the called procedure to return a completion code as a function value and for the caller to check this return value for TRUE or FALSE. If it is necessary to indicate an exceptional situation without returning a value, then generate a CALL to LIB$SIGNAL to signal the exception. See Appendix D of the System Reference Manual for details on signalling. Current practice is to use this for indicating the occurrence of hardware detected exceptions and for issuing system messages.

When a language or user wishes to issue a signal, it calls the standard procedure LIB$SIGNAL. This routine searches the stack for condition handlers. By convention, the top of the stack normally contains a handler which uses the condition value argument to retrieve a system message from the system message file. It then issues the message to the standard output device. The default handler then takes the default action depending on bit <0> of the condition value. If the bit is set (TRUE) then execution is continued following the call to LIB$SIGNAL. If the bit is clear (FALSE) then execution is terminated and the condition value is available to the command processor to control execution of the command stream.

When a language or user wishes to issue a signal and never continue, it calls the standard procedure LIB$STOP. This routine is identical to LIB$SIGNAL except that execution never continues.

Thus, the rules for handling exceptional cases in a procedure are very simple:

1. Normally return a completion code to the caller as an indicator of failure.

2. If this is not possible or desirable, issue a message by calling either LIB$SIGNAL or LIB$STOP. Call the former if the signalling procedure can meaningfully continue and the latter if the signalling procedure cannot continue.

3. If the normal situation after issuing the message is to continue execution, then the condition value should have the low order bit set. If the normal situation is to terminate after the message, then the low order bit should be clear.

In addition, the routine LIB$SIGNAL preserves all registers including R0 and R1. Thus, it is possible to insert debugging or tracing signals in a routine without alterring its register usage.

6.30 VERSION NUMBER

SEE ALSO:
Comment: Maintenance
History: Modification
IDENT Statement
Module: Preface

The VAX-11 standard version number is used to provide unique identification of all pre-released, released and inhouse software. It is used both at the module and the facility level. When used for modules, the ident represents the last change made to the module. For facilities which are always bound together such as a compiler, the ident of the module containing the start address is also used as the ident of the facility. The facility (start module) ident must be changed whenever the ident of any component module changes even if the component comes from a library.

The version number is a compound string constructed of the concatenation of the following discrete items:

<support> <version> . <update> - <edit> <patch>

where:

- <support> is a single capital letter (or null) identifying the support level of the program:

  B benchmark version
  D demonstration version
  S special customer version
  T field test version
  V released or frozen version
  X unsupported experimental version

  Typically this letter is omitted from the module ident since it more reflects the program as a whole than any of its modules.

- <version> is a decimal leading zero-suppressed number, starting with "0" and progressing by positive unit increments. Numbers are never skipped. "0" is used prior to the first release. "1" designates the first release, etc. The version identifies the major release, or generation, or base level of a program. It is incremented at the discretion of the responsible supervisor whenever the software has undergone a significant or major change. The module version is incremented upon the first edit after a release so that it reflects the next release.

- <update> if present is a period followed by a single decimal digit indicating a minor release containing internal changes but no significant external changes. Digits are never skipped. Null designates the major release. "1" designates the first update, etc. <update> is cleared when <version> is changed.

- <edit> if present is a minus sign followed by a decimal leading zero-suppressed maintenance number, starting with "1" and progressing by positive unit increments. Numbers may be skipped but may never be lower than that that of a previous edit. The edit identifies any alteration of the source code. It is incremented on every change even if modification history comments are not being kept. Whether <edit> is cleared on release is TO BE SPECIFIED.

- <patch> if present is a single capital letter identifying an alteration to the program's binary object form. The patch character begins with "B" and may be incremented up to "Z", whenever a set of patches is released. This never appears in the source of a module. <patch> is cleared whenever <version> or <update> is changed.

Customers making changes to DEC produced software are advised to follow similar procedures. Customer numbers should be designated by appending a customer version and edit number to the DEC number and putting it inside square brackets.

Examples:

```
PIP/X3              experiment before third release of PIP

LINK/V5.2-329       released second update to version 5 of LINK;
                    edit level is 329

LOGIN/V0.3-27       frozen version of LOGIN; part of base level 3
                    prior to initial release (during initial development);
                    edit level is 27

RUNOFF/V10.2-527[7-93]
                    seventh customer version of RUNOFF based on the
                    second update to the tenth DEC version; DEC
                    edit level is 527; customer edit level is 93
```

[End of Chapter 6]

Title: VAX-11 Software Eng. Assembler Formatting -- Rev 3

Specification Status: draft

Architectural Status: under ECO control

File: SE7R3.RNO

PDM #: not used

Date: 28-Feb-77

Superseded Specs: none

Author: P. Conklin, P. Marks, M. Spier

Typist: P. Conklin

Reviewer(s): R. Brender, D. Cutler, R. Gourd, T. Hastings, I. Nassi, S. Poulsen, D. Tolman

Abstract: Chapter 7 gives each piece of the assembler formatting and usage conventions in detail. The items are in alphabetical order. Each item includes references to related topics, gives the background and the rules, and then gives templates and examples.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Original	M. Spier	14-Apr-76
Rev 2	Revised from Review	P. Marks	21-Jun-76
Rev 3	After 6 months experience	P. Conklin	28-Feb-77

Rev 2 to Rev 3:

1. Split from chapter 6; see chapter 6 for earlier change history.
2. Correct comment column in all examples.
3. Add examples to .IDENT.
4. Add an ident comment to include files.
5. Add common .PSECT statements to description.
6. Limit source line length to 80 columns.
7. Local labels go to 65535.
8. Eliminate single exit point.
9. Change non-CALL to non-standard.
10. Move procedure here from chapter 6.

[End of SE7R3.RNO]

CHAPTER 7

ASSEMBLER FORMATTING AND USAGE

28-Feb-77 -- Rev 3

This chapter contains detailed information on formatting standards, and instruction usage. For ease of reference, it is organized alphabetically by topic. Each topic includes references to related topics. Most entries also include examples or sample templates illustrating the specific topic.

7.1 CALL INSTRUCTIONS

SEE Procedure

7.2 CASE INSTRUCTIONS

SPECIFICS TO BE SUPPLIED

7.3 CONDITIONAL ASSEMBLY

SEE ALSO:
  Configuration Statement

In the example of the configuration statement, the normal definition library for this compilation is assumed to contain a dummy macro named $FAC_CHANGE_DEF which can be superseded by a user supplied one. The default values are defined only if the symbols are not defined by the time the macro has been expanded. This is done in the source file in the equated symbols section:

```
;
; INCLUDE FILES:
;

        $FAC_CHANGE_DEF

;
; EQUATED SYMBOLS:
;

        .IIF NDF DEF_LINES_PPAGE, DEF_LINES_PPAGE=55
        .IIF NDF MAX_LINE_WIDTH, MAX_LINE_WIDTH=132
```

7.4 CONDITION HANDLER

SEE ALSO:
Completion Codes
Signal
UNWIND

For the primary purpose of handling hardware detected exceptions, the VAX-11 system supplies a mechanism for the programmer to specify a handler function to be called when an exception occurs. This mechanism may also be used for software detected exceptions.

Each procedure activation has a condition handler potentially attached to it via a longword in its stack frame. Initially, the longword contains 0, indicating no handler. A handler is established by moving the address of the handler's procedure entry point mask to the establisher's stack frame.

In addition, the operating system provides two exception vectors at each access mode. These vectors are available to declare handlers which take precedence over any handlers declared at the procedure level. These are used, for example, to allow a debugger to monitor all exceptions, whether or not handled. Since these handlers do not obey the procedure nesting rules, they should not be used by procedure library code. Instead, the stack based declaration should be used.

When a condition handler gets control, it is given several arguments. One of these indicates whether the exception occurred in "this" handler's establisher or in a descendant of it. Another argument is the specific condition which occurred. This is in the same form as a completion code and bits <31:3> identify the specific condition.

For further details, see Appendix D of the System Reference Manual. It describes in detail when the handler is called and what its formal parameters are. In addition, the options of the handler are detailed.

7.5 DECLARATION: EQUATED SYMBOLS

SEE ALSO:
Module: Preface
Parameters: Formal
Routine: Preface
Variables: Stack Local

Define the equated symbols in the proper place as indicated by the module preface and the routine preface sections.

- Define the equated symbols in alphabetic order if there is no other logical order indicated.
- If there is some indicated logical ordering, it may be because of either of the following reasons:
  - Equated symbol A is used in the definition of equated symbol B, hence must have been defined prior to B.
  - Equated symbols are used to define a based structure, and have to be defined in the order dictated by the structure definition. In this case precede the structure definition with a block comment stating that this is a logical structure definition, and how it is going to be used. See block comment.
- The equated symbols are defined one per line. The symbol is defined left aligned in the first character position of the line. The definition line has a comment explaining the nature and use of the symbol.
- A local equated symbol is defined by means of the "=" operator. A global equated symbols is defined by means of the "==" operator.

Example:

```
;
;       Definition of equated symbols
;

CARRET=13                               ;Carriage return character
FORMFEED=12                             ;Form feed character
LINEFEED=10                             ;Line feed character
```

For an example of a structure definition, see structures.

7.6 DECLARATION: VARIABLES

SEE:
  $OWN Macro
  .PSECT Statement
  Structures
  Variables: Stack Local

7.7 DESCRIPTOR

SEE:
  Parameters: Formal
  Functional and Interface Specifications chapter

7.8 EXPRESSIONS

The assembler allows for assembly-time expressions. Typically you will use them when accessing data structures that are relative to some base address. An important reason for using symbols in expressions is so that all references will appear in a cross reference listing.

- Avoid using absolute numbers in your expressions, especially numbers that are liable to change in the future. Define suitable equated symbols: you will both enhance the readability of your code and facilitate the modification of such numbers without having to change any of your code.

- When you have recurring expressions, then further equate the expression itself with a mnemonically meaningful symbol.

- The assembler expression evaluator does not know of operator precedence. Expressions are evaluated in a strict left-to-right order. Make use of angle brackets "< >" (the assembler's notation for algebraic parentheses) to resolve any ambiguity in evaluation precedence.

7.9 $FORMAL MACRO

SEE Parameters: Formal

7.10 .IDENT STATEMENT

SEE ALSO:
Version Number

The .IDENT statement is the second statement of the module. It has, as its parameter, the current version number and edit level of the module separated by a minus ("-"). These numbers correspond to the last entry in the module's modification history.

Example:

```
        .IDENT  /3-47/          edit 47; used in version 3
        .IDENT  /6.2-295/       edit 295; used in version 6.2
```

7.11 INCLUDE FILES

The purpose of INCLUDE files is to centralize in one place declarations and definitions that are common to multiple modules. Data structure declarations, macro declarations, and constant declarations are the principal contents of INCLUDE files.

INCLUDE files are usually in the form of a macro library. In this case, it contains only macro declarations. In order to include structure declarations and constants, the appropriate definitions are included in a structure definition macro. When this macro is called, all the symbols relating to that structure become defined. Refer to the Symbol Naming Conventions chapter for the form of these symbols and the macro name.

The source for INCLUDE files consist of the following:

1. A title comment

```
; file-name - short description
```

2. An ident comment

```
; .IDENT /6.2-295/
```

3. A full set of legal notices.

4. The rest of a module preface to describe the file.

5. The text of the INCLUDE file. The text conforms to the formatting rules for declarations.

6. An end comment

```
; file-name - LAST LINE
```

7.12 INTERLOCKED INSTRUCTIONS

SEE Synchronization: Process

7.13 LABEL

SEE ALSO:
Label: Local
Procedure: Entry
Relative Addressing
Symbol

A label is a symbol which names a statement. The label is delimited by a colon.

- A label should be meaningful in that it should convey some information about the purpose of the block it precedes.
- Left align all labels in column one of the source text.
- A label should be placed on a line of its own (i.e., not on same line as the labelled item), and be commented unless it is a local label. The comment should explain the logical meaning of the label, and under what circumstances execution reaches the label.
- A statement may sometimes have several (synonymous) labels, in which case they are placed on subsequent lines, and commented individually. NOTE: This practice is generally discouraged. Generally, each item in the program should have at most a SINGLE name. Only in rare cases will a single item justifiably require several names, such as when two distinct functions have been combined.
- The labelled statement is placed on the immediately following line.

Example:

```
   A_LABEL:                              ;Result is Negative
        STATEMENT                      ;Statement's Comment
ANOTHER_LABEL:                         ;Used if GEN SWITCH = OFF
SYNONYMOUS_LABEL:                      ;Used if GEN SWITCH = ON
        STATEMENT                      ;Statement's Comment
```

7.14 LABEL: GLOBAL

SEE ALSO:
  Declaration: Equated Symbols
  Symbol: Global

A global label is declared by means of the double colon "::" operator or in an entry operator.

Example:

```
PRINT::                                 ;Global print routine
        .WORD   ^M<register list>       ;Register save mask
```

or

```
        .ENTRY  PRINT,^M<register list> ;Global print routine
```

7.15 LABEL: LOCAL

SEE ALSO:
LSB: .ENABL/.DSABL

The local label is a special purpose construct "n$:" where "n" is a decimal constant. The value of an explicitly stated "n" may be in the range of integers 1 through 65535 (decimal). Local labels have a limited scope of reference defined by (non-local) label brackets, or by an explicit local symbol block.

- The local label is left aligned in column one of the source text, on the same line as its named statement.
- Local labels serve as necessary but otherwise mnemonically meaningless statement identifiers within a block statement.
- Local labels SHOULD NOT BE USED other then for flow of control identification within a block statement! DO NOT use local labels throughout logically unrelated sequences of statements. If need be, label block statements mnemonically in order to force a change of scope for the following local labels.
- Local labels need be unique only within their given scope; a local label's name may be reused within a new scope.
- Always number your local labels sequentially, from "10$:" upwards by increments of 10 in the order of appearance.
- When inserting a new local label between two existing ones, give it a number within the range of the two existing labels: insert "15$:" between "10$:" and "20$:", "17$:" between "15$:" and "20$:".
- The numbers should be multiples of ten at first release, and should be renumbered on any release which makes extensive changes. They should not be renumbered in the course of maintenance patches or updates.

LABEL: LOCAL

Example (correct):

```
   LABEL1:                                  ;Begin local label scope
        STATEMENT                       ;
10$:    STATEMENT                       ;
20$:    STATEMENT                       ;

   LABEL2:                                  ;Begin local label scope
10$:    STATEMENT                       ;
```

Example (incorrect):

```
   LABEL1:                                  ;Begin local label scope
50$:    STATEMENT                       ;First label not "10$:"
60$:                                    ;Free standing local label
        STATEMENT                       ;
30$:    STATEMENT                       ;Decreasing label number
120$:   STATEMENT                       ;Increment larger than 10
```

7.16 LIBRARIES

SPECIFICS TO BE SUPPLIED

7.17 LISTING CONTROL

SPECIFICS TO BE SUPPLIED

7.18 $LOCAL MACRO

SEE Variables: Stack Local

7.19 LSB: .ENABL/.DSABL

SPECIFICS TO BE SUPPLIED

7.20 MACROS

SPECIFICS TO BE SUPPLIED

7.21 $OWN MACRO

SEE ALSO:
Structures

SPECIFICS TO BE SUPPLIED

7.22 PARAMETERS: FORMAL

SEE ALSO:
Implicit Inputs and Outputs
Parameters: Input and Output
Procedure
Routine: Preface
Structures
Variables: Stack Local

The VAX-11 hardware has a built-in call/return mechanism with provision for automatic argument passing. The caller specifies a list of arguments. The called procedure expects parameters which correspond one-to-one to the caller's arguments.

The procedure's parameters will be bound with the arguments of each caller, at the moment of call. They are known as "formal parameters" because they have no identity (i.e., specific memory address) on their own, but assume the identity of whatever arguments the present caller chooses to supply.

The argument list pointer AP always points at the base of the caller-supplied argument list. The first argument list element is accessed as 1*4(AP), and the Nth as N*4(AP). Rather than address those arguments absolutely, define each procedure parameter as a symbolically equated offset relative to AP.

The definition of symbolic formal parameters is made at the end of the routine preface:

```
        $FORMAL         <-              ;
PAR1,-                                  ;PAR1.at.mf is symbolic name
PAR2,-                                  ;PAR2.at.mf is symbolic name
   .                                            .
   .                                            .
PARn>                                   ;PARn.at.mf is symbolic name
```

where the .at.mf specifies the access type, the data type, the passing mechanism, and the passing format. See the Functional and Interface Specifications chapter for more details.

In the body of the procedure, you now refer to the parameters symbolically:

- Call by reference: refer to the value of the Nth parameter by the form @PARn(AP). Refer to the ADDRESS of the Nth parameter by the form PARn(AP).

- Call by value: refer to the value of the Nth parameter by the form PARn(AP). You cannot make any meaningful reference to the parameter's address. Warning: the argument list is read only.

- Call by descriptor: the descriptor is referenced as in call by reference. The structure typically has a more specific referencing algorithm.

Giving the formal parameters symbolic names has the following advantages:

- The code is readable. The notation @FILNAM(AP) is more meaningful than the notation @12(AP).

- If it so happens that the procedure's interface has to be changed, and what used to be the Nth argument now is the N+Ith argument, only the parameter definitions have to be revised; the referencing code itself remains unaffected. Moreover, any such modification is made within the routine preface's documenting comment and is thus automatically reflected in the module's documentation.

- The symbols appear in a cross reference listing.

7.23 PROCEDURE

SEE ALSO:
Parameters: Formal
Routine: Entry: Multiple
Routine: non-standard
Routine: Order

The procedure is a body of code that is CALLed by some other body of code, or recursively by itself, to perform a certain function. The procedure has a certain functional behavior which may be controlled through caller supplied arguments. To the procedure, the caller's arguments are locally known as formal parameters; the procedure does not have to know what the caller's arguments' exact memory address is.

VAX-11 provides one calling mechanism supported by two instructions. The choice of the instruction is strictly up to the caller. The callee always uses AP to reference arguments:

- The CALLG instruction where the argument list is stored in a caller supplied area, and
- The CALLS instruction where the argument list has been pushed onto the stack by the caller, immediately prior to the call.

In either case, the argument list itself is read only. By convention, it normally consists of an array of pointers to the actual argument variables. This is NOT mandated by the machine! The argument list may well contain the values of the arguments.

- According to these conventions, all argument lists by default contain pointers to the argument variables (known as "call by reference").
- If a procedure is called with argument values ("call by value"), then this fact must be prominently displayed in the procedure preface, in form of a specific notation (see Parameters: Formal).

The procedure may have local variables. Such variables may be either permanently allocated in memory (as a .BLKB, .BLKW or .BLKL allocation) or they may be allocated on the stack (see stack local variables). Stack local variables are allocated upon entry into the procedure, and de-allocated automatically upon return from the procedure. The use of stack locals results in more efficient memory utilization, better working set behavior in the paging environment, and allows the procedure to be called recursively. Even more importantly, stack locals are truly local to the procedure activation and the chance of their values getting clobbered, by some other code that is external to the procedure, is extremely low.

The use of stack locals is recommended. Note that registers are also in the category of stack local variables, assuming that they were specified to be saved in the procedure's entry mask. In general, the

only non-stack variables to be used by a procedure are the variables corresponding to some permanent database that the procedure is responsible for maintaining. As a rule, any variable whose value MUST be remembered across procedure call/returns is permanently allocated; all other variables are temporaries and should be stack resident.

7.24 PROCEDURE: ENTRY

SEE ALSO:
Routine: Entry: Multiple

The procedure entry consists of the procedure name label, and of the procedure entry mask. The first word of a procedure that is called by either CALLG or CALLS is interpreted by the hardware to be a register-save mask. The mask, which is a word (=2 bytes), specifies those registers that are to be saved by the calling mechanism. It also specifies the integer and decimal overflow enables.

You have to specify those registers explicitly. You specify the registers used by your procedure, so that their values will be preserved and restored upon return.

Use the ^M operator to specify the list of registers to be saved:

```
ROUTNAME:                                   ;Name of the procedure
     .WORD   ^M<R2,R3,R4,R10>          ;Save four registers

or

     .ENTRY  GLOBAL_ROUTNAME,^M<R2,R3,R4,R10> ;Save four registers
```

NOTE: Whenever you modify an existing program, and decide to use a register, carefully verify the fact that the register is specified in the procedure entry's save-mask.

REMEMBER: Being overzealous in specifying "efficient" register save masks may cause bugs which are extremely difficult to find; not necessarily in YOUR procedure, but rather in the procedure that CALLed you. That calling procedure may be from the library, and the bug symptom may be extremely horrible and impossible to trace to YOUR procedure which caused the bug by clobbering the caller's register(s).

If your procedure invokes a non-standard routine your entry mask must specify all registers used by that routine (even if that routine does a PUSHR). This is necessary to allow for the case of a signal or exception being generated and a condition handler UNWINDing the stack. (See Condtion Handler, Signal, and UNWIND.)

7.25 .PSECT STATEMENT

Typically, PSECTs have the following attributes.

```
Code       PIC USR CON REL LCL   SHR   EXE RD NOWRT Align(2)
Literals NOPIC USR CON REL LCL   SHR NOEXE RD NOWRT Align(2)
Own      NOPIC USR CON REL LCL NOSHR NOEXE RD   WRT Align(2)
Global   NOPIC USR CON REL LCL NOSHR NOEXE RD   WRT Align(2)
Common   NOPIC USR OVR REL GBL NOSHR NOEXE RD   WRT Align(2)
```

Since the assembler defaults attributes, the following declarations are sufficient and hence preferred:

```
Code            .PSECT  name,PIC,SHR,NOWRT,LONG
Literals        .PSECT  name,SHR,NOEXE,NOWRT,LONG
Own/Global      .PSECT  name,NOEXE,LONG
Common          .PSECT  name,OVR,GBL,NOEXE,LONG
```

Subsequent references to the PSECT should give just the name with no attributes.

7.26 QUEUE INSTRUCTIONS

SEE ALSO:
  Synchronization: Process

SPECIFICS TO BE SUPPLIED

7.27 RELATIVE ADDRESSING

SEE ALSO:
Expressions

The assembler allows the formulation of relative addresses of the form "SYMB+OFFSET". The assembler also allows reference to be made to its current location counter value dot (".").

- Under NO CIRCUMSTANCES is it allowed to make relative address references within the executable code. Code of the form:

```
        BR      .+4                     ;This is a NO-NO
or,
        JMP     LABEL-23                ;This is a NO-NO
```

  is ABSOLUTELY NOT TOLERATED!

- Relative addressing, including dot-relative addressing, is useful --and sometimes necessary-- in the definition of data structures or in the declaration of tables. See expressions, formal parameters and stack local variables for examples.

7.28 ROUTINE: BODY

SEE ALSO:
Comment: Block
Procedure
Statement: Block

The routine's body consists of the sequence of instructions representing the function performed by that routine. The sequence should be decomposed into major groups of instructions, where each group performs a well defined logical operation. Each such group is known as a block statement, and is preceded by its block comment. It should be possible to get a fairly complete knowledge of the routine's logic from simply reading the block comments.

Block statements appear in a logical sequence. The routine's logic must naturally flow in a top-down sequence. All jumps (or branches) must go down the page! The only exception is in the case of loops, where an upwards jump is necessary.

NO SPAGHETTI-BALL CODE IS TO BE TOLERATED!

Note that most loops have their "end" test at the beginning. This is no exception to the above rule in that the loop label is at the top, then the end test including the branch to the exit, then the body followed by the branch back around the loop.

In general, a routine will not have a common exit point because a single RSB or RET instruction performs the return. However, if there is common code in several paths just before return, this should be combined as one exit sequence located at the end of the routine.

7.29 ROUTINE: ENTRY: MULTIPLE

A routine may have several entry points, for either of the following reasons:

- Two or more outwardly different routines effectively use the same algorithm and have an otherwise identical interface. For example, the routines to convert a binary value into OCTAL, DECIMAL and HEXADECIMAL character representations have a common interface and differ only by the conversion radix.

- A single function may have two or more variants necessitating different interfaces. For example, both PRINT and PRINT_NL are entries to the routine that prints a line. The first prints the line without a terminating <newline>, the second prints the line and issues a <newline>.

In either case, each entry point is to be documented with a full routine header. Define the entry point, do some setup computation (setting a flag and/or copying the arguments in the case of non-uniform parameters), then transfer to a common label. In the following example, the mandatory routine headers were ommitted for clarity's sake.

Example:

```
;
;       The binary to octal conversion entry
;

        .ENTRY  BIN_TO_OCT,^M<register list> ;Binary to octal
        MOVL    #8,RADIX                    ;Set radix = 8
        BR      common                      ;
<separator>
;
;       The binary to decimal conversion entry
;

        .ENTRY  BIN_TO_DEC,^M<register list> ;Binary to decimal
        MOVL    #10,RADIX                   ;Set radix = 10
        BR      common                      ;
<separator>
;
;       The Binary to hexadecimal conversion entry
;

        .ENTRY  BIN_TO_HEX,^M<register list> ;Binary to hex
        MOVL    #16,RADIX                   ;Set radix = 16
<separator>
COMMON:                                     ;Common conversion code
```

7.30 ROUTINE: NON-STANDARD

SEE ALSO:
Procedure
Routine: Preface

The non-standard routine differs from the procedure in the fact that it is invoked with the JSB, BSBB, or BSBW instruction and returns by means of the RSB instruction, whereas the procedure is invoked with either the CALLG or the CALLS instructions and returns by means of the RET instruction.

The non-standard routine has no formal stack frame allocation, nor any hardware supported argument passing mechanism. Arguments are passed in predesignated global localities, most typically in registers or pushed onto the stack.

Code and comment the non-standard routine according to the very same rules laid down for the procedure, as exemplified in the Program Structure Overview chapter. However:

- The non-standard routine's entry point MUST NOT consist of a register save mask. If you have to save registers, use an explicit PUSHR instruction.

- Unlike the RET instruction, the stack does not get cleaned automatically, nor do saved registers get restored automatically. Before performing the RSB instruction, adjust the top-of-stack and perform a POPR instruction (if necessary) to restore the explicitly saved registers (if any).

- In the routine preface, clearly indicate that this is a non-standard routine and not a procedure. Clearly specify where the call arguments are to be found, and in what order (especially important if they are pushed onto the stack). These are documented in the INPUT PARAMETERS section. Similarly document the output registers and stack in the OUTPUT PARAMETERS section.

7.31 ROUTINE: ORDER

The following rules apply to the ordering of routine declarations:

- All routines appear together as a group and come after all the declarations in a module.
- Routines are ordered by their use. That is, if routine "A" calls routine "B" then routine "B" appears after "A".
- Mutually recursive routines are ordered by principal entry first.

7.32 .SBTTL STATEMENT

Whenever you switch from one major logical text element to another, you would normally insert a formfeed to force the new element onto a page of its own (e.g., the module's history, declarative part, and the routine(s)). Begin each such logical element with a .SBTTL statement that will cause that subtitle text then to be reprinted on each successive page of the module element.

If two consecutive logical elements will fit entirely on one page with ample excess space, then the form feed can be replaced by four blank lines. The .SBTTL and comments are always included.

7.33 STATEMENT

SEE ALSO:
Comment
Statement: Block

The statement is a single functional step specification of the algorithm. This definition includes functional specifications made to the "assembler machine" as distinct from VAX-11 proper (i.e., assembler directives as distinct from VAX-11 instructions). It also includes higher-level instructions that were defined by means of the MACRO facility.

The statement is of the general form:

```
[LABEL]:                                ;Optional label
        OPCODE  [OPERAND LIST]          ;Opcode and operands
```

Where:

- o [LABEL] is an optional statement label.
- o OPCODE is a VAX-11 Op-Code, or an assembler directive, or a MACRO. It is placed at character position 9 (one tab stop from the left margin).
- o [OPERAND LIST] is an optional list of one or more operands, separated by commas (","). The operand list begins on character position 17 (two tab stops from left margin).

Typically, the statement requires a single line of source text, for example:

```
        MOVL    #10,R5                  ;Initialize loop counter
```

The assembler listing format allows 80 column input lines. VAX-11 instructions, however, may be very lengthy, because:

- o The instruction has a large number of operands, or because
- o The operands themselves are "voluminous".

In addition, because of the object code display constraints, a significant portion of the object listing is dedicated to other than the source text, whose display space is therefore limited. It is therefore very possible that a single statement may not gracefully fit on a single line of text (or even not fit at all).

The statement may be broken into two or more lines of text by means of a statement continuation mark, which is a hyphen ("-"). The mark must be the last non-blank character preceding the comment delimiter. For example:

```
EDIV    BIRTHDAY_CAKE,THREE, -  ;Divide THREE by CAKE
     QUOTIENT,REMAINDER         ;Compute CAKE'th of THREE
```

In general:

- The multiple line statement IS NOT a block statement.
- Use your judgement in best applying the statement continuation feature. It may be put to good use by providing more extensive commenting space on an operand by operand basis, if necessary. Alternatively, there may be good reason to write the statement on a single line (assuming that it fits) and putting the comment on the following line.
- Take pride in producing the most aesthetic looking and consistent source code possible. Having "Raggedy Anne" text and undulating comments is not very pretty. Use the multiple line statement feature to achieve the nicest looking code possible.
- Remember to comment each and every statement. In case that the statement is self evident and needs no comment, remember that a semicolon (";") comment delimiter is still mandatory.

7.34 STATEMENT: BLOCK

A number of statements forming a larger logical unit within the program is known as a block statement. A block statement must not be labelled with a local label (it may include local labels in addition to its own). The block statement need not have a label; however, if it does have local labels then it must be tagged with a label identifying the block.

- The block statement is separated from its predecessor and successor statements (and/or comments) by a blank line. Its label(s), if it has any, is an integral part of the block statement.

- The block statement is to be preceded by a block comment.

Example:

```
   <skip>
;+
;       This is the statement's block comment
;-
<skip>
OPTIONAL_LABEL:                         ;Label's comment
        STATEMENT                       ;
10$:    STATEMENT                       ;Optional local labels
        STATEMENT                       ;
<skip>
```

7.35 STRING INSTRUCTIONS

SPECIFICS TO BE SUPPLIED

7.36 STRUCTURES

SEE ALSO:
  $OWN Macro
  Parameters: Formal
  Variables: Stack Local

Structures are allocated under program control. They may appear in the stack, as formal parameters, or at arbitrary places in memory. They are given symbolic offsets from their base and are referenced relative to some base register.

To declare structures, you have to

(1) Define their symbolic offset names, and to

(2) Explicitly allocate space for them.

Example:

```
;
;       Definition of a 3-item based structure
;

ITEM1=0                                 ;ITEM1's offset
ITEM2=4                                 ;ITEM2's offset
ITEM3=8                                 ;ITEM3's offset
ST_LNG=12                               ;Length of this structure
```

- Assuming memory area VAR to be structured, you will now compute the address of ITEMn by using the expression <VAR+ITEMn>.

- Assuming the address of the structure to be in base register R1, you will access the first byte of ITEMn by specifying the operand ITEMn(R1).

ADDITIONAL SPECIFICS TO BE SUPPLIED about MDL, SDL, and SYSDEF macros.

7.37 SYMBOL

A symbol is an alphanumeric string of up to 15 characters in length. It consists of letters "a" through "z" and "A" through "Z", digits 0 through 9, and special characters underline ("_"), dot (".") and currency sign ("$").

- The assembler does not distinguish between upper- and lower-case alphabetic characters constituting a symbol. Thus "symbol", "SYMBOL", "SyMbOl", "sYmBoL" etc. are all interpreted as equivalent. To minimize reader confusion, never use lower case in symbols. Lower case should be used only in comments and in text strings.

- The underline character "_" is used to separate the parts of a compound (or qualified) name. Freely use the underline when constructing names to improve readability and comprehension.

- The ability of a programmer to infer various attributes of a symbol simply by virtue of its name is a very desireable characteristic.

- The currency sign "$" has been given a special significance within the global VAX-11 software architecture.

Refer to the Naming Conventions chapter for the exact symbol construction rules.

7.38 SYMBOL: EXTERNAL

External symbols will be declared automatically by the assembler. A declaration is needed only if the reference is to be weak (see .WEAK Declaration).

7.39 SYMBOL: GLOBAL

SEE ALSO:
.VALIDATE Declaration
.WEAK Declaration

A global symbol is defined by means of the double colon "::" for label symbols, and by means of the double equate "==" for equated symbols.

Example:

```
   SWITCH::
        .BLKW   1                       ;Global variable SWITCH
TRUE==1                                 ;Global value TRUE
```

7.40 SYNCHRONIZATION: PROCESS

SEE ALSO:
QUEUE Instructions

SPECIFICS TO BE SUPPLIED

7.41 .TITLE STATEMENT

SEE ALSO:
Module: Preface

The .TITLE statement is the very first statement of the module. Its operand is the module name. Any text following the module name is used in the header of the object code listing. The text following the module name should be a terse functional description of the module.

Example:

```
            .TITLE  FILE_MGR - The STARLET file manager subsystem
```

7.42 UNWIND

SEE ALSO:
  Condition Handler
  Signal

If a condition handler gets control, it has several options over the flow of control. It can resignal the condition for another handler to take control, or it can signal a distinct condition for the same purpose. Alternatively, it can continue from the signal. The final option is to terminate the procedures in progress, unwind the stack, and branch to a specific recovery address. This would be done when the current operation is to be aborted, but the program is not to be terminated.

When an unwind is requested, each stack frame is examined in order to restore all the saved registers and Program Status Word (PSW). Before each stack frame is removed, it is examined to see if a condition handler has been established. If so, the handler is called first. This allows a procedure to gain control if it is aborted or if any routine below it aborts. This might be used, for example, to release any resources such as dynamic storage which the routine might have acquired.

7.43 .VALIDATE DECLARATION

SEE ALSO:
  Symbol: External
  Symbol: Global
  .WEAK Declaration

This is used in addition to a global declaration for any symbol which is made global only to validate consistency across several modules. For example, if two modules assume that the length of a particular structure is 47, then both might declare

```
        .VALIDATE STR_LEN
STR_LEN==47
```

This would cause the LINKER to validate that both declarations are the same. The .VALIDATE declaration should not be made if any routine references STR_LEN as an external. It is used only to mark global definitions whose purpose is totally redundant.

7.44 VARIABLES: STACK LOCAL

SEE ALSO:
  Expressions
  Parameters: Formal
  Structures

Stack local variables are allocated at the base of the procedure's stack frame, and given symbolic names that are offsets relative to the procedure's stack frame pointer FP.

Variables may be allocated starting with the longword following FP (the word that would be used by a PUSHL instruction).

To declare stack local variables, you have to:

(1) Define their symbolic offset names, and

(2) explicitly allocate space for them on the stack.

Symbolic definition is performed using the $LOCAL macro, as in:

```
;
;       Definition of stack local variables
;

        $LOCAL  <-
<I,8>,-                                 ;Quad variable I
J,-                                     ;Long variable J
<K,2>,-                                 ;Word variable K
<B,1>>                                  ;Byte variable B
```

The actual allocation is performed using a SUBL2 instruction, as in:

```
;
;       The routine entry point
;

ROUTNAME:                               ;The routine's name
        .WORD   ^M<register list>       ;Save mask
        SUBL2   #$$LOCAL_SIZE,SP        ;Advance SP past allocation
```

Whenever you want to reference one of the stack local variables, do so by using its symbolic name VAR based on the contents of FP (e.g., "VAR(FP)"). Such as:

```
        MOVB    R7,B(FP)                ;Store byte in local B
        ADDL3   I(FP),4+I(FP),J(FP)     ;Add both halves of I into J
```

Compare the allocation of these local variables to the structure definition shown in the structures section. Notice the difference that is due to the stack's backwards growth.

7.45 .WEAK DECLARATION

SEE ALSO:
.VALIDATE Declaration

The .WEAK declaration can be made on either external or global definitions. In both cases its meaning is that the symbol should be matched by the LINKER if defined, but that this reference or definition should not force the loading of a library module.

When used on a global declaration, then the definition of the symbol in this module is not sufficient to cause this module to be loaded from a library. Thus, it should be used for any subordinate symbols defined in a library module.

When used on an external, then the reference to this symbol will not cause it to be defined by loading a library module. If some module which is loaded defines the symbol, then it will be defined for this reference. If nothing defines the symbol, it is automatically satisfied as defined as 0 without any error messages. Thus, it can be used to establish a pointer to an optional module or data base. If the module is loaded, the pointer is defined. Otherwise the pointer has value 0.

[End of Chapter 7]

Title: VAX-11 Software Eng. BASIC Formatting -- Rev 3

Specification Status: draft

Architectural Status: under ECO control

File: SE8R3.RNO

PDM #: not used

Date: 23-Feb-77

Superseded Specs: none

Author:

Typist: P. Conklin

Reviewer(s):

Abstract: Chapter 8 gives each piece of the BASIC formatting and usage conventions in detail. The items are in alphabetical order. Each item includes references to related topics, gives the background and the rules, and then gives templates and examples.

Revision History:

Rev #	Description	Author	Revised Date

CHAPTER 8

BASIC FORMATTING AND USAGE

23-Feb-77 -- Rev 3

This chapter contains detailed information on formatting standards, and instruction usage. For ease of reference, it is organized alphabetically by topic. Each topic includes references to related topics. Most entries also include examples or sample templates illustrating the specific topic.

THE CONTENTS ARE TBS

[End of Chapter 8]

Rev 2 to Rev 3:

1. Create null chapter.

[End of SE8R3.RNO]

Title: VAX-11 Software Engineering BLISS Formating and usage
Specification Status: draft

Archetectural Status: under eco control

File: SE9R3.RNO

PDM: not used

Date: 21-Feb-77

Superceded specs: none

Author: P. Marks, M. Spier

Typist: G. Hesley, R. Murray

Reviewer(s): D. Cutler P. Conklin R. Gourd I. Nassi S. Poulsen

Abstract: This chapter is a collection of procedures and examples of specific BLISS related formats and language usages. It is organized by keywords, in alphabetical order.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Original	P.Marks, M.Spier	2-Aug-77
Rev 2	Review	P.Marks,I.Nassi	1-Jan-77
Rev 3	SEM integration	R.Murray	31-Feb-77

Rev 2 to Rev 3:

1. split from chapter 6 to exclude those features common to both Bliss and Assembler.

[end of se9r3.rno]

CHAPTER 9

BLISS FORMATING AND USAGE

21-Feb-77 -- Rev 3

The following is an explanation of some of the terms used throughout this section.

Logical tab	equivalent to four (physical) spaces. Used for indenting BLISS source text. Two successive logical tabs should be typed as one physical tab.
Physical tab	the ASCII TAB character (octal 11). All standard DEC software interprets the tab as equivalent to moving the carriage or cursor to the next column number which is one more than a multiple of eight.
Tab	used throughout this manual to mean logical tab.
Indentation level	the number of logical tabs a line of text is offset to the right of the left page margin.
Indented	offset one logical tab to the right of the text on the preceding line.
Line	The contents of one record.

9.1 DECLARATION

See:
Declaration: Format
Declaration: FORWARD ROUTINE
Declaration: MACRO
Declaration: Order

9.2 DECLARATION: FORMAT

Declarations are written according to the following format:

```
declaration-keyword(s)
        declaration-item,               ! Comment
        declaration-item,               ! Comment
            ...
            ...
        declaration-item;               ! Comment
```

The following rules apply to declaration formatting:

- Each declaration-keyword appear(s) alone on a line and starts at the left margin of the block in which the declaration is being made.
- The declaration-item(s) being declared appear indented one logical tab with respect to the declaration-keyword and on a separate line(s).
- Declaration-items are in an order meaningful to the program organization, or in alphabetical order.
- Each declaration-item has a line comment on the same line describing, in most cases, the meaning and/or usage of the declaration-item being declared.

9.3 DECLARATION: FORWARD ROUTINE

SEE ALSO:
Declaration: Format

The following rules apply to FORWARD ROUTINE declarations:

- o forward ROUTINE declarations for a module are grouped together and appear at the beginning of the module.
- o The FORWARD ROUTINE declaration names all the routines to be declared in the module in order of occurrence.
- o Each routine name is on a separate line with a line comment briefly explaining its function.
- o The FORWARD ROUTINE declaration serves as a table of contents for the module.

9.4 DECLARATION: MACRO

SEE ALSO:
Declaration: Format
Expression

The following rules apply to MACRO declarations:

- o MACRO declarations follow the general formatting rules outlined under DECLARATION: format.
- o If the body of the MACRO is composed of declarations and/or expressions, then the body conforms to all the formatting rules for declarations and/or expressions.
- o If the macro has a formal-list, then the commenting rules for ROUTINES should be applied, in so far as describing each of the formal parameters and commenting on the function of this macro.

9.5 DECLARATION: ORDER

SEE ALSO:
Declaration: FORMAT
Declaration: FORWARD
Declaration: MACRO
Routine

We group the BLISS declarations as follows:

1. FORWARD declarations
2. REQUIRE declarations
3. All other declarations
4. ROUTINE declarations

The first, second and fourth groups are discussed in their own sections. The third group lumps all other declarations (e.g., STRUCTURES, LITERALS, MACROS, etc.), which have module-wide or routine-wide scope, into one major group.

The ordering of the different declarations within this third group is important and is based on the following rules:

- Group logically related declarations together. For example, a specific structure may be used in conjunction with certain macros. These declarations would then appear together as a group.
- As much as possible, these logical groups will appear in the order of their use within the module or routine.
- Separate the logical groups from each other by the use of appropriate separators.
- Within a logical group of declarations group specific declarations together by type. For example, all MACROS will be defined via one or more MACRO declarations.

A word of caution: Owing to the nature of the BLISS language, it is necessary to declare all variables, structures, routines, etc. before they are used. Care should be taken so as not to use something before it is declared. In any event, the compiler will complain.

9.6 EXPRESSION

SEE:
Expression: Assignment
Expression: CASE
Expression: Block
Expression: Format
Expression: IF/THEN/ELSE
Expression: INCR/DECR
Expression: SELECT
Expression: WHILE/UNTIL/DO

9.7 EXPRESSION: ASSIGNMENT

Assignment expressions are usually of the form:

```
name = expression
```

The following rules apply to assignment expressions:

- If the entire assignment expression will not fit on one line because of its length then place the variable and the equal sign on one line and continue the expression indented one logical tab on the next line.

Examples (correct):

```
name = a-short-expression;                ! comment

        (Note the space before and after the
         = sign.)

name = a-short-expression;                ! a long long
                                          ! comment

name =                                    ! a comment for
    a-long-long-long-long-expression;     ! this expression
```

9.8 EXPRESSION: CASE

CASE expressions are set up according to the following skeletal example:

```
CASE  index
    FROM low-case TO high-case OF
    SET

    case-label-action:

    case-label-action;

    ...
    ...

    case-label-action;

    TES
```

where case-label-action is:

```
    [case-label]:
        ! Explanatory comments
        ! for this case.

        case-action;
```

or

```
    [case-label]: case-action;          ! comment
```

The following rules apply to CASE expressions:

- The body of the CASE expression is indented one logical tab with respect to the keyword CASE.
- Each case-label-action is separated from another case-label-action by at least one blank line.
- The choice of format for the case-label-action is dependent on the size (number of expressions) of the case-action. A large case-action will use the first format; a small case-action the second format.
- Each of the case-actions follows the rules for expression formatting.
- It is desirable that the case-label be a descriptive and meaningful name that has been bound to its value. A case-label then becomes a label or signal to the reader indicating what value caused this case-action to be used.

9.9 EXPRESSION: BLOCK

A block expression provides a means of grouping declarations and/or expressions into a single structural entity.

The following rules apply for BLOCK expressions:

- The block expression is separated from its predecessor and successor expressions (and/or comments) by a blank line.
- The block expression is to be preceded by a block comment.
- Constituent declarations and expressions of a block are indented to the same level as the BEGIN-END delimiters.
- In a block expression, the last expression in the block is followed by a ";" unless the value of the block expression is actually used in an enclosing expression.

9.10 EXPRESSION: FORMAT

Specific formatting rules apply for each kind of executable expression. In general, the following rules apply:

- Expressions generally appear on separate lines.
- Expressions are left justified to the current indentation level.
- Expressions which fit on one line may appear on one line.
- Expression subparts, when indented, are indented one logical tab to the right of the start of the expression. Specific indentation rules are given in the appropriate sections.
- Compound-expressions consisting of more than one line are bounded by BEGIN-END delimiters rather than by parentheses.
- In general, for arithmetic expressions:
  - Place one space around the binary "+" and "-".
  - Place one space before the unary "+" and "-".
  - Place no spaces around the "*" and "/" operators.
  - In lists, place one space before the "(" and one space after each "," and the ")".

9.11 EXPRESSION: IF/THEN/ELSE

IF expressions are written in either of two formats:

```
IF test THEN consequence ELSE alternative
```

or

```
IF test
THEN
    consequence
ELSE
    alternative;
```

- In the first case, the entire IF expression may be placed on one line only if the IF expression fits on one line.
- Otherwise, the second format is used. The consequence and alternative expressions are indented one logical tab with respect to the keyword IF.

If the test is a compound test then the IF expression is written in one of the following manners:

```
IF test AND test AND test
THEN
    consequence
ELSE
    alternative
```

or

```
IF test AND
   test AND
   test
THEN
    consequence
ELSE
    alternative
```

- The first format is used when the compound test can fit on one line. Otherwise, the second format is used.

9.12 EXPRESSION: INCR/DECR

INCR/DECR expressions are written according to one of the following formats:

```
INCR loop-index FROM first TO last BY step DO
    loop-body;
```

or

```
INCR loop-index
    FROM first TO last BY step DO
    loop-body;
```

The following rules apply to INCR/DECR expressions:

- Use the first format when the FROM-TO-BY expression will fit on one line. Otherwise, use the second format.
- The loop-body is indented one logical tab with respect to the keyword INCR/DECR.

9.13 EXPRESSION: SELECT

SELECT expressions are set up according to the following skeletal example:

```
SELECT select-index OF
    SET

    select-label-action;

    select-label-action;

            ...
            ...

    select-label-action;

    TES
```

where select-label-action is:

```
    [select-label]:
        ! Explanatory  comments
        ! for this select-label.

        select-action
```

or

```
    [select-label]: select-action;      ! comment
```

The following rules apply to SELECT expressions:

- The body of the SELECT expression is indented one logical tab with respect to the keyword SELECT.
- Each of the select-label-action expressions is separated by at least one blank line.
- The choice of format for the select-label-actions is dependent on the size (number of expressions) of the select-action. A large select-action will use the first format; a small select-action the second format.
- It is desirable that the select-label be a descriptive and meaningful name that has been bound to its value. A select-label then becomes a label or signal to the reader indicating what condition or value caused this select-action to be SELECTed.

9.14 EXPRESSION: WHILE/UNTIL/DO

WHILE/UNTIL/DO expressions are written in the following manner:

```
WHILE test DO
    loop-body;
```

or

```
DO
    loop-body
WHILE test;
```

The following rules apply to WHILE/UNTIL/DO expressions:

- The keyword WHILE or UNTIL is aligned with the current indentation level.
- The loop-body is indented one logical tab with respect to the keyword WHILE or UNTIL and follows the rules for expression formatting.

9.15 IDENT MODULE SWITCH

The IDENT switch has, as its parameter, the current version number of the module. This version number corresponds to the last entry in the module's ABBREVIATED HISTORY.

9.16 LABELS

A label is a name, hence it must conform to the rules for constructing names. It is delimited by a colon ":".

The following rules apply to labels:

- Labels, when used, appear alone on a line. The block to which they refer follows on the next line indented one logical tab with respect to the label.
- A label is meaningful in the sense that it conveys some information about the block it is labelling.

9.17 MODULE: SWITCHES

Module switches appear in the module declaration and allow the programmer to provide information about the module and to control some aspects of the compiler's treatment of the module. Of special importance is the IDENT switch (see IDENT Module Switch) and the MAIN switch which specifies which routine is to be used to begin program execution.

- Each module switch will appear on a line by itself. The IDENT switch is first, the MAIN switch is second; any other switches follow.

Example (correct):

```
        MODULE  EXAMPLE (
                              IDENT = '03',
                              MAIN  = BEGINHERE,
                              RESERVE = (R0, R1)
                             ) =
```

9.18 NAME

A name consists of one to fifteen characters from the sets:

1. A B C D E ... X Y Z
2. a b c d e ... x y z
3. 0 1 2 3 4 5 6 7 8 9
4. underline "_"
5. dollar "$"

No distinction is made between upper and lowercase letters except in string literals. Thus, Date_Of_Birth is equivalent to date_of_birth.

The following rules apply to names:

- Freely use the underline "_" when constructing names to improve readability and comprehension. For example: WRITEARECORD becomes WRITE_A_RECORD.
- The ability of a programmer to infer various attributes of a symbol simply by virtue of its name is a very desirable characteristic.
- Predefined and syntactically meaningful names are to be used only for their intended purpose.

9.19 REQUIRE FILES

The purpose of REQUIRE files is to centralize in one place declarations and definitions that are common to multiple modules. Data STRUCTURE declarations, MACRO declarations, and LITERAL declarations are the principal contents of REQUIRE files.

REQUIRE files consist of the following:

1. ! file-name - description
2. A copyright statement and disclaimer.
3. A MODULE PREFACE.
4. The text of the REQUIRE file. The text conforms to the formatting rules for declarations.
5. ! file-name - LAST LINE

9.20 ROUTINE

SEE:

Declaration: Order
Routine: Format
Routine: Name
Routine: Order
Routine: Preface

9.21 ROUTINE: FORMAT

The following rules apply for ROUTINE formatting:

- The routine declaration is to start at the left margin.
- The routine body is to be indented one logical tab to the right of the routine declaration.
- All other indentation follows the rules for declaration and/or expression formats.

9.22 ROUTINE: NAME

Global routine names should follow the naming conventions stated earlier. Local routine names may be chosen at as desired.

9.23 ROUTINE: ORDER

The following rules apply to the ordering of routine declarations:

- All routine declarations appear together as a group and constitute the last set of declarations in a module.
- Routines are ordered by their use. That is, if routine "A" calls routine "B" then routine "B" is declared after "A".

- The ordering of routines is reflected in the FORWARD declaration group appearing at the beginning of the module.
- Mutually recursive routines are ordered by principle entry first.

9.24 STRUCTURE: DECLARATION

SEE:
STRUCTURE: Block

The format for the structure declaration is as follows:

```
STRUCTURE
    structure-name [access formal list;allocation formal list]=
        [structure size]
        structure body;
```

The following rules apply to the structure declaration:

- The structure declaration format generally conforms to that of macros
- The structure-name is indented one logical tab.
- The structure size and structure body are indented another logical tab.
- The structure body contains one expression. The format rules regarding expressions are in force starting with the indicated indentation level.

In the instance where the expression part of the structure body is simple, it may be contained on one line as seen below:

```
STRUCTURE
    BLOCK[O,P,S,E;N,UNIT = %UPVAL] =
        [N*UNIT]
        (BLOCK+O*UNIT)<P,S,E>;
```

or, may be of such complexity as to require the use of most rules for formatting expressions.

```
STRUCTURE
    VECTOR1CH[I;N,UNIT = %UPVAL] =
        [N*UNIT]
        BEGIN
            LOCAL T;
            T=.I;
            IF .T LSS 1 OR .T GTR N
            THEN
                BEGIN
                ERROR(.T);
                T=1;
                END;
            VECTOR1CH + (.T - 1) * UNIT
        END;
```

9.25 STRUCTURE: BLOCK

SEE:
STRUCTURE: Declaration

The structure called BLOCK is a predeclared structure which may be used without an explicit declaration. If declared it would look as follows:

```
STRUCTURE
    BLOCK[O,P,S,E;N,UNIT = %UPVAL] =
        [N*UNIT]
        (BLOCK + O * UNIT)<P,S,E>;
```

Consider the following example.

```
OWN
        X:BLOCK[2];
        .
        .
        .
A = .X[0,0,16,0];
B = .X[0,16,16,0]
C = .X[1,0,32,0]
```

X is defined as a two word BLOCK whose first word has two fields, each 16 bits long and whose second word is a field 32 bits long. The above assignment statements use the BLOCK definition to access each field.

NOTE

For a further explination of the structure declaration and built-in structures, see the chapter on Data Structures in the BLISS Language Guide.

The BLISS programmer is strongly urged to hide the 4-tuple used to access a BLOCK by using a "field macro" as follows:

```
MACRO
    FIELD_ONE = 0,0,16,0%,
    FIELD_TWO = 0,16,16,0%,
    FIELD_THREE = 1,0,32,0%;
```

Thus the access to the BLOCK X becomes:

```
A = .X[FIELD_ONE];
B = .X[FIELD_TWO];
C = .X[FIELD_THREE];
```

This achieves a greater degree of readibility and facititates future changes to the structure of X.

[end of chapter 9]

Title: VAX-11 Software Eng. COBOL Formatting -- Rev 3

Specification Status: draft

Architectural Status: under ECO control

File: SE10R3.RNO

PDM #: not used

Date: 23-Feb-77

Superseded Specs: none

Author:

Typist: P. Conklin

Reviewer(s):

Abstract: Chapter 10 gives each piece of the COBOL formatting and usage conventions in detail. The items are in alphabetical order. Each item includes references to related topics, gives the background and the rules, and then gives templates and examples.

Revision History:

Rev #	Description	Author	Revised Date

Rev 2 to Rev 3:

1. Create null chapter.

[End of SE10R3.RNO]

CHAPTER 10

COBOL FORMATTING AND USAGE

23-Feb-77 -- Rev 3

This chapter contains detailed information on formatting standards, and instruction usage. For ease of reference, it is organized alphabetically by topic. Each topic includes references to related topics. Most entries also include examples or sample templates illustrating the specific topic.

THE CONTENTS ARE TBS

[End of Chapter 10]

Title: VAX-11 Software Eng. Fortran Formatting -- Rev 3

Specification Status: draft

Architectural Status: under ECO control

File: SE11R3.RNO

PDM #: not used

Date: 23-Feb-77

Superseded Specs: none

Author:

Typist: P. Conklin

Reviewer(s):

Abstract: Chapter 11 gives each piece of the Fortran formatting and usage conventions in detail. The items are in alphabetical order. Each item includes references to related topics, gives the background and the rules, and then gives templates and examples.

Revision History:

Rev #	Description	Author	Revised Date

Rev 2 to Rev 3:

1. Create null chapter.

[End of SE11R3.RNO]

CHAPTER 11

FORTRAN FORMATTING AND USAGE

23-Feb-77 -- Rev 3

This chapter contains detailed information on formatting standards, and instruction usage. For ease of reference, it is organized alphabetically by topic. Each topic includes references to related topics. Most entries also include examples or sample templates illustrating the specific topic.

THE CONTENTS ARE TBS

[End of Chapter 11]

Title: VAX-11 Software Engineering Naming Conventions -- Rev 3

Specification Status: draft

Architectural Status: under ECO control

File: SE12R3.RNO

PDM #: not used

Date: 28-Feb-77

Superseded Specs: VAXS notes; based on STARLET Working Design Document

Author: P. Conklin, S. Gault

Typist: P. Conklin

Reviewer(s): R. Brender, D. Cutler, R. Gourd, T. Hastings, I. Nassi, D. Tolman

Abstract: Chapter 12 gives the system wide naming conventions for all public symbols. These rules are to be followed by all DEC software for all symbols which are global or appear in parameter definition files. This chapter also includes the list of all facility prefixes.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Original	S. Gault	Oct-76
Rev 2	Revised from Review	S. Gault	Jan-77
Rev 3	After 6 months experience	P. Conklin	28-Feb-77

Rev 2 to Rev 3:

1. Add table of prefixes.

2. Add reasons for the rules.

3. Add BLISS field extract macro names. Add .PSECT names. Add non-CALL entry names. Change $C_ to $K_ for constants.

4. Add transportable data types of A, C, G, H, and U. Note reservations of I and R for specific purposes, use of X and Y for context dependent purposes, use of Z for unspecified or nonstandard forms, use of N and P for decimal strings, and O as a general escape valve.

5. Add all known facility prefixes.

6. Reserve data type J to customers.

7. Note reserved status codes<2:0>. Note that <31:16> indicate facility. Add facility codes to section 7.3.

8. Change non-call routine name pattern to agree with OTS.

9. Change BLISS field reference mnemonics. Reserve E to DEC.

10. Clarify that numeric string is all byte forms.

11. Add argument style column to facility table.

12. Clarify that system macro names are general and don't have the facility name.

13. Clarify that BLISS field names have offset, position, size, and sign.

14. Clarify that assembler V symbols are within containing field.

15. Clarify that masks are not right justified.

16. Add facility to structure def macros.

17. Define sizes of transportable codes for reference. Change H to be good for counters (16 to 18 bits).

18. Add B32, FAB, IO, NAM, NET, PLI, RAB, RM, SWP, TST, XAB prefix. Remove CHF prefix.

19. Ban local synonyms for public symbols.

20. Move completion code description to chapter 6.

21. Clarify that H is integer.

22. Clarify that the N and P count is a digit count.

23. Clarify private symbol usage.

24. Add facility codes for all procedure library facilities.

[End of SE12R3.RNO]

CHAPTER 12

NAMING CONVENTIONS

28-Feb-77 -- Rev 3

The conventions described in this chapter were derived to aid implementors in producing meaningful public names. Public names are all names which are global (known to the linker) or which appear in parameter or macro definition files and libraries in more than one facility.

These public names are all constrained to follow these rules for the following reasons:

- By using names reserved to DEC, we ensure that customer written software will not be invalidated by subsequent releases of DEC products which add new symbols.

- By using definite patterns for different uses, we allow the reader to judge the type of object being referenced. For example, the form of macro names is different from offsets, which is different from status codes.

- By using certain codes within a pattern, we associate the size of an object with its name. This increases the likelihood that the reference will use the correct instructions.

- By using a facility code in symbol definitions, we give the reader an indication of where the symbol is defined. We also allow separate groups of implementors to choose names which will not conflict with one another.

Never define local synonyms for public symbols. The full public symbol should be used in every reference to give maximum clarity to the reader.

12.1 PUBLIC SYMBOL PATTERNS

All DEC public symbols contain a currency sign. Thus, customers and applications developers are strongly advised to use symbols without currency signs to avoid future conflicts.

Public symbols should be constructed to convey as much information as possible about the entity they name. Frequently, private names follow a similar convention; the private convention then is the same as the public one with an underline instead of the currency sign. These are used both within a module and globally between modules of a facility which is never in a library. All names which might ever be bound into a user's program must follow the rules for public names; in the case of undocumented names a double currency sign convention can be used such as in 3 below.

Public names are of the following forms:

1. Service macro names are of the form:

   $macroname

   A trailing _S or _A distinguishes the stack and separate arglist forms. These names appear in the system macro library and represet a call to one of many facilities. The facility name usually does not appear in the macro name.

2. Facility specific public macro names are of the form:

   $facility_macroname

3. System macros which use local symbols or macros always use ones of the form:

   $facility$macroname

   This is the form to be used for symbols generated by a macro and used across calls to it and for internal macros which are not documented.

4. Status codes and condition values are of the form:

   facility$_status

   See completion codes in the Commenting Conventions chapter.

5. Global entry point names are of the form:

   facility$entryname

6. Global entry point names which have non-standard calls are of the form:

        facility$entryname_Rn

   where registers R0 to Rn are not preserved. Note that the caller of such an entry point must include at least registers R2 through Rn in its own entry mask.

7. Global variable names are of the form:

        facility$Gt_variablename

   The letter G stands for global variable and the t is a letter representing the type of the variable as defined in the next section.

8. Addressable global arrays use the letter A (instead of the letter G) and are of the form:

        facility$At_arrayname

   The letter A stands for global array and t is one of the letters representing the type of the array element according to the list in the next section.

9. In the assembler, public structure offset names are of the form:

        structure$t fieldname

   The t is a letter representing the data type of the field as defined in the next section. The value of the public symbol is the byte offset to the start of the datum in the structure.

10. In the assembler, public structure bit field offset and single bit names are of the form:

        structure$V_fieldname

    The value of the public symbol is the bit offset from the start of the containing field (not from the start of the control block).

11. In the assembler, public structure bit field size names are of the form:

        structure$S_fieldname

    The value of the public symbol is the number of bits in the field.

12. For BLISS, the functions of the symbols in the previous three items are combined into a single name used to reference an arbitrary datum. Names are of the form:

        structure$x_fieldname

    where x is t for standard sized data and x is V for arbitrary and bit fields. The macro includes the offset, position, size, and sign extension suitable for use in a REF BLOCK structure. Most typically, this name is definable as

```
MACRO
    structure$V_fieldname =
        structure$t_fieldname,
        structure$V_fieldname,  !assembler meaning
        strucutre$S_fieldname,
        <sign extension> %;
```

13. Public structure mask names are of the form:

        structure$M_fieldname

    The value of the public symbol is a mask with bits set for each bit in the field. This mask is not right justified; rather it has structure$V_fieldname zero bits on the right.

14. Public structure constant value names are of the form:

        structure$K_constantname

15. .PSECT names are of the form:

        facility$mnemonic

16. Module names are of the form:

        facility$mnemonic

    The module is stored in a file with filename "mnemonic" in a directory corresponding to the facility.

17. Public structure definition macro names are of the form:

        $facility_structureDEF

    Invoking this macro defines all the structure$xxx symbols.

Example of usage:

IOC$IODONE	Entry point of the routine IODONE in the I/O subsystem.
UCB$B_FORK_PRI	Offset in the UCB structure to a byte datum containing the fork priority.

UCB$L_STATUS	Offset in the UCB structure to a longword datum containing status bits.
CRB$M_BUSY	Mask pattern for the busy bit in the CRB structure.
CRB$V_BUSY	Bit offset in the CRB structure of the busy bit.

12.2 OBJECT DATA TYPES

The following are the letters used for the various data types or are reserved for the following purposes:

letter	data type or usage
A	address (*)
B	byte integer
C	single character (*)
D	double precision floating
E	reserved to DEC
F	single precision floating
G	general value (*)
H	integer value for counters (*)
I	reserved for integer extensions
J	reserved to customers for escape to other codes
K	constant
L	longword integer
M	field mask
N	numeric string (all byte forms)
O	reserved to DEC as an escape to other codes
P	packed string
Q	quadword integer
R	reserved for records (structure)
S	field size
T	text (character) string
U	smallest unit of addressable storage (*)
V	field position (assembler); field reference (BLISS)
W	word integer
X	context dependent (generic)
Y	context dependent (generic)
Z	unspecified or non-standard

N, P, and T strings are typically variable length. Frequently in structures or I/O records they contain a byte-sized digit or character count preceding the string. If so, the location or offset is to the count. Counted strings cannot be passed in CALLs. Instead, a string descriptor is generated.

* - The letters A, C, G, H, and U should be used in preference to L, B, L, W, and B respectively when transportability is involved. The following table defines their sizes:

letter	16	32	36
A	16	32	18
C	8	8	7
G	16	32	36
H	16	16	18
U	8	8	36

12.3 FACILITY PREFIX TABLE

Following is a list of all the facility prefixes. This list will grow over time as new facility prefixes are chosen. No one should use a new code without first "signing out" the prefix with the author of this chapter. Each facility has a typical style of interface, see the Functional and Interface Specifications chapter, and a condition value<31:16> code.

prefix	facility	interface type (see Chap 13)	condition <31:16>
BAS	BASIC support library	V	26
B32	BLISS-32 support library	V	27
BLI	BLISS transportable support library	V	20
CH	Character handling (BLISS)	-	-
CHF	Condition Handling Facility arguments	-	-
CME	Compatibility mode emulator	J	??
COB	COBOL support library	V	25
DEB	Debugger	V	??
FAB	RMS File Access Block	-	-
FOR	Fortran support library	V	24
IO	Input/Output functions	-	-
LIB	Miscellaneous routines	any	21
MTH	Math library	F	22
NAM	RMS Name Block	-	-
NET	Network ACP	J	??
OTS	Common Object Time System	V	23
PLI	PL/1 support library	?	??
PR	Processor Registers	-	-
PRV	Privileges	-	-
PSL	Program Status Longword fields	-	-
RAB	RMS Record Access Block	-	-
RM	RMS internals and status codes	V	1
RMS	Record Management System	V	-
SRM	System Reference Manual Misc. offsets	-	-
SS	System Service Status Codes	-	0
SYS	System Services	V	-
TST	Test packages	any	-
XAB	RMS Extra information Access Block	-	-

Individual products such as compilers also get unique facility codes formed from the product name. They must be signed out in the above list. Facility prefixes should be chosen to avoid conflict with file types.

Structure name prefixes are typically local to a facility. Refer to the individual facility documentation for its structure name prefixes. This does not cause problems since these names are not global, so are not known to the linker. They become known at assembly or compile time only by invoking the structure's definition macro explicitly.

[End of Chapter 12]

Title: VAX-11 Software Eng. Interface Specifications -- Rev 3

Specification Status: draft

Architectural Status: under ECO control

File: SE13R3.RNO

PDM #: not used

Date: 28-Feb-77

Superseded Specs: Part of OTS design chapter 2

Author: P. Conklin, T. Hastings

Typist: P. Conklin

Reviewer(s): R. Brender, D. Cutler, R. Gourd, I. Nassi, M. Spier, D. Tolman

Abstract: Chapter 13 describes the standards and conventions used by all modules in the VAX-11 Procedure Library, including the Object Time System. The necessary standards are specified to permit many different individuals to contribute modules independently to the VAX-11 library with a consistent interface documentation. To achieve these modularity objectives, this chapter also standardizes the way arguments are passed, and in particular, the way in which strings are returned. It describes a language independent notation for procedure parameters, including the type of access, the data type, the argument passing mechanism, and the form of the argument.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Original	T. Hastings	17-Jan-77
Rev 2	Revised from Review	T. Hastings	21-Jan-77
Rev 3	Integrated with Soft Eng Manual	P. Conklin	28-Feb-77

Rev 2 to Rev 3:

1. Extracted procedure specification notation from OTS chapter 2 (PL2R1).
2. Added routine interface types including applications languages.
3. Remove descriptor code numbers and redundant alphabectical table.
4. Quad by-value is ok only on function values.
5. Note op sys option on s descriptor.
6. Change <data type> U to Z for compatibility.
7. Note length and string descriptor pair is length first.
8. Clarify that data type is always the ultimate use.
9. Allow references only to data size.
10. Add data types c, u, h, g.
11. Drop label arg form.
12. Add summary table.
13. Add data type cp.
14. Allow value args to be less than longword in reference use. (Allocation is still longword.)

Rev 1 to Rev 2:

1. Add <data type> codes la, las, and lc.
2. Add <arg form> code d.
3. Add braces notation for repeated arguments.
4. Add = notation for default value
5. Add <arg form> code p.
6. Clarify use of <data type> with call by value <arg mechanism> for other than 32 bits.
7. Change <data type> C to T for compatibility.

8. Change <data type> la to a for compatibility.

9. Change <data type> las to arl, arw, arb.

[End of SE13R3.RNO]

CHAPTER 13

FUNCTIONAL AND INTERFACE SPECIFICATIONS

28-Feb-77 -- Rev 3

This chapter describes the standards and conventions used by all modules in the VAX-11 Procedure Library, including the Object Time System. The necessary standards are specified to permit many different individuals to contribute modules independently to the VAX-11 library with a consistent interface documentation. To achieve these modularity objectives, this chapter also standardizes the way arguments are passed, and in particular, the way in which strings are returned. It describes a language independent notation for procedure parameters, including the type of access, the data type, the argument passing mechanism, and the form of the argument.

The VAX-11 Procedure Library is a collection of routines that provide various services to the calling program. It is made up of a number of sub-libraries. The Math library contains all those functions that perform the traditional Fortran mathematical functions. The common Object Time System is a collection of resource and environment control routines that are common to all application language environments. Each compiler has a library of routines for which it implicitly generates code. Finally, the general library contains routines that are of general use and typically would be called explicitly by the programmer.

13.1 ROUTINE INTERFACE TYPES

In order to achieve the VAX-11 goal of being able to mix languages within a program, all routines are designed with certain attributes in common. The data types and mechanism passing rules are constrained to maximize the ability to interface to routines. A common notation is used to express the specification of the interface.

The access types, data types, mechanisms, and argument forms are defined in the VAX-11 System Reference Manual. Section 2 of this chapter lists them and gives the procedure interface notation for them. In the design of a procedure interface, in addition to the data types that must be designed, four other choices are important.

1. Whether the routine is CALLed or has a non-CALL interface.

2. Whether its scalar input arguments are by value or by reference.

3. How output strings are returned; this is discussed in the next paragraph.

4. Whether the routine has a function value and whether the value is a status code or a scalar result.

Within any given facility, it is generally preferable to have only one style of these interface choices. The facility table in the Naming Conventions chapter indicates what the conventional interface is for each facility. These are defined below. Other combinations can be chosen but the prospect of user confusion must be traded off against the possible inefficiency of forced consistency.

Output strings can be returned by one of four methods.

- The simplest is for the caller to allocate a fixed length string buffer and pass a descriptor of it. The callee writes the result to this buffer with blank fill.

- The next most general is for the caller to allocate a fixed length string buffer that can hold the maximum length result. The caller passes two arguments, one is the address of where to write the actual length and the other is a descriptor to the buffer. By convention, these two arguments are always adjacent in the argument list with the length first.

- The third mechanism is to pass a varying string descriptor. In this case, the caller allocates a maximum buffer and passes a descriptor that contains fields for both the maximum length and the actual length. The callee updates the actual length field in the descriptor.

- The fourth method is for the caller to pass a dynamic string descriptor. In this case the callee allocates the string buffer and places both the address and the length into the

dynamic descriptor.

The choice between these methods is a function of what environmental assumptions can be made in the design of the procedure. For the fixed length method, no assumptions are made. The others all assume that the calling language can support variable length strings or substrings. The dual argument form can be used without requiring variable length strings, but gives most of the advantages of them to languages that support them. The varying and dynamic schemes both require languages that support varying length strings. Furthermore, the dynamic method requires the support of a dynamic storage management system.

The most common combinations of interface specifications are given in the following table. The column "scalars" shows how scalars are passed. The column "strings" shows how output strings are returned. The column "function" shows what kind of function value is returned.

type of call	instruction	passing scalars	output strings	function value
J (non-CALL)	JSB	parameter	-	-
V (by Value)	CALL	AP by value	length,descr	.lc
F (Function)	CALL	AP by reference	none	scalar
Fortran	CALL	AP by reference	fixed	any
COBOL	CALL	AP by reference	fixed	none
BASIC	CALL	AP by reference	dynamic	any

13.2 NOTATION FOR DESCRIBING PROCEDURE ARGUMENTS

A concise language-independent notation is used to describe each argument to a library procedure. It is suggested that this notation be used for documenting all procedures in the procedure library and in the procedure header itself under CALLING SEQUENCE or FORMAL PARAMETERS. The notation is a compatible extension to the one used in the VAX-11 System Reference Manual. However, the goal of the notation is to describe the formal parameter specified by each list entry in a language independent way. The System Reference Manual only describes the immediate operand specifier, rather than the argument being pointed to. Therefore, additional qualifiers have been added to the System Reference Manual notation. Note that if a parameter is an address which is saved for later access by another procedure, the notation should reflect the ultimate access to be made by the second procedure.

The notation specifies for each argument:

1. A mnemonic name
2. The type of access the procedure will make (read, write,...)
3. The data type of the argument (longword, floating,...)
4. The argument passing mechanism (value, reference, descriptor)
5. The form of the argument (scalar, array,...)

13.2.1 Procedure Parameter Qualifiers

Subroutines are described as:

```
CALL subroutine_name(arg1, arg2, ..., argn)
```

and functions are described as:

```
function_value = function_name(arg1, arg2, ..., argn)
```

where argi and function_value are:

```
<name>.<access type><data type>.<arg mechanism><arg form>
```

where:

1. <name> is a mnemonic for the procedure formal specifier or function value specifier.

2. <access type> is a single letter denoting the type of access that the procedure will (or may) make to the argument:

   r - argument may be read only

   m - argument may be modified, i.e., read and written.

   w - argument may be written only.

   j - argument is an address to be (optionally) JMPed to after stack unwind (return). No <data type> field is given since the argument is a sequence of instructions, e.g., Fortran ERR=.

   c - argument is an address of a procedure to be (optionally) CALLed after stack unwound (return). No <data type> field is given since the argument is a sequence of instructions.

   s - argument is an address of a procedure subroutine to be (optionally) CALLed without unwinding the stack. No <data type> field is given since the argument is a sequence of instructions.

   f - argument is an address of a function to be (optionally) CALLed without unwinding the stack. The <data type> field indicates the data type of the function value.

   a - reserved for use in the System Reference Manual (address). Not used here since the object pointed to is specified.

   b - reserved for use in the System Reference Manual (branch destination). Not used here since a branch destination cannot be a procedure formal.

   v - reserved for use in the System Reference Manual (variable bit field).

3. <data type> is a letter denoting the primary data type with trailing qualifier letters to further identify the data type. Note that the routine must reference only the size specified to avoid improper access violations.

Letters	Use
z	Unspecified
v	Bit (variable bit field)
bu	Byte Logical (unsigned)
c	Single character
u	Smallest unit for addressable storage
wu	Word Logical (unsigned)
lu	Longword Logical (unsigned)
a	Absolute virtual address
cp	Character pointer
lc	Longword containing a completion code
qu	Quadword Logical (unsigned)
b	Byte Integer (signed)
arb	Byte containing a relative virtual address (*)
w	Word Integer (signed)
h	Integer value for counters
arw	Word containing a relative virtual address (*)
l	Longword Integer (signed)
g	General value
arl	Longword containing a relative virtual address (*)
q	Quadword Integer (signed)
f	Single-Precision Floating
d	Double-Precision Floating
fc	Complex (Floating)
dc	Double-Precision Complex
t	text (character) string
nu	Numeric string, unsigned
nl	Numeric string, left separate sign
nlo	Numeric string, left overpunched sign
nr	Numeric string, right separate sign
nro	Numeric string, right overpunched sign
nz	Numeric string, zoned sign
p	Packed decimal string
x	Data type indicated in descriptor

* - arl, arw, and arb is a self-relative address using the same format as the hardware displacements. That is the self-relative address is a signed offset in bytes with respect to the first byte following the argument.

4. <arg mechanism> is a single letter indicating the argument mechanism that the called routine expects:

   v - value, i.e., call-by-value where the contents of the argument list entry is itself the argument of the indicated data type. Note: Call-by-value argument list entries are always allocated as a longword. The quadword data types can be used as values only for function values, never as a formal parameter. Note: the VAX-11 calling standard requires that <access type> must be r whenever <arg mechanism> is v, except for function values where <access type> is always w and <arg mechanism> is usually v.

   r - reference, i.e., call-by-reference where the contents of the argument list entry is the longword address of the argument of the indicated data type. If the argument is a scalar of the indicated data type or is a label, <arg form> must be absent. If the argument is an array, <arg form> must be present.

   d - descriptor, i.e., call-by-descriptor where the contents of the argument list entry is the longword address of a descriptor. The descriptor is two or more longwords that specify further information about the argument, see the System Reference Manual Appendix C. Note: when <arg mechanism> is d, <arg form> must be present to indicate the type of descriptor.

5. <arg form> is a letter denoting the form of the argument:

   Null means scalar of indicated data type.

   a - array reference or array descriptor, i.e., call-by-reference or call-by-descriptor as indicated by <arg mechanism>. For array call-by-reference the contents of the argument list entry is the address of an array of items of the indicated data type. The length is fixed, implied by entries in the array, e.g., a control block, determined by another argument, or specified by prior agreement. For array call-by-descriptor, the contents of the argument list entry is the longword address of an array descriptor block see the System Reference Manual Appendix C.

   s - string descriptor, i.e., call-by-descriptor where the contents of the argument list entry is the longword address of a two longword string descriptor. The descriptor contains the length, data type, and address of the string. When the string is written neither the length nor the address fields in the descriptor are modified and the string is filled with trailing spaces or a separate argument is updated with the written length.

v - varying string descriptor, i.e., call-by-descriptor where the contents of the argument list entry is the longword address of a three longword string descriptor. The descriptor contains length, data type, address, and maximum length. See Appendix C of the System Reference Manual. When the string is written, the length field of the descriptor is also modified but the address and maximum length fields are unaltered.

d - dynamic string descriptor, i.e., call-by-descriptor where the contents of the argument list entry is the longword address of a two longword string descriptor of the same format as s. However, when the string is written, both the length and address fields may be modified. Space is allocated dynamically by routines in the procedure library and is garbage collected periodically

p - Procedure descriptor, i.e., call-by-descriptor where the contents of the argument list entry is the longword address of a two longword procedure descriptor. The descriptor contains the address of the procedure and the data type that the procedure returns if it is a function. <access type> must be c, f, j, or s.

13.2.2 Optional Arguments And Default Values

Optional arguments are enclosed in square brackets, e.g. CALL FOR$READ_SU (unit.rb.v [,err.j.rl [,end.j.rl]]). The caller may omit optional parameters at the end of a parameter list by passing a shortened list. The caller may omit optional parameters anywhere by passing a 0 value as the contents of the argument list entry. A caller may not omit a parameter that is not indicated as optional. The called procedure is not obligated to detect such a programming error. An equal sign (=) after an argument inside square brackets indicates the default value if the argument is omitted. For example, success.wlc.v = SYS$DELLOG (lognam.rt.ds [,tblflg.rb.v=0]).

13.2.3 Repeated Arguments

Arguments or pairs of arguments that may be repeated one or more times are indicated inside braces, e.g. CALL FOR$OPEN ({keywd.rw.v, info.rl.v}). Repeated arguments that may be omitted entirely are indicated inside braces inside square brackets, e.g. CALL FOR$CLOSE ([{logical_unit.rl.v}]).

13.2.4 Examples

```
Sine_of angle.wf.v = MTH$SIN (angle_in_radians.rf.r)

CALL FOR$READ_SF (unit.rb.v, format.mbu.ra [,err.j.rl [,end.j.rl]])
```

Note: That (1) end may be omitted, (2) err and end may both be omitted. However, unit and format must always be present. The argument count byte in the argument list specifies how many arguments are present. Alternatively err, end, or both could have a 0 argument list entry in the above.

Common combinations are:

Completion code:	Status.wlc.v =...
longword call-by-value input arg:	no of pages.rlu.v
address of an array of signed words for input:	array.rw.ra
address of a control block:	fab.mz.ra
address of a precompiled format statement:	format.rbu.ra
label to jump to:	error_label.j.r
floating input call-by-reference arg:	angle_in_rad.rf.r
floating complex call-by-reference input arg:	angle.rfc.r
read only Fortran character string:	string rt.ds
BASIC character string to be written:	string.wt.dd

13.2.5 Summary Chart Of Notation

<name>.<access type><data type>.<arg mechanism><arg form>

<access type>		<data type>	
r	Read	z	Unspecified
m	Modify		
w	Write	v	Bit (variable bit field)
j	RET and JMP	bu	Byte Logical (unsigned)
c	RET and CALL	c	Single character
s	sub CALL	u	Smallest unit for addressable storage
f	function CALL	wu	Word Logical (unsigned)
		lu	Longword Logical (unsigned)
		a	Absolute virtual address
		cp	Character Pointer
		lc	Longword containing a completion code
		qu	Quadword Logical (unsigned)
		b	Byte Integer (signed)
		arb	Byte-sized relative virtual address
		w	Word Integer (signed)
		h	Integer value for counters
		arw	Word-sized relative virtual address
		l	Longword Integer (signed)
		g	General value
		arl	Longword-sized relative virtual address
		q	Quadword Integer (signed)
		f	Single-Precision Floating
		d	Double-Precision Floating
		fc	Complex (Floating)
		dc	Double-Precision Complex
		t	text (character) string
		nu	Numeric string, unsigned
		nl	Numeric string, left separate sign
		nlo	Numeric string, left overpunched sign
		nr	Numeric string, right separate sign
		nro	Numeric string, right overpunched sign
		nz	Numeric string, zoned sign
		p	Packed decimal string
		x	Data type indicated in descriptor

<arg mechanism>		<arg form>	
v	Value	<null>	scalar
r	Reference	a	array
d	Descriptor	s	fixed string
		v	varying length string
		d	dynamic string
		p	procedure

[End of Chapter 13]

TITLE: BLISS Transportability Guidlines -- Rev 3

Specification Status: draft

Architectural status: Under ECO Control

File: SE14F3.FNO

PDM #: not used

Date: 21-Feb-77

Superseded Specs: none

Author(s): P. Marks, R. Murray, I. Nassi

Typist; G. Hesley, R. Murray

Reviewer(s): R. Brender, D. Cutler, P. Conklin, T. Hastings, S. Hawkinson, D. Tolman, R. Winslow

Abstract: Chapter 14 addresses the process of writing transportable BLISS programs. Tools and techniques are discussed in detail.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Original	I. Nassi	22-Dec-76
Rev 2	Skipped	I. Nassi	25-Jan-77
Rev 3	SEM Integration	R. Murray	21-Feb-77

;change history

Rev 2 to Rev 3

1. Software Engineering Manual integration, this document added as a chapter

Rev 1 to Rev 2

1. revision 2 skipped to align revision histories on all chapters to Rev 3

[end of SE14R3.RNO]

CHAPTER 14

BLISS TRANSPORTABIBILTIY GUIDLINES

21-FEB-77 -- Rev 3

This Chapter addresses the task of writing transportable programs. It is shown that the writing of such code is much easier if considered from the beginning of the project. The properties which cause a program to loose transportablilty are explored. Techniques by which the programmer may avoid these pitfalls are discussed.

14.1 INTRODUCTION

14.1.1 Purpose And Goals

The purpose of this document is to facilitate the process of writing transportable BLISS programs, that is, BLISS programs intended to be executed on architecturally different machines. There are various kinds of solutions to the problem of transportability, each requiring different levels of effort. We feel free in recommending various kinds of solutions. When program text should be rewritten, for example, we suggest doing so. However, it is our belief that large portions of programs can be written which will require absolutely no modification in order to be functionally equivalent over differing architectures. The levels of solutions we see, in order of decreasing desirability, are:

- o no change is needed to program text - transportability is perfectly straightforward.
- o parameterization solves the transportability problem - the program makes use of some features that have an analog on all the other architectures.
- o parallel definitions are required - either programs make use of features of an architecture that do not have analogs across all other architectures, or different, separately transportable aspects of a program interact in non-transportable ways.

The goal is to make transportability as painless as possible, which means that the effort needed in transporting programs should be minimized.

Central to the ideas presented here is the notion that transportability is more easily accomplished if considered from the beginning. Transporting programs after they are running becomes a much more complex task. We suggest frequently running parallel compilations, for instance. It is fortunate therefore, that with the right tools and techniques, transportability is not difficult to achieve. We would also like to point out that the first program is the hardest. Before undertaking a large programming project, we suggest writing and transporting a less ambitious program.

These guidelines are the result of a concentrated study of the problems associated with transportability. We make no claim that these guidelines are complete. We do claim that some of what is contained here will be non-obvious to programmers. We have attempted to identify those areas which, if the programmer is not forewarned, will cause problems. We will be suggesting solutions to all identified problems.

Many of the problems that are discussed here have solutions that are currently being incorporated into the BLISS language, so another way of viewing this document is as a partial rationale for some of these language changes, and a rationale for the definition of BLISS-16 and BLISS-36.

14.1.2 Organization

These guidelines are organized into three sections. The section on General Strategies discusses some high level approaches to writing transportable BLISS software. The section on Tools describes various features of the BLISS language that can be used in solving transportability problems. The section on Techniques analyzes various transportability problems and suggests solutions to them.

14.2 GENERAL STRATEGIES

14.2.1 Introduction

This section presents certain gross or global considerations that are important to the writing of transportable BLISS programs, namely:

- Isolation, and
- Simplicity

14.2.2 Isolation

The following maxim should be kept in mind when you are designing and/or coding a program that is to be transported:

- If it is NON-transportable, isolate it.

You will probably encounter situations for which it is desirable to use machine-specific constructs in your BLISS program. In these cases, simply isolating the constructs will facilitate any future movement of the program to a different machine.

In most cases, only a small percentage of the program or system will be sensitive to the machine on which it is running. By isolating those sections of a program or a system, the effort involved in transporting the program will be confined mainly to these easily identifiable, machine-specific sections.

Specifically, follow these rules:

- If machine-specific data is to be allocated - place the allocation in a separate MODULE or in a REQUIRE file.
- If machine-specific data is to be accessed - place the access in a ROUTINE or in a MACRO and then place the ROUTINE or MACRO in a separate MODULE or in a REQUIRE file.
- If a machine-specific function or instruction is to be used, isolate it by placing it too in a REQUIRE file.
- If it is impossible or impractical to isolate this part of your program from its module, comment it heavily. Make it very obvious to the reader that this code is non-transportable.

The above rules are applicable in the local context of a routine or module. In a larger or more global context (for instance, in the design of an entire system) isolation is implemented by the technique

of modularization.

By separating those parts of the system which are machine or operating system dependent from the rest of the system, the task of transporting the entire system is simplified. It becomes a matter of recoding a small section of the total system. The major portion of the code (if written in a transportable manner) should easily make the move to a new machine with a minimum of re-coding effort.

BLISS is a language which facilitates both the design and programming of programs and systems in a modular fashion. This feature should be taken advantage of when writing a transportable system.

14.2.3 Simplicity

A basic concept in writing transportable BLISS software is simplicity - simplicity in the use of the language.

BLISS was originally developed for the implementation of systems software. As a result of this, BLISS is nearly unique among high-level programming languages in that it allows ready access to the machine on which the program will be running. The programmer is allowed to have complete control over the allocation of data, for example.

The same language features that allow access to underlying features of the hardware are very often used to excess. In order to identify those features of the language causing a program to be non-transportable, it is often the case that such features be invoked explicitly, making the program inherently more complex. Reducing the complexity of data allocation, for example, results in a transportable subset of the BLISS language. This reduction of complexity is one of the basic themes that runs through the guidelines.

In effect, the coding of transportable programs is a simpler task because the number of options available has been reduced. Simplicity in the coding effort is one of the reasons for the development of higher-level languages like BLISS. The use of the defaults in BLISS will result in programs which are much more easily transported.

14.3 TOOLS

This section on tools presents various language features that provide a means for writing transportable programs. These features are either intrinsic to BLISS or have been specifically designed for transportability/software engineering uses.

The tools described here will be used throughout the companion section on techniques.

14.3.1 Literals

Literals provide a means for associating a name with a compile-time constant expression. In this section, we will consider some built-in literals which will aid us in writing transportable programs. In addition, we will discuss restrictions on user-defined literals.

14.3.2 Predeclared Literals

One of the key techniques in writing transportable programs is parameterization. Literals are a primary parameterization tool. The BLISS language has a set of predeclared, machine specific literals that can be most useful.

These literals parameterize certain architectural values of the three machines. The values of the literals are dependent on the machine that the program is currently being compiled for. Here are their names and values:

Description	Literal Name	10/20	VAX-11	11
Bits per addressable unit	%BPUNIT	36	8	8
Bits per address value	%BPADDR	18	32	16
Bits per BLISS value	%BPVAL	36	32	16
Units per BLISS value	%UPVAL	1	4	2

The names beginning with '%' are the literal names that can be used. These literal names will be used throughout the guidelines.

Bits per value is the maximum number of bits in a BLISS value. Bits per unit is the number of bits in the smallest unit of storage that can have an address. Bits per address refers to the maximum number of bits an address value can have. Units per value is the quotient %BPVAL/%BPUNIT. It is the maximum number of addressable units associated with a value.

We can derive other useful values from these built-in literals. For example:

```
LITERAL
        HALF_VALUE = %BPVAL / 2;
```

defines the number of bits in half a word (half a longword on VAX-11).

14.3.2.1 User Defined Literals -

A literal is not strictly speaking a self-defining term. The value and restrictions associated with a literal are arrived at by assigning certain semantics to its source program representation. It is convenient to define the value of a literal as a function of the characteristics of a particular architecture, which means that there are certain architectural dependencies inherent in the use of literals.

Because the size of a BLISS value determines the value and/or the representation of a literal, there are some transportability considerations. BLISS value (machine word) sizes are different on each of the three machines. On VAX-11, the size is 32 bits; on the 10/20 systems, it is 36; and the 11 value is 16.

There are two types of BLISS literals: numeric-literals and string-literals. The values of numeric-literals are constrained by the machine word size. The ranges of values for a signed number, i, are:

VAX-11: $-(2^{31}) \leq i \leq (2^{31}) - 1$

10/20: $-(2^{35}) \leq i \leq (2^{35}) - 1$

11: $-(2^{15}) \leq i \leq (2^{15}) - 1$

ALL: $-(2^{(\%BPVAL-1)}) \leq i \leq (2^{(\%BPVAL-1)})-1$

Double precision floating point numbers (%D'number' in BLISS-32) are not supported in BLISS-36 or in BLISS-16.

A numeric literal, %C'single-character', has been implemented. Its value is the ASCII code corresponding to the character in quotes and when stored, it is right-justified in a BLISS value (word or longword). A more thorough discussion of its usage can be found in the section entitled: "Data: Character Sequences".

There are two ways of using string-literals: as integer-values and as character strings. When string-literals are used as values, they are not transportable. This arises out of the representational differences and from differing word sizes. The following table illustrates these potential differences for an %ASCII type string literal:

	VAX-11	10/20	11
Maximum number of characters.	4	5	2
Character placement.	right to left	left to right	right to left

This type of string literal usage and also its use as a character string are discussed in the section entitled: "Data: Character Sequences".

14.3.3 MACROS

BLISS macros can be an essential tool in the development of transportable programs. Because they evaluate (expand) during compilation, it is possible to tailor a program to a specific machine.

A good example can be found in the section on structures. There, two macros are developed which are completely transportable. The macros can determine the number of addressable units needed for a vector of elements, where the element size is specified in terms of bits.

There are also pre-defined machine conditionalization macros available. These macros can be used to compile selectively only certain declarations and/or expressions depending on which compiler is being run.

Their definitions for the bliss-32 set are:

```
MACRO
        %BLISS16[] = % ,
        %BLISS36[] = % ,
        %BLISS32[] = %REMAINING % ;
```

There are analogous definitions for the other machines. The net effect is that in the BLISS-32 compiler, the arguments to %BLISS16 and %BLISS36 will disappear, while arguments to %BLISS32 will be replaced by the text given in the argument list.

14.3.4 Module Switches

A module switch and a corresponding on-off switch are provided to aid in the writing of transportable programs. This switch, LANGUAGE, is provided for two reasons:

- To indicate the intended transportability goals of a module and
- To provide diagnostic checking of the use of certain language features.

The programmer can therefore indicate the target architectures (environments) for which a program is intended.

Diagnostic checking consists of the compiler determining whether certain language features are available for all of the intended target environments.

The LANGUAGE switch may be used in the module header or switches declaration to designate which of the several BLISS processors are intended to compile the module.

The syntax is:

```
LANGUAGE (language-type ,...)
```

where language-type is any combination of BLISS36, BLISS16 or BLISS32.

If no LANGUAGE switch is specified, the default is all three languages, and as a consequence, only the most restricted language facilities are made available.

Each compiler will give a warning diagnostic if its own language is not in the list of language-types.

Within the scope of a language switch, each compiler will give a warning diagnostic for any language construct which is not in the intersection of the specified set of languages.

NOTE

As of this writing the particular language features that will be subject to diagnosis have yet to be detailed. However, using it now will serve to document the program, and to make the program immune to compiler enhancements that restrict certain features under certain switch settings.

Here is an example of how the LANGUAGE switch would be used:

```
MODULE FOO(...,LANGUAGE(BLISS36, BLISS16, BLISS32),...) =
BEGIN

                ...
                ...
                ...

BEGIN

!+
! BLISS16 no longer in effect.
!-

SWITCHES
        LANGUAGE(BLISS36, BLISS32);
                ...
                ...
                ...

        Any use of language features, within
        this block, which are specific to
        BLISS16 will result in a diagnostic
        warning.
        The compilation of this section
        of code by a BLISS-16 compiler will
        result in a diagnostic warning.

                ...
                ...
                ...

END;

!+
!  All three language settings are restored.
!-
```

14.3.5 Reserved Names

The following page contains a list of BLISS reserved names. The list represents the union of reserved names in all three BLISS dialects. Hence, if one is writing a transportable program, one should avoid using any of these names as a user-defined name, since such use results in a compiler diagnostic. Items marekd with an asterisk should not be used when writing code intended to be transportable.

*ADDRESSING_MODE
*ALIGN
ALWAYS
AND
BEGIN
BIND
BIT
*BUILTIN
BY
*BYTE
CASE
CODECOMMENT
COMPILETIME
DECR
DECRA
*DECRU
DO
ELSE
ELUDOM
ENABLE
END
EQL
EQLA
*EQLU
EQV
EXITLOOP
EXTERNAL
FIELD
FORWARD
FROM
GEQ
GEQA
*GEQU
GLOBAL
GTR
GTRA
*GTRU
IF
INCR
INCRA
*INCRU
INITIAL
INRANGE
KEYWORDMACRO
LABEL
LEAVE
LEQ
LEQA
*LEQU
LIBRARY
LINKAGE
LITERAL
LOCAL
*LONG
LSS
LSSA
*LSSU
MACRO
MAP
MOD
MODULE
NEQ
NEQA
*NEQU
NOT
NOVALUE
OF
OR
OTHERWISE
OUTRANGE
OWN
PLIT
PRESET
*PSECT
RECORD
REF
REGISTER
REP
REQUIRE
RETURN
ROUTINE
SELECT
SELECTA
SELECTONE
SELECTONEA
*SELECTONEU
*SELECTA
SET
*SHOW
*SIGNED
STACKLOCAL
STRUCTURE
SWITCHES
TES
THEN
TO
UNDECLARE
UNTIL
UPLIT
*VOLATILE
*WEAK
WHILE
WITH
*WORD
XOR

14.3.6 REQUIRE Files

REQUIRE files are a way of gathering machine specific declarations and/or expressions together in one place.

In many cases, it will be either impossible or unnecessary to code a particular BLISS construct (e.g. routines, data declarations, etc.) in a transportable manner. Developing parallel REQUIRE files, one for each machine, can often provide a solution to transporting these constructs.

For example, if a certain set of routines are very machine specific, then the solution may be to code two or three functionally equivalent routines, one for each machine type, and segregate them each in their own REQUIRE file.

Each BLISS compiler has a pre-defined search rule for REQUIRE file names based on their file types. Each compiler will search first for a file with a specific file type, then it will search for a file with the file type '.BLI'.

The search rules for each compiler are:

Compiler	1st	2nd
BLIS36	.B36	.BLI
BLIS16	.B16	.BLI
BLIS32	.B32	.BLI

Hence, the following REQUIRE declaration:

```
REQUIRE
        'IOPACK';                   ! I/O Package
```

will search for IOPACK.B36, IOPACK.B16 or IOPACK.B32, depending on which compiler is being run. Failing that it will look for IOPACK.BLI.

Inherent in these search rules is a naming convention for REQUIRE files. If the file is transportable, give it the file type '.BLI'. If it is specific to a particular dialect, give it the corresponding file type (e.g. '.B36').

14.3.7 ROUTINES

The key to transportability is the ability to identify properties of an environment, abstract the property by giving it a name, and then define the semantics of the property in all applicable environments. The closed subroutine has long been regarded as the principal abstraction mechanism in programming languages. With BLISS, we see other abstraction mechanisms being used, like structures, macros, literals, require files, etc., but the routine can still be easily used as a transportability abstraction mechanism.

For instance, when designing a system of transportable modules which uses the concept of floating point numbers and associated operations, there will be a need to perform floating point arithmetic. The question naturally arises as to the environment in which the arithmetic should be done. If the floating point arithmetic resides entirely in a well-defined set of routines, and no knowledge of the various representations of floating point numbers is used except through these well defined interface routines, then it becomes possible to perform "cross-arithmetic", which becomes highly desirable when writing cross-compilers, for instance. Even if the ability to perform cross-arithmetic is not desired, isolating floating point operations in routines is a good idea since these routines can then be reused more easily in another project. A little thought will indicate that the floating point routines themselves have to be transportable if they are going to perform cross-arithmetic, but need not be transportable if cross arithmetic is a non-goal.

The principal objection to using routines as an abstraction mechanism is that the cost of calling a procedure is non-trivial, and that cost is strictly program overhead. Composing this sort of abstraction in the limit will produce serious performance degradation. For this reason, a programmer should probably try not to use the routine as an abstraction mechanism if a small amount of forethought will be sufficient to enable the writing of a single transportable module.

14.4 TECHNIQUES

This section on techniques shows you how to write transportable programs. The section is organized in dictionary form by BLISS construct or concept. Each sub-section contains:

- A discussion of the construct or concept.
- Transportability problems that its use may engender.
- Specific guidelines and restrictions on the use of the construct or concept.
- Examples - both transportable and non-transportable.

The examples, in all cases, attempt to use the tools described in the TOOLS section.

14.4.1 Data

14.4.1.1 Introduction -

This section deals with the allocation of data in a BLISS program. For the purposes of this section we do not deal with character sequence (string) data or address data. These types of data are discussed in their own sections (See: "Data: Addresses and Address Calculation" and "Data: Character Sequences"). Primarily, we discuss the allocation of scalar data (e.g. counters, integers, pointers, addresses, etc.) A presentation of more complex forms of data can be found in the sections entitled: "Structures and Field-Selectors" and "PLITs and Initialization". First there is a discussion of transportability problems encountered due to differing machine architectures. Next a discussion of the BLISS allocation-unit attribute is presented. Finally, a discussion of other BLISS data attributes that must be considered when writing transportable programs is discussed.

14.4.1.2 Problem Genesis -

The allocation of data (via the OWN, LOCAL, GLOBAL, etc. declarations) tends to be one of the most sensitive areas of a BLISS program in terms of transportability. This problem of transporting data arises chiefly from two sources:

- The machine architectures and
- The flexibility of the BLISS language.

When we are considering writing a BLISS program that will be transported to another machine, we are confronted with the problem of allocating data on (at least two) architecturely different machines.

Although we have already discussed differing word sizes, there are further differences. On the VAX-11 machine data may be fetched in longwords (32 bits), in words (16 bits) and in bytes (8 bits); on the 11, both words and bytes may be fetched. Only 36-bit words on the 10/20 systems may be directly fetched (i.e. without a byte pointer).

If we were writing our program in MACRO-10 or MARS we would not consider these differences to be important - clearly, our assembly language program was not intended to be transportable.

What decisions, however, must the BLISS programmer make in the transportable allocation of data? Need he or she be concerned with how many bits are going to be allocated?

These questions (and their answers) can be complicated by the other chief source of data transportability problems, namely the BLISS language itself.

BLISS is different than many other higher-level languages in that it allows ready access to machine-specific control, particularly in storage allocation. This is fortunate for the programmer who is writing highly machine-specific, efficient software. This programmer needs much more control over exactly how many bits of data will be used. This feature of BLISS, however, can complicate the decisions that need to be made by the BLISS programmer who is writing a transportable program. Does he or she allocate scalars by bytes, or by words, or by longwords?

14.4.1.3 Transportable Declarations -

Consider the following simple example of a data declaration in BLISS-32:

```
OWN
        PAGE_COUNTER: BYTE;     ! Page counter
```

The programmer has allocated one byte (8 bits) for a variable named PAGE_COUNTER. No matter what his or her intentions were in requesting only one byte of storage, this declaration is non-transportable. The concept of BYTE (in this context) does not exist on the 10/20 systems. In fact, in BLISS-36 the use of the word BYTE results in an error message.

If this declaration had been originally coded as:

```
OWN
        PAGE_COUNTER;           ! Page counter
```

then this could have been transported to any of the three machines. The functionality (in this case, storing the number of pages) has not been lost. We allowed the BLISS compiler to allocate storage by default by not specifying any allocation-unit in the OWN declaration. In all the BLISS dialects the default size for allocation-unit consists of %BPVAL bits. Thus our first transportable guideline is:

- Do not use the allocation-unit attribute in a scalar data declaration.

Besides the allocation-unit there are other attributes that may present transportability problems if used. In particular, when allocating data:

- Do not use the following attributes:

  Extension (SIGNED and UNSIGNED),
  Alignment,
  Volatile,
  Range,
  Weak

  which is to say: think twice before you write a declaration. Do you really need to specify any data attributes other than structure attributes?

The Extension-attribute specifies whether the sign bit is to be extended in a fetch of a scalar. This attribute is meaningful only on VAX-11 and is not supported by BLISS-36 or BLISS-16. No sign extension can be performed if the allocation unit is not specified.

The Alignment-attribute tells the compiler at what address boundary a data segment is to start. It is not supported in BLISS-36 or BLISS-16; hence, it is non-transportable. Suitalbe default alignments are available dependent on the size of the scalar.

The Volatile-attribute notifies the compiler that code to fetch the contents of this data segment must be generated anew for each fetch in the BLISS program. It is not supported in BLISS-36 or BLISS-16 and will result in a compiler diagnostic.

The Range-attribute specifies the number of bits needed to represent the value of a literal that is declared global in a separately compiled module. The STARLET linker is the only linker that currently supports external literals.

The Weak-attribute is a STARLET-specific attribute and is not supported by BLISS-36 or BLISS-16. It can not be used in a transportable program.

These guidelines are relatively simple, yet they should relieve the BLISS programmer of needing to worry about how the program data will actually be allocated by the compiler. There is often very little reason to specify an allocation-unit or any attributes. The default values are almost always sufficient.

In the case of scalar data, the use of the default allocation-unit will sometimes result in the allocation of more storage than is strictly necessary. This gain in program data size (which, in most instances, is small) should be weighed against a decrease in fetching time for a particular scalar value, and the knowledge that because of the default alignment rules, no storage savings may, in fact, be realized.

In the BLISS language, the default size of %BPVAL bits was chosen (among other reasons) because this is the largest, most efficiently accessed unit of data for a particular machine. Which is to say, the

saving of bits does not necessarily mean a more efficient program.

There will undoubtedly be cases where it is impossible to avoid the use of one or more of the above attributes. In fact, it may be desirable to take advantage of a specific machine feature. In these cases follow this guideline:

- Conditionalize and/or heavily comment the use of declarations which may be non-transportable.

This guideline is the "escape-hatch", if you will, in this set of guidelines. It should only be used sparingly and where justified. To use it often will only result in more code that will need to be re-written when the program has to be transported to another machine - and that's not our goal.

14.4.2 Data: Addresses And Address Calculations

14.4.2.1 Introduction -

This section will discuss address values and calculations using address values. First, there will be a presentation of the problems that might occur when using an address or the result of an address calculation as a value. A transportable solution to some of these problems is then presented. Next, a discussion of the need for address forms of the BLISS relational operators and control expressions and how and when to use them will be presented. Finally, some important differences in the interpretation of address values between BLISS-10 and BLISS-36 are discussed.

14.4.2.2 Addresses And Address Calculations -

The value of an undotted variable name in BLISS is an address. In most cases, this address value is used only for the simple fetching and storing of data. When address values are used for other purposes, we must be concerned with the portability of an address or an address calculation. By address calculation we mean any arithmetic operations performed on address values.

The primary reason for our concern is the different sizes (in bits) of addressable units, addresses, and BLISS values (machine words) on the three machines. For convenience in writing transportable programs, these size values have been parameterized and are now predeclared literals. A table of their values can be found in the section entitled: "Literals".

To see how these size differences can have an effect on writing transportable programs, let's consider a common type of address expression; namely an expression that computes an address value from a base (a pointer or an address) and an offset. That is, some expression of the form:

... base + index ...

Now consider the following BLISS assignment expression using this form of address calculation:

```
OWN
        ELEMENT_2;
.
.
.

ELEMENT_2 = .(INPUT_RECORD + 1);
```

The intent (most likely) was to access the contents of the second value in the data segment named INPUT_RECORD and to place that value in an area pointed to by ELEMENT_2. The effect, however, is different on each machine as we shall see.

By adding 1 to an address (in this case, INPUT_RECORD) we are computing the address of the next addressable unit on the machine. In BLISS-32 and BLISS-16 this would be the address of the next byte (8 bits), but in BLISS-36 this would be the address of the next word (36 bits). This is probably not a transportable expression because of the different sizes of the addressable units and the resultant values.

Based on the above example, we introduce the following guideline:

- When a complex address calculation is not an intrinsic part of the algorithm being coded, do not write it outside of a structure declaration.

There is a way, however, of making such an address calculation transportable. It involves the use of the values of the predeclared literals. In the last example, if the index had been 4 in BLISS-32 or 2 in BLISS-16 then in each case we would have accessed the next word.

We need to calculate a multiplier that will have a value of 4 in BLISS-32, 2 in BLISS-16 and 1 in BLISS-36. Such a multiplier already exists as another predeclared literal. Its definition is %BPVAL/%BPUNIT, and it is called %UPVAL.

Using this literal in our example we would have:

```
ELEMENT_2 =
        .(INPUT_RECORD + 1 * %UPVAL);
```

The address expression is now tranportable.

This last example raises an interesting point. If an address calculation of this form is used then it is very likely that the data segment should have had a structure such as a VECTOR, BLOCK or BLOCKVECTOR associated with it. The last example could have then been coded as:

```
OWN
    INPUT_RECORD:
        FLEX_VECTOR[RECORD_SIZE,%BPVAL],
    ELEMENT_2;
.
.
.

ELEMENT_2 = .INPUT_RECORD[1];
```

The transportable structure FLEX_VECTOR and a more thorough discussion of structures can be found in the section entitled: Structures and Field Selectors.

14.4.2.3 Relational Operators And Control Expressions -

The previous example illustrated the use of address values in the context of computations. Other common uses of addresses are in comparisons (testing for equality, etc.) and as indices in loop and select expressions. The use of address values in these contexts points to another set of differences found amongst the three machines.

In BLISS-32 and BLISS-16, addresses occupy a full word (%BPADDR equals %BPVAL) and unsigned integer comparisons must be performed. However, in BLISS-36, addresses are smaller than the machine word (18 versus 36 bits) and signed integer operations are performed for efficiency reasons.

It can be seen that to perform a simple relational test of address values:

```
... ADDRESS_1 LSS ADDRESS_2 ...
```

requires two different interpretations. This expression would evaluate correctly on the 10/20 systems. But, on the VAX-11 and 11 machines, the following would have had to have been coded for the comparison to have been made correctly:

```
... ADDRESS_1 LSSU ADDRESS_2 ...
```

Another type of relational operator, designed specifically for address values, is needed. Such operators exist and are referred to as address-relational-operators. BLISS-36, BLISS-16 and BLISS-32 have, in fact, a full set of them (e.g. LSSA, EQLA, etc.) which support address comparisons.

In BLISS-16 and BLISS-32, the address-relationals are equivalent to the unsigned-relationals. In BLISS-16, the address-relationals are equivalent to the signed-relationals. For all practical cases, a user need not be concerned with this, since this "equivalencing" permits equivalent address comparisons to be performed across architectures.

In addition, there are address forms of the SELECT (SELECTA), SELECTONE (SELECTONEA), INCR (INCRA) and DECR (DECRA) control expressions. The following guidelines establish a usage for these operators and contol expressions:

- If address values are to be compared, use the address form of the relational operators.
- If an address is used as an index in a SELECT, SELECTONE, INCR or DECR expression, use the address form of these control expressions.

A violation of either of these guidelines can have unpredictable results.

14.4.2.4 BLISS-10 Addresses Versus BLISS-36 Addresses -

There is a fundamental conceptual change from BLISS-10 to BLISS-36 in the defined value of a name. BLISS-10 defines the value of a data segment name to be a byte pointer consisting of the address value in the low half of a word, and position and size values of 0 and 36 in the high half of the word. BLISS-36, however, defines the value as simply the address in the low half and zeros in the high half. This change was made solely for reasons of transportability, since it allows BLISS to assign uniform semantics to an address.

The fetch and assignment operators are redefined to use only the address part of a value. Thus the expressions:

```
Y = .X;
Y = F(.Y) + 2;
```

are the same in both BLISS-10 and BLISS-36, but

```
Y = X;
```

assigns a different value to Y in BLISS-36 and in BLISS-10.

Field selectors are still available but must be thought of as extended operands to the fetch and assignment operators, instead of as value producing operators applied to a name. Thus the meaning of:

```
Y<0,18> = .X<3,7>;
```

is unchanged, but

```
Y = X<3,7>;
```

is invalid. Moreover, it is highly recommended that field selectors never appear outside of a structure declaration, since bit position and size are apt to be highly machine dependent. A more thorough discussion can be found in the section entitled: Structures and Field Selectors.

14.4.3 Data: Character Sequences

14.4.3.1 Introduction -

This section will discuss the use of character sequences (strings) in BLISS programs. Historically, there has been no consistent method for dealing with strings and the functions operating upon them. Ad hoc string functions have been the rule, having been implemented by individuals or projects to suit their particular needs. This section will begin by looking at quoted strings in two different contexts. We will discuss transportability problems associated with quoted strings, and guidelines for their use.

Quoted strings are used in two different contexts:

- o as values (integers) and
- o as character strings

14.4.3.2 Usage As Numeric Values -

The use of quoted strings as values (in assignments and comparisons) illustrates the problem of differing representations on differing architectures. Describing the natural translation of a string literal for each architecture will illustrate the problem. For example, consider the following code sequence:

```
OWN
        CHAR_FOO;       ! To hold a literal

CHAR_FOO = 'FOO';
```

A natural interpretation for BLISS-32 to use is that one longword would be allocated and the three characters would be assigned to increasing byte addresses within the longword. In memory, the value of CHAR_FOO would have the following representation:

```
CHAR_FOO:   / 00 O O F / (32)
```

BLISS-16 would not allow this assignment because only two ASCII characters are allowed per string-literal. This restriction arises from the fact that BLISS-16 works with a maximum of 16-bit values and three 8-bit ASCII characters require 24 bits.

On the 10/20 systems a word would be allocated and the characters would be positioned starting at the high-order end of the word. Thus the string-literal would have the following representation in memory:

```
CHAR_FOO:   / F O O 00 00 0 / (36)
```

Even if the 10/20 string-literal had been right-justified in the word, it still would not equal the VAX-11 representation, numerically. So, in fact, the following would not be transportable:

```
WRITE_INTEGER( 'ABC' );
```

since 'ABC' is invalid syntax in BLISS-16, has the value -33543847936 in BLISS-36, and the value 4276803 in BLISS-32.

Based on these problems with representation our first guideline is:

- Do not use string-literals as numeric values.

In those cases where it is necessary to perform a numeric operation (e.g. a comparison) with a character as an argument, you must use the %C form of integer literal. This literal takes one character as its argument and returns as a value the integer index in the collating sequence of the ASCII character set, so that:

```
%C'B' = %X'42' = 66
```

The %C notation was introduced to standardize the interpretation of a quoted string across all possible ASCII-based environments. %C'quoted-character' can be thought of as "right-adjusting" the character in a bit string containing %BPVAL bits.

14.4.3.3 Usage As Character Strings -

The necessity of using more than one character in a literal leads us to the other situation in which quoted strings are used: as character strings.

To facilitate the allocation, comparison and manipulation of character sequences, a built-in character sequence function package has been introduced to the BLISS language. It has been implemented in BLISS-32 and BLISS-36 and plans exist to implement it in BLISS-16.

These built-in functions provide a very complete and powerful set of operations on characters. Our next guideline is:

- You must use the built-in function package when allocating and operating upon character sequences. This is the only way one can guarantee the portability of strings and string operations.

A more detailed description of these functions can be found in the Character Handling Functions chapter of the BLISS-VAX Language Guide, Second Edition.

14.4.4 PLITs And Initialization

14.4.4.1 Introduction -

This section is primarily concerned with PLITs and their uses. First, there is general discussion of PLITs and the contexts in which they often appear. A presentation of how scalar PLIT items should be used follows. Next, the problems involved in using string literals in PLITs and suggested guidelines for their use are presented. Finally, the use of PLITs to initialize data segments will be illustrated by the development of a transportable table of values.

14.4.4.2 PLITs In General -

Because BLISS values are a maximum of a machine word in length, any literal that requires more than a word for its value needs a separate mechanism, and that mechanism is the PLIT (or UPLIT). Hence, PLITs are a means for defining references to multi-word constants. PLITs are often used to initialize data segments (e.g. tables) and are used to define the arguments for routine calls.

PLITs themselves are transportable; however, their constituent elements and their machine representation are not always transportable.

A PLIT consists of one or more values (PLIT items). PLIT items may be strings, numeric constants, address constants or any combination of these last three, providing that the value of each is known prior to execution time.

14.4.4.3 Scalar PLIT Items -

The first transportability problem that might be encountered with the use of PLITs is in the specification of scalar PLIT items. As with any other declaration of scalar items (pointers, integers, addresses, etc.) it is possible to define them with an allocation-unit attribute. For example, in BLISS-32, we can specify such machine specific sizes as BYTE and LONG. Thus the following example is non-transportable and, in fact, will not compile on BLISS-36 or BLISS-16:

```
BIND
        Q1 = PLIT BYTE(1, 2, 3, LONG -4);
```

This last example provides the first PLIT guideline:

- Do not use allocation-units in the specification of a PLIT or PLIT item.

Thus, the BIND should have been coded as follows:

```
BIND
        Q1 = PLIT(1, 2, 3, -4);
```

This last guideline is necessary because of the differences in the sizes of words on the three machines, a feature of the architectures. A discussion of the role of machine architectures in the transportability of data can be found in the section entitled: "Data". Further guidelines are presented in the section entitled: "Intializing Packed Data".

14.4.4.4 String Literal PLIT Items -

The next guideline is based on the representation of PLITs in memory. Specifically the problem is encountered when scalar and string PLIT items appear in the same PLIT.

The difficulty arises primarily from the representation of characters on the different machines. A more thorough discussion of character representation can be found in the section entitled: "Data: Character Sequences".

Care must be exercised when strings are to be used as items in PLITs. For example, we may wish to specify a PLIT that consists of two elements: a 5-character string and an address of a routine. If we specify it as:

```
BIND
        CONABC = PLIT('ABCDE', ABC_ROUT);
```

then the VAX-11 representation is as follows:

```
CONABC:                 / D C B A /  (32)
                        /       E /  (32)
                        / address /  (32)
```

on the 11, it would be:

```
CONABC:                     / B A /  (16)
                            / D C /  (16)
                            /   E /  (16)
                        / address /  (16)
```

and the 10/20 representation would be:

```
CONABC:                 / A B C D E / (36)
                        / address   / (36)
```

The three PLITs are not equivalent. Three longwords are required for the BLISS-32 representation, four words are needed for BLISS-16, and two words are needed for the BLISS-36 representation. If we wished to access the two elements of this PLIT by the use of an address offset, we would have problems. For example, the second element (the address) is accessed by the expression:

```
... CONABC + 1 ...
```

in the BLISS-36 version, but not in the BLISS-32 or BLISS-16 versions. For the BLISS-32 version, we would need the expression:

```
... CONABC + 8 ...
```

and for BLISS-16, it would have to be:

```
... CONABC + 6 ...
```

Taking a data segment's base address and adding to it an offset (as in this case) is particularly sensitive to transportability. A discussion on the use of addresses can be found in the section entitled: "Data: Addresses and Address Calculations".

This section on addresses suggests the use of the literal, %UPVAL, to ensure some degree of transportability. Its value is the number of addressable units per BLISS value (machine word). As already discussed, in BLISS-32, the literal equals 4; in BLISS-16, it is 2; and in BLISS-36, its value is 1.

Multiplying an offset by this value can, in some cases, ensure an address calculation that will be transportable. So to access the second element in the above PLIT, one would write:

```
... CONABC + 1*%UPVAL ...
```

But this won't work for the VAX-11 representation. An offset value of 8 is needed because the string occupies two words (BLISS values). The situation is similar for the 11 version, where the string occupies 3 words and would need a offset value of 6 not 2.

The problem with this particular example (and, in general, with strings in PLITs) is not in the use of a string literal but in its position within the PLIT. Because the number of characters that will fit in a BLISS value differs on all three machines (see the section: Data: Character Sequences), the placement of a string in a PLIT will very often result in different displacements for the remaining PLIT items.

There is a relatively simple solution to this problem:

- In a PLIT there can only be a maximum of one string literal, and that literal must be the last item in a PLIT.

Following this guideline, the example should have been coded:

```
BIND
        CONABC = PLIT( ABC_ROUT, 'ABCDE');
```

and this expression:

```
... CONABC + 1*%UPVAL ...
```

would have resulted in the address of the second element in the PLIT (in this case the string).

14.4.4.5 An Example Of Initialization -

As mentioned in the beginning of this section, PLITs are often used to initialize data segments such as tables. A data segment allocated by an OWN or GLOBAL declaration can be initialized by using the INITIAL attribute. The INITIAL attribute specifies the initial values and consists of a list of PLIT items.

A good example which shows how relatively easy it is to initialize data in a transportable way is to illustrate the process one might use to build a table of employee data. Information on each employee will consist of three elements: an employee number, a cost center number and the employee's name. The employee's name will be a fixed length, 5-character field.

For example, a line of the table would contain the following information:

```
345     201        MARKS
```

Converting this line into a list of PLIT items which conform to this section's guidelines would result in the following:

```
(345, 201, 'MARKS')
```

Notice that no allocation units were specified and that the character string was specified last. We will now use this line to initialize a small table of only one line. The table will have the built-in BLOCKVECTOR structure attribute. The table declaration would look like:

```
OWN
        TABLE:
                BLOCKVECTOR[1,3]
                INITIAL(
                        345,
                        201,
                        'MARKS'
                        );
```

A problem, however, has developed. This definition would work well in BLISS-36. That is, three words would have been allocated for TABLE. The first word would have been initialized with the employee number; the second word with the cost center; and the third with the name. But the declaration would not be correct in BLISS-32 or BLISS-16, simply because not enough storage would have been allocated for all the initial values. BLISS-32 would have required 4 longwords and the BLISS-16, 5 words.

The problem arises as a result of the way in which strings are represented and allocated on the three machines (see the section: Data: Character Sequences). The solution is simple. We only need to determine the number of BLISS values (words) that will be needed for the character string on each machine. There is a function that will give this value. It is named CH$ALLOCATION and it is part of the Character Sequence Function Package. It takes as an argument the number of characters to be allocated and returns the number of words needed to represent a string of this length. We can use this value as an allocation actual in the table definition, as follows:

```
OWN
        TABLE:
                BLOCKVECTOR[1,2 + CH$ALLOCATION(5)]
                INITIAL(
                        345,
                        201,
                        'MARKS'
                        );
```

The declaration is now transportable. By using the CH$ALLOCATION function we can be assured that enough words will be allocated on each machine. No recoding will be necessary.

We are free to add other lines to the table and not be concerned with the representation or allocation of the data. Here is a larger example of the same kind of table. We won't develop it step by step, but point out and explain some of the highlights.

The example:

```
                    ...
                    ...
                    ...

        !+
        !       Table Parameters
        !-

        LITERAL
                NO_EMPLOYEES = 2,
                EMP_NAME_SIZE = 25,
                EMP_LINE_SIZE = 2 +
                    CH$ALLOCATION(EMP_NAME_SIZE);

        !+
        !       Employee Name Padding Macro
        !-

        MACRO
                NAME_PAD(NAME) =
                    NAME, REP CH$ALLOCATION(EMP_NAME_SIZE -
                                %CHARCOUNT(NAME)) OF (0) %;

        !+
        !       Employee Information Table
        !
        !       Size: NO_EMPLOYEES * EMP_LINE_SIZE
        !-

        OWN
                EMP_TABLE:
                        BLOCKVECTOR[NO_EMPLOYEES, EMP_LINE_SIZE]
                        INITIAL(
                                345,
                                201,
                                NAME_PAD('MARKS PETER'),

                                207,
                                345,
                                NAME_PAD('NASSI ISAAC')

                                );
                        ...
                        ...
                        ...
```

The literals serve to parameterize certain values that are subject to change. The literal EMP_LINE_SIZE has as its value the number of words needed for a table entry. The character sequence function, CH$ALLOCATION, returns the number of words needed for EMP_NAME_SIZE characters.

The macro will, based on the length of the employee name argument (NAME), generate zero-filled words to pad out the name field. Thus, we are assured of the same number of words being initialized for each employee name, no matter what its size might be. This is important because storage is allocated according to the fixed length of a character field (employee name). The actual string length may, of course, be less than that value.

This last example was developed with the specification that the employee name field was fixed in length (EMP_NAME_SIZE). What if, however, we wished to have the table hold variable length names? That is, for certain reasons, we wished to allocate only enough storage to hold the table data, not the maximum amount.

The table structure developed above won't work because it is predicated upon the constant size of the name field. If we were to use variable length character strings, either too much or not enough storage would be allocated. And there would be no consistent way of accessing the employee name (where would the next one start?). We could, if we knew the length of every employee name, determine in advance the number of words needed. But this is not a very practical solution.

One transportable solution is to remove the character string from the table and replace it with a pointer (a word in length) to the string. The Character Package has a function, CH$PTR, which will construct a pointer to a character sequence. As an added benefit, this pointer can be used as an argument to the functions in the Character Package. The cost of this technique is the addition of an extra word (the character sequence pointer) for each table entry.

Here is a typical example, again based on the employee table:

```
                    ...
                    ...
                    ...

        !+
        !       Table Parameters
        !-

        LITERAL
                NO_EMPLOYEES = 2,
                EMP_LINE_SIZE = 3;

        !+
        !       Macro to construct a CS-pointer to employee name
        !-

        MACRO
                NAME_PTR(NAME) =
                    CH$PTR(UPLIT( NAME )) %;

        !+
        !       Employee Information Table
        !
        !       Size: NO_EMPLOYEES by EMP_LINE_SIZE
        !-

        OWN
                EMP_TABLE:
                        BLOCKVECTOR[NO_EMPLOYEES, EMP_LINE_SIZE]
                        INITIAL(
                                345,
                                201,
                                NAME_PTR('MARKS PETER'),

                                207,
                                345,
                                NAME_PTR('NASSI ISAAC')

                                );
                    ...
                    ...
                    ...
```

14.4.4.6 Initializing Packed Data -

In this section we will discuss some transportability considerations involved in the initialization of packed data. By packed data, we mean that for data values v1, v2, ..., vn with bit-positions p1, p2, ..., pn and bit-sizes of s1, s2, ..., sn, respectively, the value of

the PLIT-item would be represented by the following expression:

```
v1^p1 OR v2^p2 OR ... OR vn^pn
```

where

```
max(p1, p2, ..., pn) < %BPVAL

s1 + s2 + ... + sn < %BPVAL
```

and for all i

```
-2**si < vi < 2**(si - 1)
```

The OR operator could be replaced by the addition operator (+), but the result would be different if, by accident, there were overlapping values. Notice that the packing of data in a transportable manner is dependent on the value of %BPVAL.

We will illustrate the initialization of packed data by modifying the employee table example that was developed above. When accessing a field within a block, it is a common practice to make each field reference (i.e., offset, position and size) into a macro. So, for example, the field reference macros for the original employee table would look like:

```
MACRO
        EMP_ID          = 0,0,%BPVAL,0 %,
        EMP_COST_CEN    = 1,0,%BPVAL,0 %,
        EMP_NAME_PTR    = 2,0,%BPVAL,0 %;
```

We can make use of these macros in developing an initialization macro. In essence, we are making use of some already parameterized values. This is another example of how we can use parameterization as one of the key techniques in writing transportable code.

If we knew that the number of bits needed to represent the values of EMP_ID and EMP_COST_CEN would each not exceed 16, we could pack these two fields into one BLISS value in BLISS-32 and BLISS-36. In BLISS-16 the definition of the employee table, as it now stands, would allocate only 16 bits for each field, since %BPVAL equals 16. In BLISS-36, we will choose to use an 18-bit size for these two fields, since we know that both DECsystem-10 and DECsystem-20 hardware have instructions that operate efficently on half-words.

Thus, for BLISS-36 and BLISS-32 the field reference macros would look like:

```
MACRO
        EMP_ID          = 0,0,%BPVAL/2,0 %,
        EMP_COST_CEN    = 0,%BPVAL/2,%BPVAL/2,0 %,
```

```
        EMP_NAME_PTR    = 1,0,%BPVAL,0 %;
```

Based on these macros, we can now write a macro that will take as arguments the initial values and then do the proper packing:

```
MACRO
        SHIFT(W,P,S,E) = P %,

        EMP_INITIAL(ID,CC,NAME)[] =
            ID^SHIFT(EMP_ID) OR  ! First
            CC^SHIFT(EMP_COST_CEN) ,  !

            NAME_PTR ( NAME^SHIFT(EMP_NAME_PTR)) %;

                                        ! Second
```

The macro SHIFT simply extracts the position parameter of the field reference macro. The initialization macro, EMP_INITIAL, makes use of this shift value in packing the words. The goal here is to require the user to specify as arguments only the information needed to initialize the table, and not to specify information that is part of its representation.

An example of using these macros to initialize packed data follows:

```
        !+
        !  Employee Field Reference macros
        !-

        MACRO
                EMP_ID  = 0,0,%BPVAL/2,0 %,
                EMP_COST_CEN    = 0,%BPVAL/2,%BPVAL/2,0 %,
                EMP_NAME_PTR    = 1,0,%BPVAL,0 %;

        MACRO

        !+
        !  Macro to create the shift value from the
        !  position parameter of a field reference macro
        !-

                SHIFT(W,P,S,E) = P %,

        !+
        !  Employee table initializing macro
        !  Three values are required
        !-

                EMP_INITIAL(ID,CC,NAME)[] =

                        ID^SHIFT(EMP_ID) OR
                        CC^SHIFT(EMP_COST_CEN) ,    ! First

                        NAME^SHIFT(EMP_NAME_PTR) %;  ! Second

        !+
        !  Employee table definition and initialization
        !-

        OWN
                EMP_TABLE:
                        BLOCKVECTOR[NO_EMPLOYEES, EMP_LINE_SIZE]
                        INITIAL( EMP_INITIAL(
                                345,
                                201,
                                'MARKS PETER',

                                207,
                                345,
                                'NASSI ISAAC'

                                ));
```

What has been illustrated in the previous example is the parameterization of certain values such as field sizes. In transporting this program we can benefit from the localization of certain machine values as in the field reference macros. This code is

transportable between BLISS-32 and BLISS-36. To compile this program with the BLISS-16 compiler, we need to change the field reference macros. The packing macros would no longer be needed, though they could be used for consistency purposes. In that case, they would also need to be changed.

As a final example of initializing packed data, we will use another BLOCK structure that is defined in section 12.7.3 of the BLISS-32 Language Guide. Details as to what DCB is and how it accesses data are discussed in the Language Guide. Here, we will only be concerned with initializing this type of structure.

The DCB BLOCK consists of five fields. Four of the fields are packed into one word, their total combined size being 32 bits, and the fifth field which is 32 bits in length occupies another word.

In this case it is possible to transport the DCB initialization very easily between BLISS-32 and BLISS-36. The reason is that the total number of bits required for each word does not exceed the value of %BPVAL for each machine. Hence, in this case at least, we do not have to modify the design of the BLOCK in any way. Typically, however, one would design the structure for each target machine. This is most easily accomplished by placing its definition in a REQUIRE file. We will again make use of the field reference macros as we did in the previous example.

Here is the example showing a way in which it could be initialized. We have extended the structure by making it a BLOCKVECTOR. The example:

```
        !+
        !  DCB size parameters
        !-

        LITERAL
                DCB_NO_BLOCKS = total number of blocks,
                DCB_SIZE = size of a block;

        !+
        !  DCB Field Reference macros
        !-

        MACRO
                DCB_A = 0,0,8,0 %,
                DCB_B = 0,8,3,0 %,
                DCB_C = 0,11,5,0 %,
                DCB_D = 0,16,16,0 %,
                DCB_E = 1,0,32,0 %;
        MACRO

        !+
        ! Macro to create the shift value from the
        ! position parameter of a field reference macro
        !-

                SHIFT(O,P,S,E) = P %,

        !+
        ! DCB initializing macro.
        !  Five values are required.
        !-

                DCB_INITIALIZE(A,B,C,D,E)[] =
                        A^SHIFT(DCB_A) OR
                        B^SHIFT(DCB_B) OR
                        C^SHIFT(DCB_C) OR
                        D^SHIFT(DCB_D) ,

                        E^SHIFT(DCB_E) %;

        !+
        !  DCB Blockvector definition and initialization
        !-

        OWN
                DCB_AREA:
                    BLOCKVECTOR[DCB_NO_BLOCKS, DCB_SIZE]
                    INITIAL(
                            DCB_INITIALIZE (
                            1,2,3,4,
                            5,
```

```
6,7,8,9,
10,
...
```

Note that this structure could be transported to BLISS-16 by making suitable changes to the field reference macros and the packing macro. The only consideration might be whether the last field, DCB_E, did require a full 32 bits.

14.4.5 Structures And Field Selectors

14.4.5.1 Introduction - Two BLISS constructs will be discussed in this section: structures and field selectors. While the use of one does not necessarily imply the use of the other, we will see that for transportability reasons field selector usage will be confined to structure declarations. Hence, these two constructs need to be discussed together.

We will begin with a general discussion of structures, in which it will be shown that a certain machine specific feature of structures can be used in a transportable manner. The best way to illustrate the process of writing transportable structures is to take the reader through the intellectual considerations that contribute to its design, so the development of a transportable structure - FLEX_VECTOR - will be presented. At this point field selectors will be discussed. Finally, a more general structure - GEN_VECTOR - will be developed.

14.4.5.2 Structures -

Structure declarations are sensitive to transportability in that one may specify parameters corresponding to characteristics of particular architectures. Also, in BLISS-32, the reserved words BYTE, WORD, LONG, SIGNED, and UNSIGNED have values of 1, 2, 4, 1 and 0 respectively when used as structure actual parameters.

We can take advantage of the ability to specify architecture-dependent information in developing transportable structure declarations. Later in this section we will develop a structure which will use the UNIT parameter to gain a degree of transportability. The UNIT parameter specifies the number of addressable allocation-units. This number will be used in determining the amount of storage that is to be allocated for each element of the structure.

As mentioned repeatedly in these guidelines, the prime transportability problem is differing machine architectures. Machine word-sizes, for example aren't the same. That is, the number of bits per machine-word differs on all three machines. The machine word is also the maximum size of a BLISS value. There are two other important architectural differences: bits per address and bits per addressable unit.

Bits per address is the maximum size, in bits, of a memory address. Bits per addressable-unit is the size, in bits, of the smallest directly addressable unit in memory.

The values of machine word-size (BLISS value), bits per addressable-unit and bits per address for the three machines have been implemented as predeclared literals, with the names %BPVAL, %BPUNIT and %BPADDR, respectively. A table of their values can be seen in the section entitled: "Literals".

14.4.5.3 FLEX_VECTOR -

We can make use of these values in developing FLEX_VECTOR. First let's state the use to which this structure will be put: We wish to define a structure that will by default allocate and access a vector consisting of only the smallest addressable units. If the default value given in the structure declaration is not used, we want to be able to specify the vector element size in terms of the number of bits. It should be noted that the existing VECTOR mechanism will not do this.

For example, we would like to have a vector of 9-bit elements. The first decision that has to be made is whether or not we want each element to be exactly 9 bits, or at least 9 bits. For this example, we choose the smallest natural unit whose size is greater than or equal to 9 bits. Since there are no 9-bit (in length) addressable units on any of the machines, we have a choice of 8, 16, 32 or 36-bit units.

We can see that 9 bits will fit in the only addressable unit on the 10/20 systems - the word. On the 11 we will need two bytes or a 16-bit word and on the VAX-11 machine we will again need two bytes.

How then do we develop a structure that will do this allocation and will also be transportable and usable on the three systems? Clearly the structure will need some knowledge of the machine architecture. This is where the role of parameterization comes in.

The predeclared literals have all the information we need. In fact we need only one set of values - bits per addressable-unit(%BPUNIT).

This parameter will be one of the allocation formals. Other formals that we will need are the number of elements (N) and the index parmeter (I) for accessing the vector.

We begin by showing the access and allocation formal list for FLEX_VECTOR:

```
STRUCTURE
        FLEX_VECTOR[ I; N, UNIT = %BPUNIT, EXT = 1 ] =
```

Notice that by setting UNIT equal to %BPUNIT the default (if UNIT is not specified) will be %BPUNIT.

Now we must develop the formula for the structure-size expression. The expression will make use of the allocation formals UNIT and N; and, in addition, the value of the parameter %BPUNIT.

If UNIT were only allowed to assume values of integer multiples of %BPUNIT (i.e. 1*%BPUNIT, 2*%BPUNIT, etc.), we would only need a structure-size expression of the following form:

```
[ N * (UNIT) / %BPUNIT ]
```

Dividing the element size (UNIT) by %BPUNIT would give the size of each element in the vector in terms of an integer multiple. This value would then be multiplied by the number of elements to give the total size of the data to be allocated.

We wish, however, for the structure to be more flexible in that we will be able to specify any size element (within certain limits). The structure-size must be slightly more complex:

```
[ N * (UNIT + %BPUNIT - 1) / %BPUNIT ]
```

The structure-size expression now computes enough %BPUNIT's to hold the entire vector. The reader should try some values of UNIT for differing %BPUNIT in order to see how this expression evaluates.

This sub-expression:

```
(UNIT + %BPUNIT - 1) / %BPUNIT
```

which we will call NO_OF_UNITS is very important in effecting the transportability and flexibility of this particular structure. The key to transporting this structure is the knowledge that it has of a certain machine architectural parameter: bits per addressable-unit. This particular expression makes use of this knowledge, hence, it can adapt to any machine. This sub-expression will be used twice more in the structure-body expression.

The structure-body is an address-expression. This expression will consist of the name of the structure (the base address) plus an offset based on the index I. In addition, a field selector will be needed to access the proper number of bits at the calculated address.

The offset is simply the expression NO_OF_UNITS multiplied by the index I. (Remember that indices start at 0). The size parameter of the field selector is the expression NO_OF_UNITS multiplied by the

size of an addressable-unit - %BPUNIT. The structure-body will look like:

```
(FLEX_VECTOR +
    I * ((UNIT + %BPUNIT - 1) / %BPUNIT))

<0,((UNIT + %BPUNIT - 1)/%BPUNIT)*%BPUNIT,EXT>;
```

The value of the position parameter in the field-selector is a constant 0 for we are always starting at an addressable boundary.

The following table shows the structure on the three machines for different values of UNIT:

VAX-11	
UNIT = 0	no storage FLEX_VECTOR<0,0,1>
UNIT = 1 to 8	[N * 1] Bytes (FLEX_VECTOR + I)<0,8,1>
UNIT = 9 to 16	[N * 2] Bytes (FLEX_VECTOR + I * 2)<0,16,1>
UNIT = 17 to 32	[N * 4] Bytes (FLEX_VECTOR + I * 4)<0,32,1>
11	
UNIT = 0 to 16	same as VAX-11
10/20	
UNIT = 0	no storage (FLEX_VECTOR)<0,0,1>
UNIT = 1 to 36	[N] Words (FLEX_VECTOR + I)<0,36,1>

From the table above we can see that if the default value for UNIT were set to %BPVAL, this structure would be equivalent to a VECTOR of longwords on VAX-11, and a VECTOR of words on the 10/20 and 11 systems.

Elements in a data segment which has this particular structure attribute are accessed very efficiently because they are always on addressable boundaries. Also, they are always some multiple of an addressable unit in length.

If we wish this structure to access elements exactly the size specified then we need only change the size parameter of the field selector. This expression then becomes:

```
... FLEX_VECTOR<0, UNIT>;
```

This is a less efficient means of accessing data (when UNIT is not a multiple of %BPUNIT) because the compiler needs to generate field selecting instructions in the case of the VAX-11 and 10/20 machines and a series of masks and shifts for the 11.

14.4.5.4 Field Selectors -

In the last structure declaration, it was necessary to make use of a field selector. At this, we will discuss the use of field selectors in a more general context.

The use of field selectors can be non-transportable because they make use of the value of the machine word size. The unrestricted usage of field selectors may cause problems in a program when it is moved to another machine. These problems are best illustrated by the following table of restrictions on position (p) and size (s) for the three machines:

Machine:	10/20	11	VAX-11
	$0 \leq p$ $p + s \leq 36$ $0 \leq s \leq 36$	$0 \leq p$ $p + s \leq 16$ $0 \leq s \leq 16$ p, s constant	$0 \leq s \leq 32$

From the table we can see that:

- The most restrictive is the 11.
- The moderate restrictions are those of the 10/20.
- The least restrictive is VAX-11.

If we wished to ensure the transportable use of field selectors, we would have to abide by the set of restrictions imposed in BLISS-16. These, however, are restrictions imposed by the values of p and s. There is also a contextual restriction on the use of field selectors. The following guideline should be followed:

- Field selectors may only appear in the definition of user-defined structures.

By restricting the domain of field selectors to structures, we are in fact isolating their use.

We will now develop another transportable structure which will be affected by the table of field selector value restrictions.

14.4.5.5 GEN_VECTOR -

You have probably noticed that FLEX_VECTOR does not attempt to pack data. Using the example of 9-bit elements, we can see that there will be some wasting of bits - from 7 bits on the 11 and VAX-11 to 27 on the 10/20 systems.

We can develop a variation of FLEX_VECTOR which will provide a certain degree of packing. For example, in the case of 9-bit elements it would be possible to pack at least four of them into a 10/20 word and three into a VAX-11 longword. Unfortunately, this vector is not maximally transportable, but its design and the identification of its non-transportable aspects should be very helpful.

This structure, which will be named GEN_VECTOR, will pack as many elements as possible into a BLISS value (word) so we will make use of the machine specific literal %BPVAL. But, since allocation is in terms of %BPUNIT, we will need a literal that has as a value the number of allocation units in a BLISS value. This literal has been predeclared for transportability reasons and has the name %UPVAL, and is defined as %BPVAL/%BPUNIT.

Elements will not cross word boundaries. This constraint is in effect because of the restrictions placed on the value of the position parameter of a 10/20 and 11 field selector. For the same reason elements can not be longer than %BPVAL, as given in the table of field selector restrictions above.

As in FLEX_VECTOR, the allocation expression of GEN_VECTOR will need to calculate the number of allocation units needed by the entire vector. This will again be based on the number of elements (N) and the size of each element (S). But because the elements will be packed, the expression will be slightly more complicated.

The first value we need is the number of elements that will fit in a BLISS value. The expression:

```
(%BPVAL/S)
```

will compute this value. Given this, to obtain the number of BLISS values or words needed for the entire vector, we divide this value into N:

```
(N/(%BPVAL/S))
```

We now have the total number of words (in units of %BPVAL) needed. However, data is not allocated by words on both of the machines. Multiplying this value by %UPVAL will result in the number of allocation units needed by the vector:

```
((N/(%BPVAL/S))*%UPVAL)
```

For clarity's sake and because this expression will be used again we will make it into a macro with N and S as parameters:

```
MACRO
        WHOLE_VAL(N,S) =

            ((N/(%BPVAL/S))*%UPVAL)%;
```

The name of the macro suggests that we have calculated the number of whole words needed. If, in fact, N were an integral multiple of the number of elements in a word then this macro would be sufficient for allocation purposes.

Since we can't count on this always happening, we need another expression to calculate the number of allocation units needed for any remaining elements. The number of elements left over is the remainder of the last division in this expression:

```
(N/(%BPVAL/S))
```

The MOD function will calculate this value, as follows:

```
(N MOD (%BPVAL/S))
```

If we then multiply this value by the size of each element we will have the total number of bits that remain to be allocated:

```
(N MOD (%BPVAL/S)) * S
```

This value will always be strictly less than %BPVAL. For the same reasons outlined above we will make this expression into a macro with N and S as parameters:

```
MACRO
        PART_VAL(N,S) =

            ((N MOD (%BPAVAL/S)) * S)%;
```

Taking this value, adding a "fudge factor" and then dividing by %BPUNIT will give us the number of allocation units needed for the remaining bits:

```
(PART_VAL(N,S) + %BPUNIT -1)/%BPUNIT
```

The total number of allocation units has been calculated and the structure allocation expression will look like:

```
[WHOLE_VAL(N,S) +
 (PART_VAL(N,S) + %BPUNIT - 1)/%BPUNIT]
```

As it works out, the structure-body expression for GEN_VECTOR will be simple to write because of the expressions that have already been written.

The accessing of an element in GEN_VECTOR requires that we compute an address offset which is then added to the name of the structure. This offset is some number of addressable units based on the value of the index I. We already have an expression which will calculate this number of addressable units. It is the macro WHOLE_VAL. Thus, the first part of the accessing expression will look like:

```
GEN_VECTOR + WHOLE_VAL(I,S)
```

Note that the macro was called with the index parameter I.

This expression will result in the structure being aligned on some addressable boundary. But since the element may not begin at this point (that is, the element may be located somewhere within a unit %BPVAL bits in length), one more value is needed. That value is the position parmeter of a field selector. The macro PART_VAL will calculate this value based on the index I:

```
<PART_VAL(I,S),S,EXT>
```

The size parameter is the value S. The position parameter will be calculated at run-time, based on the value of the index I. Since I is not constant, we can no longer use this structure in BLISS-16. The position and size parameters of a field selector in BLISS-16 must be

compile-time constants. See the table of field selector restrictions above.

This completes the definition of GEN_VECTOR. The entire declaration will look like:

```
STRUCTURE
        GEN_VECTOR[I;N,S,EXT=1] =

            [WHOLE_VAL(N,S) +
             (PART_VAL(N,S) + %BPUNIT - 1)/%BPUNIT]

            (GEN_VECTOR + WHOLE_VAL(I,S))

            <PART_VAL(I,S),S,EXT>;
```

The reader should compile this structure and see how it works in BLISS-32 and BLISS-36.

14.4.5.6 Summary -

No claim is made that either of these two structures will solve all the problems associated with transporting vectors. Many such problems will have unique and different solutions. BLOCKS or BLOCKVECTORS have not been discussed, but it is hoped that the reader will get from the examples a feeling for the techniques involved in transporting structures.

There is no easy solution to transporting data structures. One should consider, when developing data structures, the machines that the program or system is targeted for and make full use of the predeclared literals such as %BPUNIT.

This exercise in the development of transportable structures has illustrated two points:

- parameterization and
- field selector usage.

By parameterizing certain machine-specific values and by taking full advantage of the powerful STRUCTURE mechanism, we have developed two transportable structures.

The accessing of odd (not addressable) units of data is accomplished by the use of field selectors. The field selector should only be used in structure declarations.

[end chapter 14]

Title: VAX-11 Software Eng. Diagnostic Conventions -- Rev 3

Specification Status: draft

Architectural Status: under ECO control

File: SE15R3.RNO

PDM #: not used

Date: 28-Feb-77

Superseded Specs: Diagnostic Coding Conventions

Author: F. Bernaby

Typist: P. Conklin

Reviewer(s): E. Kenney

Abstract: Chapter 15 contains the diagnostic conventional extensions to the rest of this document. It represents an effort to produce diagnostic products in a consistent manner.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Original	F. Bernaby	Sep-76
Rev 2	skipped to maintain numbers		
Rev 3	Integrated with SE manual	P. Conklin	28-Feb-77

Rev 1 to Rev 3:

1. Just merge file.

2. Update module preface.

[End of SE15R3.RNO]

CHAPTER 15

DIAGNOSTIC CONVENTIONS

28-Feb-77 -- Rev 3

15.1 INTRODUCTION

VAX-11 diagnostics will be written in conformance with the conventions expressed in this manual.

These conventions will be adopted to:

1. achieve clear and meaningful documentation of individual tests.
2. reduce the need for diagnostic users to analyze test code.
3. simplify the program maintenance task.

15.2 DIAGNOSTIC SECTIONS

Each diagnostic will be sub-divided into 15 sections. These sections provide a logical way of partitioning the program.

PROGRAM HEADER	provides the module preface for program
PROGRAM EQUATES	area for macro & symbol definitions
PROGRAM DATA	area for data used by more than one test
PROGRAM TEXT	area for all ASCII messages
PROGRAM ERROR REPORT	area reserved for print module
HARDWARE PTABLE	table of hardware parameters
SOFTWARE PTABLE	table of software parameters
DISPATCH TABLE	table of test addresses for test sequencing
REPORT CODE	print module for statistical reports
INITIALIZE CODE	routine for initializing unit under test(unit)
CLEANUP CODE	routine for cleaning up error states in u
PROGRAM SUBROUTINES	area for routines used by more than 1 test
HARDWARE TEST	actual diagnostic test code
HARDWARE PARAMETERS	code used by supervisor to get hardware ptable entries
SOFTWARE PARAMETERS	code used by supervisor to get software ptable entries

15.2.1 Program Header Section

<EXAMPLE>

```
        .TITLE  SYSEXR - System exerciser
        .IDENT  /2-3/

;
; COPYRIGHT (C) 1977
; DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASSACHUSETTS 01754
;
; THIS SOFTWARE IS FURNISHED  UNDER A LICENSE FOR USE ONLY ON A SINGLE
; COMPUTER  SYSTEM AND  MAY BE  COPIED ONLY WITH  THE INCLUSION OF THE
; ABOVE COPYRIGHT NOTICE.  THIS SOFTWARE, OR ANY OTHER COPIES THEREOF,
; MAY NOT BE PROVIDED OR  OTHERWISE MADE AVAILABLE TO ANY OTHER PERSON
; EXCEPT FOR USE ON SUCH SYSTEM AND TO ONE WHO AGREES TO THESE LICENSE
; TERMS.  TITLE TO AND  OWNERSHIP OF THE  SOFTWARE  SHALL AT ALL TIMES
; REMAIN IN DEC.
;
; THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT NOTICE
; AND SHOULD  NOT BE CONSTRUED  AS A COMMITMENT  BY DIGITAL  EQUIPMENT
; CORPORATION.
;
; DEC ASSUMES  NO  RESPONSIBILITY  FOR  THE USE OR  RELIABILITY OF ITS
; SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DEC.
;

;++
; FACILITY: diagnostic exerciser
;
; ABSTRACT:
;
;       This program will exercise the VAX-11 system. It generically
;       treats devices as magtape, disk, or terminals.
;       Up to 32 units may be selected for testing.
;
; ENVIRONMENT:  System
;
; AUTHOR: Frank Bernaby, CREATION DATE: 16-Sep-76
;
; MODIFIED BY:
;
;       Joe Hacker, 4-Jul-77: VERSION 2
; 02    -  Added I/O tests for 6250 tape drives.
; 03    -  Brought module preface to standard form.
;--
```

15.2.2 Program Equates(declarations)

<EXAMPLE>

```
;++
; LISTING CONTROL
;--

.NLIST  MC,MD,CND                       ;NO LIST MACRO'S & CONDITIONALS
.LIST   ME                              ;LIST MACRO EXPANSION

;++
; MACRO LIBRARY CALLS
;--

.MCALL  QIO$S,QIO$C,DPB$,WTEF$C

;++
; INCLUDE FILES: SYSMAC.SML
;--

;++
; EXTERNAL SYMBOLS: DEBUG
;--

.GLOBAL DEBUG                           ;ENTRY POINT OF DEBUGGER


;++
;EQUATED SYMBOLS
;--

;++
; UBA REGISTER DEFINITIONS
;--

UBA_BASE_ADDRESS=177000                 ;UBA BASE ADDRESS
UBA_CSR_OFFSET=0                        ;CONTROL/STATUS REGISTER
UBA_FMR_OFFSET=2                        ;FAILED MAP REGISTER
UBA_IRP_OFFSET=4                        ;MAP REGISTER POINTER
UBA_IRC_OFFSET=6                        ;MAP REGISTER CONTENTS
UBA_SV4_OFFSET=10                       ;REQ SEND VECTOR #4
UBA_SV5_OFFSET=12                       ;REQ SEND VECTOR #5
UBA_SV6_OFFSET=14                       ;REQ SEND VECTOR #6
UBA_SV7_OFFSET=16                       ;REQ SEND VECTOR #7
```

15.2.3 Program Data

<EXAMPLE>

```
;++
; TABLE OF UBA ADDRESS
;
; THIS TABLE IS REFERENCED WHEN ONE OF THE UBA REGISTERS
; MUST BE ADDRESSED. THE UBA REFERENCE IS AN INDIRECT
; REFERENCE THROUGH THIS TABLE. EXAMPLE:
;
;       MOVW    @UBACSR,R0              ;READ CSR INTO R0
;
;--

TABLE_UBA_ADDRESSES:

UBACSR: .LONG   UBA_BASE_ADDRESS+UBA_CSR_OFFSET
UBAFMR: .LONG   UBA_BASE_ADDRESS+UBA_FMR_OFFSET
UBAIRP: .LONG   UBA_BASE_ADDRESS+UBA_IRP_OFFSET
UBAIRC: .LONG   UBA_BASE_ADDRESS+UBA_IRC_OFFSET
UBASV4: .LONG   UBA_BASE_ADDRESS+UBA_SV4_OFFSET
UBASV5: .LONG   UBA_BASE_ADDRESS+UBA_SV5_OFFSET
UBASV6: .LONG   UBA_BASE_ADDRESS+UBA_SV6_OFFSET
UBASV7: .LONG   UBA_BASE_ADDRESS+UBA_SV7_OFFSET


;++
; DEVICE STATUS BUFFER
;
; THIS AREA IS RESERVED FOR STORING DEVICE STATUS AT
; THE CONCLUSION OF AN I/O OPERATION. THIS STATUS
; IS PROVIDED VIA THE QIO MECHANISM.
;
;--

DEVICE_STATUS:  .BLKL   64              ;RESERVE 64 LONG WORDS
```

15.2.4 Program Text

<EXAMPLE>

```
;++
; QUESTIONS
;--

QST1_UBA_BASE:          .ASCIZ  %ENTER UBA BASE ADR: %
QST2_UBA_VECTOR:        .ASCIZ  %ENTER UBA VECTOR ADR: %
QST3_UBA_LEVEL:         .ASCIZ  %ENTER UBA BR LEVEL: %
QST4_RECORD_LENGTH:     .ASCIZ  %ENTER RECORD LENGTH: %
QST5_DATA_PATTERN:      .ASCIZ  %ENTER DATA PATTERN: %

                        .
                        .
                        .


;++
; FORMAT STATEMENTS
;--


FMT1_RKCS_DECODE:       .ASCIZ  /%ARKCS: %XW%A : %RW%N/
FMT2_TIMEOUT:           .ASCIZ  /%ATIMEOUT WHILE REFERENCING RK05 REGISTEI
XL%N/
FMT3_MACHINE_CHECK:     .ASCII  /%AMACHINE CHECK ABORT: %XW%N%XW%A ITEMS/
                        .ASCIZ  / ON STACK, PC= %XL%A SP= %XL%N/
FMT4_SEEK_ERROR:        .ASCIZ  /%ASEEK BAD ERROR REGISTER: %XW%N/
FMT5_ABORT:             .ASCIZ  /%N%N%APROGRAM ABORTING OPERATION%N/
FMT6_PROG_SUMMARY:      .ASCII  /%NPROGRAM SUMMARY/
                        .ASCII  /%NWORDS TRANSFERRED: %XL/
                        .ASCII  /%NHARD ERRORS: %XL/
                        .ASCIZ  /%SOFT ERRORS: %XL%N/
                           .
                           .
                           .
```

15.2.5 Program Error Report

<EXAMPLE>

```
;++
; PRINT ENTRY POINTS FOR ERROR MESSAGES
;--

MSG1_TIMEOUT:           PRINT   FMT2_TIMEOUT,<R1>       ;PRINT TIMEOUT
                        PRINT   FMT5_ABORT,             ;PRINT ABORT
                        RSB                             ;EXIT

MSG2_MACHINE_CHK:       PRINT   FMT3_MACHINE_CHECK,<R6,R7,(R8),R9>
                        RSB

MSG3_DEV_STATUS:        PRINT   FMT1_RKCS_DECODE,<R3,#BITRKCS>
                        RSB                             ;EXIT
```

15.2.6 Hardware Ptable

<EXAMPLE>

```
;++
; HARDWARE PARAMETER TABLE FOR PROGRAM
;
; THIS TABLE PROVIDES THE REQUIRED HARDWARE PARAMETERS
; FOR TEST EXECUTION. THE ENTRIES ARE OBTAINED FROM EITHER
; THE USER VIA GPHRD COMMANDS OR FROM THE SYSTEM
; CONFIGURATION TABLE.
;
;--


HARD_UBA:

HARD_UBA_BASE:          .LONG   0               ;BASE ADDRESS OF UBA
HARD_UBA_VECTOR:        .LONG   0               ;UBA VECTOR ADDRESS
HARD_UBA_LEVEL:         .LONG   0               ;UBA BR LEVEL

                        .
                        .
                        .
```

15.2.7 Software Ptable

<EXAMPLE>

```
;++
; SOFTWARE PARAMETER TABLE FOR  PROGRAM
;
; THIS TABLE CONTAINS ALL THE REQUIRED SOFTWARE PARAMETERS.
; THESE PARAMETERS ARE OBTAINED VIA GPSFT COMMANDS.
;
;--

SOFT_UBA:

SOFT_RECORD_LENGTH:     .LONG   0               ;RECORD LENGTH
SOFT_DATA_PATTERN:      .LONG   0               ;REQUIRED DATA PATTERN
SOFT_DATA_PATH:         .LONG   0               ;UBA DATA PATH
SOFT_MAP_BASE:          .LONG   0               ;BASE MAP REG TO USE
SOFT_MAP_LENGTH:        .LONG   0               ;# OF MAP REG TO USE

                        .
                        .
                        .
```

15.2.8 Dispatch Table

<EXAMPLE>

```
;++
; PROGRAM DISPATCH TABLE
;
; THIS TABLE IS BUILT BY A SUPERVISOR MACRO
;
;--


TEST_DISPATCH:

                N                               ;" N " TEST IN TABLE
                T1S0                            ;ADDRESS OF TEST #1
                T2S0                            ;ADDRESS OF TEST #2
                T3S0                            ;ADDRESS OF TEST #3

                .
                .
                .

                TNS0                            ;ADDRESS OF TEST #N
```

15.2.9 Report Code

<EXAMPLE>

```
;++
; STATISTICAL REPORT MODULE
;
; THIS PRINT MODULE PROVIDES REPORTS OF A STATISTICAL NATURE.
; THE FIRST ENTRY IS INVOKED BY THE SUPERVISOR COMMAND 'REPORT'.
; THE REMAINING ENTRIES ARE PROGRAM INVOKED.
;
;--

REP1_PROG_SUMMARY:  PRINT   FMT6_PROG_SUMMARY,<R6,R5,R7>    ;PRINT SUMMARY
Y
                    RSB                                     ;EXIT


REP2_DATA_SUMMARY:  PRINT   FMT7_DATA_SUMMARY,<R3,R4,DATA_TABLE>
                    PRINT   FMT8_DATA_STAT
                    RSB                                     ;EXIT
```

15.2.10 Intialize Code

<EXAMPLE>

```
;++
; FUNCTIONAL DESCRIPTION: INIT
; THIS ROUTINE INITIALIZES THE TEST PROGRAM.
; IT PERFORMS:
; 1. ALLOCATION OF UNIT(S) UNDER TEST
; 2. INITIAL ALLOCATION OF BUFFER SPACE
; 3. INITIAL MAPPING OF MEMORY SPACE
;
; CALLING SEQUENCE: SUPERVISOR INVOKED
;
; INPUT PARAMETERS: PTABLE
;
;--

        BGNINT                                  ;START OF CODE

          .
          .
          .

        ENDINT                                  ;END OF INITIALIZE
```

15.2.11 Cleanup Code

<EXAMPLE>

```
;++
; FUNCTIONAL DESCRIPTION: CLNUP
; THIS ROUTINE PERFORMS THE NECCESSARY CLEANUP BEFORE
; THE TEST PROGRAM EXITS BACK TO SUPERVISOR LEVEL.
; IT PERFORMS:
; 1. DEALLOCATION OF BUFFER SPACE
; 2. RESET OF UNIT UNDER TEST(UUT)
; 3. DEALLOCATION OF UNIT UNDER TEST
;
; CALLING SEQUENCE: JSB CLNUP
;
; INPUT PARAMETERS: PTABLE
;
;--

        BGNCLN                                  ;START OF CLEANUP
          .
          .
          .
        ENDCLN                                  ;END OF CLEANUP
```

15.2.12 Program Subroutines

<EXAMPLE>

```
        .SBTTL  PROGRAM SUBROUTINES


;++
; FUNCTIONAL DESCRIPTION: $RANDOM
; THIS ROUTINE GENERATES A RANDOM NUMBER THAT IS RETURNED
; IN R0. THE SEED FOR THE NUMBER IS PASSED ON THE STACK.
;
; CALLING SEQUENCE: PUSHL    SEED              ;PUT SEED VALUE ON STCK
                    CALLS    #1,$RAND          ;CALL ROUTINE
;
; INPUT PARAMETERS: SEED
; SEED = BASE VALUE THAT GENERATOR STARTS WITH.
;
;--

$RANDOM:
        .WORD   ^M<R1,R2>                      ;SAVE REG MASK
        MOV     4(AP),R1                       ;FETCH SEED FRM STCK

                .
                .
                .


        MOVL    R1,R0                          ;RETURN VALUE IN R0
$RANDOM_EXIT:

        RET                                    ;RETURN TO CALLER
```

```
;++
; FUNCTIONAL DESCRIPTION: UBA_SETUP
;
; THIS ROUTINE HANDLES THE SETUP OF THE UBA
; TO ALLOW UNIBUS DEVICES TO TRANSFER DATA
; BETWEEN SBI MEMORY AND UNIBUS MEMORY OR
; UNIBUS DEVICES
;

; CALLING SEQUENCE: CALLG       #UBA_LIST,$UBA_SETUP
;
; INPUT PARAMETERS:     UBA_LIST
;
; THIS LIST IS A TABLE LIKE:
;
;       UBA_LIST:               5               ;NUMBER OF ARGUMENTS
;       UBA_BUS_ADR:            .LONG   0       ;BUS ADR AT DEVICE
;       UBA_LENGTH:             .LONG   0       ;RECORD LENGTH
;       UBA_MAP_BASE:           .LONG   0       ;STARTING MAP REG
;       UBA_DAT_PATH:           .LONG   0       ;UBA DATA PATH
;       UBA_SBI_PHYSICAL:       .LONG   0       ;STARTING PHYSICAL ADR
;
;--


$UBA_SETUP:

        .WORD   ^M<R1,R2,R3,R4>         ;SAVE R1-R4
        MOVL    4(AP),R1                ;GET ADDR OF ARGUMENT LIST

        .
        .
        .

$UBA_SETUP_EXIT:

        RET                             ;EXIT
```

15.2.13 Hardware Test

The actual harware test will go within this section of the program. All diagnostics that run with the diagnostic supervisor will, when neccessary, make supervisor 'calls' to provide a function rather than code that function into the program.

If a routine is used by more than one test, that routine will be placed in the program subroutine section. Linkage to that routine will be via 'CALLS' or 'CALLG' instructions. If these routines must pass data back to the test, the test will specify where this data will go by supplying the needed argument(s).

This section is sub-divided by tests and subtests. The test subdivision provides for blocking the diagnostic of into major logic areas. While, the subtest provides a way of further subdividing each test into smalller logic areas.

Therefore the basic organization will look like this.

```
BGNTST

        BGNSUB

        <TEST CODE FOR T1S1>

        ENDSUB
        BGNSUB

        <TEST CODE FOR T1S2>

        ENDSUB

ENDTST
BGNTST

        BGNSUB

        <CODE FOR T2S1>

        ENDSUB

ENDTST
```

Each test and subtest must have a specific level of documentation.

Each test must specify a complete test description and any assumptions that are assumed by this test. Assumptions implies what logic is assumed to have been successful tested when this test starts.

Each subtest must have the test description and assumptions. In addition, the subtest must have a complete description of how the subtest works, what errors the subtest will detect, and what the debug procedure is for the subtest failure.

<EXAMPLE>

```
        BGNTST

;++
;
; TEST DESCRIPTION:
;
; THIS TEST CHECKS THE MAP REGISTERS IN THE UBA. IT PERFORMS THIS TEST
; BY CHECKING THAT ALL REGISTERS HOLD ZEROS AND ONES. THEN THE TEST
; WILL FLOAT A ONE THROUGH ALL REGISTERS. FINALLY, THE TEST WILL FLOAT
; A ZERO THROUGH ALL REGISTERS
;
; ASSUMPTIONS:
;
; TEST1-TEST2
; THIS TEST ASSUMES THAT THE DATA PATH FROM THE CPU TO THE UBA
; HAS BEEN CHECKED AND THAT REGISTER ADDRESSING WORKS CORRECTLY.
;
;--

T3S0:
```

```
BGNSUB

;++
;
; TEST DESCRIPTION:
;
; THIS SUBTEST CHECKS THAT UBM000-UBM496 WILL
; HOLD AN ALL ZEROS DATA PATTERN AND AN ALL ONES DATA
; PATTERN.
;
; ASSUMPTIONS:
;
; TEST1-TEST2
;
; TEST STEPS:
;
; 1. INIT MAP REGISTER INDEX TO ZERO(R3)
; 2. CLEAR SELECTED MAP REGISTER-MP(R3)
; 3. IF MP(R3) .EQU 0 THEN CONTINUE ELSE REPORT ERROR
; 4. COMPLEMENT SELECTED REGISTER-MP(R3)
; 5. IF MP(R3) .EQU -1 THEN CONTINUE ELSE REPORT ERROR
; 6. SELECT NEXT REGISTER(UPDATE R3)
; 7. IF R3 .GTR 496 THEN EXIT ELSE GOTO STEP 2
;
; ERRORS:
;
; 1. TIMEOUT- UBA FAILED TO RESPOND
; 2. ZEROS DATA FAILURE
; 3. ONES DATA FAILURE
;
; DEBUG:
;
; ERROR #1-
; THIS ERROR COULD MEAN POWER FAILURE. CHECK SUPPLIES
;
; ERROR #2-
; CHECK BIT(S) THAT FAILED FOR STUCK AT ONE STATE
;
; ERROR #3-
; CHECK BIT(S) THAT FAILED FOR STUCK AT ZERO STATE
;
;--


T3S1:

<TEST CODE>

T3S1X:
ENDSUB
```

15.2.14 Hardware Parameter Code

<EXAMPLE>

```
;++
; THE HARDWARE PARAMETER TABLE IS BUILT FROM THE INSTRUCTIONS
; IN THIS SECTION. THESE INSTRUCTIONS GET EXECUTED IF THE USER
; STARTS THE PROGRAM WITHOUT SPECIFYING A CONFIGURATION TABLE.
; THE SUPERVISOR WILL RECOGNIZE THIS AN DISPATCH TO THIS SECTION.
; THE INPUT TO THESE REQUEST CAN COME FROM EITHER THE USER
; OR A SCRIPT FILE.
;
; INPUT IS ELICITED BY GPHRD. THIS COMMAND HAS THE FOLLOWING
; FORMAT:
; GPHRD (TABLE OFFSET,FORMAT STATEMENT,RADIX,BYTE OFFSET,LOWER LIMIT,
;        UPPER LIMIT)
;
;--


        BGNHRD                                          ;BEGINNING OF HARDWARE CODE

HPM1:   GPRMD   (BASADR,QST1,0,1,177000,177170) ;GET UBA BASE ADR
HPM2:   GPRMD   (VCTADR,QST2,0,1,100,400)       ;GET UBA VECTOR ADR
           .
           .
           .
        ENDHRD                                          ;RETURN TO SUPERVISOR
```

15.2.15 Software Parameter Code

<EXAMPLE>

```
;++
; THE SOFTWARE PARAMETER TABLE IS BUILT FROM THIS CODE IF THE
; DIAGNOSTIC SUPERVISOR IS DIRECTED TO ACCEPT SOFTWARE
; PARAMETERS FROM EITHER A SCRIPT FILE OR THE USER.
; GPSFT COMMANDS ARE USED TO BUILD THE TABLE. THE FOMRAT
; OF THE ARGUMENTS IS THE SAME AS FOR GPHRD(SEE 8.2.14).
;
;--


        BGNSFT                                          ;FETCH SOFTWARE PARAMS

SPM1:   GPRMD   (RCDLEN,QST4,0,1,20,2000)       ;GET RECORD LENGTH
SPM2:   GPRMD   (DATPTN,QST5,0,1,1,17)          ;GET DATA PATTERN
           .
           .
           .
        ENDSFT                                          ;RETURN TO SUPERVISOR
```

15.3 SYMBOL CONVENTIONS

The following symbol conventions should be used for all VAX-11 diagnostics:

TnSm	specifies test n subtest m
FMTn	specifies format statement n
ASCn	specifies ASCII string n
MSGn	specifies error message n
REPn	specifies statistical report n
QSTn	specifies question n
ISRn	specifies interrupt service routine n
SPMn	software parameter n
HPMn	hardware parameter n

The symbol construction should be as follows:

<prefix>_<descriptive name>_<optional modifier>

15.4 MACRO EXPANSION CONVENTIONS

Macros will be expanded or not expanded based on the following rules. If a macro generates inline test code it will be expanded but not it's call. If a macro makes a call to a subroutine the macro call is shown but not it's expansion. Both the call and expansion can be displayed it the program is assembled with a debug switch set.

[End of Chapter 15]

Title: VAX-11 Assembler Software Engineering Sample -- Rev 3

Specification Status: draft

Architectural Status: under ECO control

File: SEAR3.RNO

PDM #: not used

Date: 28-Feb-77

Superseded Specs: none

Author: P. Conklin

Typist: P. Conklin

Reviewer(s): R. Brender, D. Cutler, R. Gourd, T. Hastings, I. Nassi, S. Poulsen, D. Tolman

Abstract: Appendix A contains a copy of a sample module written in assembly language.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Original	M. Spier	14-Apr-76
Rev 2	Revised from Review	P. Marks	21-Jun-76
Rev 3	After 6 months experience	P. Conklin	28-Feb-77

Rev 2 to Rev 3:

1. Added example.

[End of SEAR3.RNO]

APPENDIX A

ASSEMBLER SAMPLE

28-Feb-77 -- Rev 3

The listing on the next page shows a routine from the procedure library. There is no suggestion that this routine actually works, only that it follows the conventions set forth in this document. In fact, its "facility" does not even exist. Note that it consists of two externally callable routines and a number of internal routines.

```
        .TITLE  CHF$SIGNAL - Condition Handling Facility SIGNAL and STOI
        .IDENT  /1-3/

;
; COPYRIGHT (C) 1977
; DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASSACHUSETTS 01754
;
; THIS SOFTWARE IS FURNISHED  UNDER A LICENSE FOR USE ONLY ON A SINGLE
; COMPUTER  SYSTEM AND  MAY BE  COPIED ONLY WITH  THE INCLUSION OF THE
; ABOVE COPYRIGHT NOTICE.  THIS SOFTWARE, OR ANY OTHER COPIES THEREOF,
; MAY NOT BE PROVIDED OR  OTHERWISE MADE AVAILABLE TO ANY OTHER PERSON
; EXCEPT FOR USE ON SUCH SYSTEM AND TO ONE WHO AGREES TO THESE LICENSE
; TERMS.  TITLE TO AND  OWNERSHIP OF THE  SOFTWARE  SHALL AT ALL TIMES
; REMAIN IN DEC.
;
; THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT NOTICE
; AND SHOULD  NOT BE CONSTRUED  AS A COMMITMENT  BY DIGITAL  EQUIPMENT
; CORPORATION.
;
; DEC ASSUMES  NO  RESPONSIBILITY  FOR  THE USE OR  RELIABILITY OF ITS
; SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DEC.
;

;++
; FACILITY:     Condition Handling
;
; ABSTRACT:
;
;       The Condition Handling Facility supports the exception
;       handling mechanisms needed by each of the common languages.
;       It provides the programmer with some control over fixup,
;       reporting, and flow of control on errors.  It provides
;       subsystem and application writers with the ability to
;       override system messages in order to give a more suitable
;       application oriented interface.
;
;       To understand CHF more fully, refer to its functional
;       specification and to the STARLET exception routine (EXCEPTION).
;
;
; ENVIRONMENT:  Any access mode--normally user mode
;
; AUTHOR: Peter F. Conklin, CREATION DATE: 12-Nov-76
;
; MODIFIED BY:
;
;       Peter F. Conklin, 5-Jan-77: VERSION 01
; 01    -  Original, based on CHF Rev 4 spec
; 02    -  (CVC) Updated to Rev 2 coding standards
; 03    -  Correct code in internal handler.
;--
```

```
        .SBTTL  DECLARATIONS
;
; INCLUDE FILES:
;
        $PSLDEF                         ;PSL definitions
        $SSDEF                          ;System Status code definitions

;
; MACROS:
;
;       NONE
;
; EQUATED SYMBOLS:
;

CANT_MSG_CTRL_L=40                      ;length control string for CHF$STOP
CANT_MSG_BUF_L=40                       ;length insert message for CHF$STOP


CHF$_=^X2222@16                         ;***Temp*** CHF facility code

CHF$_CANT_CONT==CHF$_+4                 ;Can't continue from CHF$STOP
CHF$_NO_HANDLER==CHF$_+8                ;No handler found

SRM$L_HANDLER=0                         ;Call frame handler
SRM$W_SAVE_PSW=4                        ;Call frame PSW
SRM$W_SAVE_MASK=6                       ;Call frame save mask
SRM$L_SAVE_AP=8                         ;Call frame save AP
SRM$L_SAVE_FP=12                        ;Call frame backward link
SRM$L_SAVE_PC=16                        ;Call frame save PC

;
; OWN STORAGE:
;
;       NONE
```

```
          .SBTTL  CHF$STOP -  Stop execution via signalling
;++
; FUNCTIONAL DESCRIPTION:
;
;       This procedure is called whenever it is impossible
;       to continue execution and no recovery is possible.
;       It signals the exception. If the handler(s) return
;       with a continue code, a message "Can't continue"
;       is issued and the image is exitted. This procedure
;       is guaranteed to never return.
;
;
; CALLING SEQUENCE:
;
;       CALL CHF$STOP (condition_value.rlc.v, [{parameters.rz.v}])
;
; INPUT PARAMETERS:
;
;       NONE
;
; IMPLICIT INPUTS:
;
;       NONE
;
; OUTPUT PARAMETERS:
;
;       NONE
;
; IMPLICIT OUTPUTS:
;
;       NONE
;
; COMPLETION CODES:
;
;       NONE
;
; SIDE EFFECTS:
;
;       The process is EXITted if a handler specifies continue.
;
;--

          $FORMAL         <-
CONDITION_VALUE-          ;CONDITION_VALUE.rlc.v is the condition
        >                 ;other arguments are parameters
```

```
   .ENTRY  CHF$STOP,^M<R2>          ;Stop
BSBB    SIGNAL                    ;go do the signaling
MOVAL   -CANT_MSG_CTRL_L(SP),SP   ;allocate room for control string
PUSHAB  (SP)                      ;set pointer to it
PUSHL   #CANT_MSG_CTRL_L          ;make into string descriptor
MOVL    SP,R2                     ;save a copy of descriptor
$GETERR_S CONDITION_VALUE(AP),(R2),(R2)
                                  ;get error string
MOVAL   -CANT_MSG_BUF_L(SP),SP    ;allocate room for string
PUSHAB  (SP)                      ;get pointer to it
PUSHL   #CANT_MSG_BUF_L           ;make into string descriptor
MOVL    SP,R0                     ;get pointer to it
PUSHL   R0                        ;set as arg for later
$FAOL_S (R2),(R0),(R0),CONDITION_VALUE(AP)
                                  ;format error string
PUSHL   #CHF$_CANT_CONT           ;set "can't continue" code
CALLS   #2,LIB$OUT_MESSAGE        ;issue message, with the
                                  ; original SIGNAL's message
                                  ; as the insert
BRW     SIG_EXIT                  ;stop with original exception
                                  ; as the code
```

```
          .SBTTL  CHF$SIGNAL -  Signal Exceptional Condition
;++
; FUNCTIONAL DESCRIPTION:
;
;       This procedure is called whenever it is necessary
;       to indicate an exceptional condition and the procedure
;       can not return a status code. If a handler returns
;       with a continue code, CHF$SIGNAL returns with
;       all registers including R0 and R1 preserved.  Thus,
;       CHF$SIGNAL can also be used to plant performance and
;       debugging traps in any code.  If no handler is found,
;       or all resignal, a catch-all handler is CALLed.
;
;
; CALLING SEQUENCE:
;
;       CALL CHF$SIGNAL (condition_value.rlc.v, [{parameters.rz.v}])
;
; INPUT PARAMETERS:
;
;       NONE
;
; IMPLICIT INPUTS:
;
;       NONE
;
; OUTPUT PARAMETERS:
;
;       NONE
;
; IMPLICIT OUTPUTS:
;
;       NONE
;
; COMPLETION CODES:
;
;       NONE
;
; SIDE EFFECTS:
;
;       If a handler unwinds, then control will not return.
;       A handler could also modify R0/R1 and change the
;       flow of control.  If neither is done, then all
;       registers and condition codes are preserved.
;
;--


            .ENTRY  CHF$SIGNAL,0            ;Signal
       BSBB    SIGNAL                  ;go do the signaling
       RET                             ;return to caller
```

```
          .SBTTL  SIGNAL -  Internal Routine to Signal Exceptions
;++
; FUNCTIONAL DESCRIPTION:
;
;       This routine is used by CHF$STOP and CHF$SIGNAL to do
;       the actual exception signaling. It sets up the handler
;       argument list. It then checks both exception vectors for
;       a handler. It then searches backward up the stack, frame
;       by frame looking for a handler. Each handler found is
;       called. If the handler returns failure (resignal), the
;       search continues. If no handler is found or if all handlers
;       resignal a catch-all handler is called. The catch-all
;       issues the standard message for the condition and then
;       returns success if condition-value<0> is set. If a
;       handler returns success (continue) the routine returns
;       to CHF$STOP or CHF$SIGNAL with R0/R1 intact.
;
;       During the stack search, if another signal is found to
;       be still active, the frames up to and including the
;       establisher of the handler are skipped. Refer to the
;       section Multiply Active Signals in the functional
;       specification. An active signal is defined as a routine
;       which is called from the system vector SYS$CALL HANDLR.
;
;       If a memory access violation is found during the stack
;       search, it is assumed that the stack is finished and
;       the routine calls the catch-all handler.
;
;
; CALLING SEQUENCE:
;
;       JSB
;
; INPUT PARAMETERS:
;
;       AP points to the arg list
;
; IMPLICIT INPUTS:
;
;       NONE
;
; OUTPUT PARAMETERS:
;
;       NONE
;
; IMPLICIT OUTPUTS:
;
;       NONE
;
; COMPLETION CODES:
;
;       NONE
```

```
;
; SIDE EFFECTS:
;
;       If a handler unwinds, then control will not return.
;       A handler could also modify R0/R1 and change the flow
;       of control. If neither is done, then all registers
;       are preserved.
;
;--


SIGNAL:
        PUSHR   #^M<R0,R1>              ;save R0/R1 in mechanism vector
        MOVAB   W^SIGNAL_HANDLER,SRM$L_HANDLER(FP) ;establish a handler
                                        ; to catch access violations
        MNEGL   #3,-(SP)                ;initial depth is -3
        PUSHL   FP                      ;vector frame = current
        PUSHL   #4                      ;mechanism has 4 elements
        PUSHAL  (SP)                    ;second arg is mechanism vector
        PUSHAL  (AP)                    ;first arg is signal vector
        PUSHL   #2                      ;two arguments to handler

;
;  At this point the stack is all set for a call to any handler:
;
;       00(SP) = 2
;       04(SP) = signal vector address
;       08(SP) = mechanism vector address
;       12(SP) = mechanism vector length (4)
;       16(SP) = mechanism vector frame (FP)
;       20(SP) = mechanism vector depth (-3)
;       24(SP) = mechanism vector R0
;       28(SP) = mechanism vector R1
;       32(SP) = RSB return to CHF$STOP or CHF$SIGNAL
;       36(SP)++ RET frame to invoker
;

;
; loop here looking for a handler to call
;

10$:    INCL    20(SP)                  ;move to next depth
        BGEQ    20$                     ;branch if searching stack
        MOVPSL  R0                      ;get current PSL
        EXTZV   #PSL$V_CURMOD,#PSL$S_CURMOD,R0,R0
                                        ;get current mode
        MOVQ    @#CTL$AQ_EXCVEC[R0],R0  ;get both exception vectors
        CMPB    #-1,20(SP)              ;see which vector this time
        BEQL    40$                     ;branch if secondary
        MOVL    R0,R1                   ;if primary, move to R1
        BRB     40$                     ; and branch
```

```
;
; here if searching stack
;
20$:    BLBS    22(SP),SIGNAL_CATCH     ;if loop too long, give up
        MOVL    16(SP),R0               ;get last frame examined
        MOVL    SRM$L_SAVE_FP(R0),16(SP) ;get previous frame
        BEQL    SIGNAL_CATCH            ;branch if no more stack

;
; Here with R0 containing a frame whose predecessor might be
; CHF or EXCEPTION calling to a handler. If so, the return
; PC would be SYS$CALL_HANDL+4 because both JSB to that
; vector to call handlers. If so, we have the situation of
; multiply active signals and need to bypass frames until this
; handler's establisher is skipped. The depth parameter is
; not incremented because a handler and its establisher are
; considered part of the same entity.
;
        CMPL    SRM$L_SAVE_PC(R0),#SYS$CALL_HANDL+4
                                        ;see if multiply active
        BNEQU   30$                     ;branch if not
        BSBB    OLD_SP                  ;adjust R0 to what SP
                                        ; contained before the call
        MOVL    12(R0),R0               ;get mechanism vector
        MOVL    4(R0),16(SP)            ;get establisher's frame
                                        ; as last frame
        BRB     20$                     ;search again

30$:    MOVL    @16(SP),R1              ;get handler if any
40$:    TSTL    R1                      ;see if handler
        BEQL    10$                     ;if no handler, loop
        JSB     @#SYS$CALL_HANDL        ;CALL handler via "vector"
        BLBC    R0,10$                  ;if resignal, loop
        MOVAL   -12(FP),SP              ;clean up stack
        POPR    #^M<R0,R1>              ;restore R0/R1
        RSB                             ;return to CHF$STOP or CHF$SIGNAL

;
; Here when no handler is found, or if all handlers resignal.
; This is either done when the stack saved FP is 0, meaning end
; of the stack, or when an access violation occurs, indicating
; that the stack is bad. The catch-all handler is called and
; then a no-handler message is issued.
;
SIGNAL_CATCH:
        MOVAB   B^SIG_CATCH_ALL,R1      ;set address of handler
        JSB     @#SYS$CALL_HANDL        ;CALL handler via "vector"
        PUSHL   #CHF$_NO_HANDLER        ;get "no handler" code
        CALLS   #1,LIB$OUT_MESSAGE      ;output message
        BRB     SIG_EXIT                ;go exit with condition
                                        ; value as result
```

```
        .SBTTL  SIG_CATCH_ALL -  Internal Catch-all Handler
;++
; FUNCTIONAL DESCRIPTION:
;
;       This handler is used in SIGNAL to catch
;       signals when no handler is found or all resignal.
;
; CALLING SEQUENCE:
;
;       handled = SIG_CATCH_ALL (condition.rl.ra, mechanism.rl.ra)
;
; INPUT PARAMETERS:
;
;       NONE
;
; IMPLICIT INPUTS:
;
;       NONE
;
; OUTPUT PARAMETERS:
;
;       NONE
;
; IMPLICIT OUTPUTS:
;
;       NONE
;
; COMPLETION CODES:
;
;       NONE
;
; SIDE EFFECTS:
;
;       If condition_value<0> is clear, $EXIT is done.
;
;--


SIG_CATCH_ALL:
        .WORD   0                       ;No registers
        MOVAL   @4(AP),AP               ;get condition args
        CALLG   (AP),LIB$OUT_MESSAGE    ;issue standard message
        BLBC    CONDITION_VALUE(AP),SIG_EXIT
                                        ;if failure, go exit
        RET                             ; otherwise, return

;
; Here to give up and exit to the system. The condition value
; argument is given as the exit status.
;

SIG_EXIT:
        $EXIT_S CONDITION_VALUE(AP)     ;exit with condition
                                        ; value as the result
```

```
        .SBTTL  OLD_SP -  Internal Routine to Calculate Old SP
;++
; FUNCTIONAL DESCRIPTION:
;
;       This routine is called to calculate what SP was before
;       a particular CALL that resulted in a specific stack
;       frame.
;
;
; CALLING SEQUENCE:
;
;       JSB
;
; INPUT PARAMETERS:
;
;       R0 = address of stack frame in question
;
; IMPLICIT INPUTS:
;
;       NONE
;
; OUTPUT PARAMETERS:
;
;       R0 = value of SP before CALL in question
;
; IMPLICIT OUTPUTS:
;
;       NONE
;
; COMPLETION CODES:
;
;       NONE
;
; SIDE EFFECTS:
;
;       R1 is clobbered
;
;--


OLD_SP:
        EXTZV   #14,#2,SRM$W_SAVE_MASK(R0),-(SP)
                                        ;get stack offset
        EXTZV   #0,#12,SRM$W_SAVE_MASK(R0),R1
                                        ;get register mask
        ADDL2   #20,R0                  ;standard frame
        ADDL2   (SP)+,R0                ;SP correction
10$:    BLBC    R1,20$                  ;if register bit set,
        ADDL2   #4,R0                   ; count the register
20$:    ASHL    #-1,R1,R1               ;discard bit
        BNEQU   10$                     ;loop until all done

        RSB                             ;return
```

```
      .SBTTL  SIGNAL_HANDLER -  Internal Routine to Handle Access Violation
;++
; FUNCTIONAL DESCRIPTION:
;
;       This handler is used in SIGNAL to catch
;       access violations during the stack search.
;       If it gets an access violation exception from
;       this procedure it terminates the search.
;
;
; CALLING SEQUENCE:
;
;       handled = SIGNAL_HANDLER (condition, mechanism)
;
; INPUT PARAMETERS:
;
;       NONE
;
; IMPLICIT INPUTS:
;
;       NONE
;
; OUTPUT PARAMETERS:
;
;       NONE
;
; IMPLICIT OUTPUTS:
;
;       NONE
;
; COMPLETION CODES:
;
;       0 if not handled
;       success if unwound
;
; SIDE EFFECTS:
;
;       The stack is unwound and SIGNAL_CATCH is branched to.
:
;--
```

```
   SIGNAL HANDLER:
        .WORD   0                       ;No registers
        MOVQ    4(AP),R0                ;get both arguments
        TSTL    8(R1)                   ;verify "this" establisher
        BNEQU   10$                     ;branch if not
        CMPL    4(R0),#SS$_ACCVIO       ;see if memory access violation
        BNEQU   10$                     ;branch if not

   ;
; here if access violation in signal procedure
;

           MOVL    CHF$L_SIG_ARGS(R0),R1   ;get number of signal args      ;M03
        MOVAL   SIGNAL_CATCH,-4(R0)[R1] ;change PC of exception         ;M03
        MOVL    $SS$_CONTINUE,R0        ;resume                         ;M03
        RET                             ; execution                     ;M03
  10$:  CLRL    R0                      ;not handled function value
        RET                             ;return to unwind



                    .END
```

[End of Appendix A]

Title: VAX-11 BLISS Software Engineering Sample -- Rev 3

Specification Status: draft

Architectural Status: under ECO control

File: SEBR3.RNO

PDM #: not used

Date: 27-Feb-77

Superseded Specs: none

Author: P. Conklin

Typist: P. Conklin

Reviewer(s): R. Brender, D. Cutler, R. Gourd, T. Hastings, I. Nassi, D. Tolman

Abstract: Appendix B contains a copy of a sample module written in BLISS.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Original	M. Spier	14-Apr-76
Rev 2	Revised from Review	P. Marks	21-Jun-76
Rev 3	After 6 months experience	P. Conklin	27-Feb-77

Rev 2 to Rev 3:

1. Added example.

[End of SEBR3.RNO]

APPENDIX B

BLISS SAMPLE

27-Feb-77 -- Rev 3

The listing on the next page shows a routine from the procedure library. There is no suggestion that this routine actually works, only that it follows the conventions set forth in this document.

```
MODULE LIB$OUT_MESSAGE ( !Library routine to output a system message
                IDENT='1-4'
                ) =
BEGIN

!
! COPYRIGHT (C) 1977
! DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASSACHUSETTS 01754
!
! THIS SOFTWARE IS FURNISHED  UNDER A LICENSE FOR USE ONLY ON A SINGLE
! COMPUTER  SYSTEM AND  MAY BE  COPIED ONLY WITH  THE INCLUSION OF THE
! ABOVE COPYRIGHT NOTICE.  THIS SOFTWARE, OR ANY OTHER COPIES THEREOF,
! MAY NOT BE PROVIDED OR  OTHERWISE MADE AVAILABLE TO ANY OTHER PERSON
! EXCEPT FOR USE ON SUCH SYSTEM AND TO ONE WHO AGREES TO THESE LICENSE
! TERMS.  TITLE TO AND  OWNERSHIP OF THE  SOFTWARE  SHALL AT ALL TIMES
! REMAIN IN DEC.
!
! THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT NOTICE
! AND SHOULD  NOT BE CONSTRUED  AS A COMMITMENT  BY DIGITAL  EQUIPMENT
! CORPORATION.
!
! DEC ASSUMES  NO  RESPONSIBILITY  FOR  THE USE OR  RELIABILITY OF ITS
! SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DEC.
!

!++
! FACILITY:     Procedure Library
!
! ABSTRACT:
!
!       This routine takes a system message (status) code, gets
!       it from the system message file and formats it with FAO.
!       It then outputs the message to OUTPUT.
!
! ENVIRONMENT:  Any access mode--normally user mode
!
! AUTHOR: Peter F. Conklin, CREATION DATE: 16 Dec 76
!
! MODIFIED BY:
!
!       Peter F. Conklin, 29-Dec-76: VERSION 01
! 01    -  Original, using QIO to TT: only.
! 02    -  Update to standard module format
! 03    -  Change to use GETERR_FRST and GETERR_NEXT
!          and to use PUT_SYSOUT
! 04    -  (CVC) Correct sense of multi-line loop.
!--
```

```
!
! TABLE OF CONTENTS:
!

FORWARD ROUTINE
        LIB$OUT_MESSAGE:NOVALUE; !output message

!
! INCLUDE FILES:
!
!       NONE
!
! MACROS:
!
!       NONE
!
! EQUATED SYMBOLS:
!

LITERAL
        MSG_CTRL_L=132,         !length of control string
        MSG_BUF_L=132;          !length of message

!
! OWN STORAGE:
!
!       NONE
!
! EXTERNAL REFERENCES:
!

EXTERNAL ROUTINE
        LIB$GETERR_FRST:NOVALUE, !get start of message
        LIB$GETERR_NEXT,        !get more of message                    !A03
        SYS$FAOL:NOVALUE,       !format message
        LIB$PUT_SYSOUT:NOVALUE; !put message to SYSOUT:                 !A03
```

```
GLOBAL ROUTINE LIB$OUT_MESSAGE (        !Output system message
        MESSAGE_CODE,   !standard completion code
        LIST)           !substitutable params
        :NOVALUE =

!++
! FUNCTIONAL DESCRIPTION:
!
!       This routine takes a system message (status) code, gets
!       each line of the message from the system message file
!       via the library routines GETERR_FRST and GETERR_NEXT,
!       formats it with FAO, and outputs it via the library
!       routine PUT_SYSOUT.
!
! FORMAL PARAMETERS:
!
!       MESSAGE_CODE.rlc.v      <31:16> facility code
!                               <15:3>  message indicator
!                               <2:0>   severity indicator:
!                                       0 = warning
!                                       1 = success
!                                       2 = error
!                                       4 = severe error
!
!       [{LIST.rz.v}]  remaining parameters are used in call to FAO
!
! IMPLICIT INPUTS:
!
!       NONE
!
! IMPLICIT OUTPUTS:
!
!       NONE
!
! COMPLETION CODES:
!
!       NONE
!
! SIDE EFFECTS:
!
!       One or more records are output on device OUTPUT:
!
!--

    BEGIN

    LOCAL
        CONTROL,                !message line control code
        MSG_CTRL:VECTOR[CH$ALLOCATION(MSG_CTRL_L)], !control string
        MSG_BUF:VECTOR[CH$ALLOCATION(MSG_BUF_L)], !text string
        MSG_CTRL_D:VECTOR[2],   !control string descriptor
        MSG_BUF_D:VECTOR[2];    !text string descriptor
```

```
    !
    ! Initialize string descriptors
    !

    MSG_CTRL_D[0] = MSG_CTRL_L;
    MSG_CTRL_D[1] = MSG_CTRL;
    MSG_BUF_D[1] = MSG_BUF;

    !
    ! Get the message control string for the first line
    !

    LIB$GETERR_FRST (.MESSAGE_CODE, MSG_CTRL_D, MSG_CTRL_D, CONTROL);    !A03

    !+
    ! Loop, processing each line and getting the next
    ! The loop ends when GETERR_NEXT returns false
    !-

    DO                                                                   !A03
        BEGIN                                                            !A03

        !+
        ! If the control code is ' ' or 'T', then
        ! format message for output with FAO and then output it.
        ! Note that GETERR returns the control code as uppercase only.
        !-

        IF .CONTROL<0,8> EQLU ' ' OR .CONTROL<0,8> EQLU 'T'              !A03
        THEN
            BEGIN
            MSG_BUF_D[0] = MSG_BUF_L;
            SYS$FAOL (MSG_CTRL_D, MSG_BUF_D, MSG_BUF_D, MESSAGE_CODE);
            LIB$PUT_SYSOUT (MSG_BUF_D);                                  !A03
            END;

        !
        ! Reset control text length in descriptor and get next line
        !

        MSG_CTRL_D[0] = MSG_CTRL_L;                                      !A03
        END                                                              !A03
    WHILE LIB$GETERR_NEXT (MSG_CTRL_D, MSG_CTRL_D, CONTROL);             !M04
                                                                         !A03


    END;                        !End of LIB$OUT_MESSAGE
```

```
END                                     !End of module
ELUDOM
```

[End of Appendix B]

Title: COMMON BLISS Software Engineering Sample -- Rev 3

Specification Status: draft

Architectural Status: under ECO control

File: SECR3.RNO

PDM #: not used

Date: 27-Feb-77

Superseded Specs: none

Author: R. Murray

Typist: R. Murray

Reviewer(s): R. Brender, D. Cutler, R. Gourd, T. Hastings, I. Nassi, D. Tolman

Abstract: Appendix C contains a copy of a sample module written in BLISS.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Original	P. Belmont	14-Apr-76
Rev 2	Revised from Review	P. Marks	21-Jun-76
Rev 3	After 6 months experience	R. Murray I. Nassi	27-Feb-77

Rev 2 to Rev 3:

1. Added example.

[End of SECR3.RNO]

APPENDIX C

COMMON BLISS SAMPLE

27-Feb-77 -- Rev 3

The following is a running BLISS program that illustrates many of the conventions discussed in this manual. It relies on a small number of external routines for console I/O. These are:

```
TTY_GET_CHAR
TTY_PUT_CHAR
TTY_PUT_CRLF
TTY_PUT_INTEGER
TTY_PUT_ASCIZ
TTY_PUT_MSG
```

```
MODULE LIB$CALC (       ! INTEGER ARITHMETIC EXPRESSION EVALUATOR
                IDENT = '03',
                MAIN = MAINLOOP
                ) =
BEGIN

! COPYRIGHT 1976, DIGITAL EQUIPMENT CORP., MAYNARD, MA 01754
!
! THIS SOFTWARE IS FURNISHED TO THE PURCHASER UNDER  A  LICENSE
! FOR  USE  ON A SINGLE COMPUTER SYSTEM AND CAN BE COPIED (WITH
! INCLUSION OF DIGITAL'S COPYRIGHT NOTICE) ONLY FOR USE IN SUCH
! SYSTEM,  EXCEPT  AS  MAY  OTHERWISE BE PROVIDED IN WRITING BY
! DIGITAL.
!
! THE INFORMATION IN THIS DOCUMENT IS SUBJECT TO CHANGE WITHOUT
! NOTICE AND SHOULD NOT BE CONSTRUED AS A COMMITMENT BY DIGITAL
! EQUIPMENT CORPORATION.  DIGITAL ASSUMES NO RESPONSIBILITY FOR
! ANY ERRORS THAT MAY APPEAR IN THIS DOCUMENT.
!
! DIGITAL EQUIPMENT CORPORATION ASSUMES NO  RESPONSIBILITY  FOR
! THE  USE OR RELIABLILTY OF ITS SOFTWARE ON EQUIPMENT WHICH IS
! NOT SUPPLIED BY DIGITAL EQUIPMENT CORPORATION.
!

!++
!
! FACILITY: GENERAL LIBRARY
!
! FUNCTIONAL DESCRIPTION:
!       THIS PROGRAM PARSES AND EVALUATES ARITHMETIC EXPRESSIONS,
!       KEEPS 26 VALUES AROUND, AND GENERALLY ACTS LIKE AN "AID"
!       WITH DECIMAL INTEGERS ONLY.
!
! ENVIRONMENT: USER MODE WITH EXTERNAL ROUTINES
!
!
!
! AUTHOR: P. BELMONT    CREATION DATE: 01-JAN-76
!
! MODIFIED BY:
!       PETER C. MARKS, 10-MAY-76
! 01    -  CONFORMATION TO S. E. MANUAL STANDARDS
!
!       RICHARD M. MURRAY, 21-FEB-77
! 01    -  CONFORM TO REVISED STANDARD
!
!       ISAAC R. NASSI, 30-APR-77
! 01    -  BUG FIXES, TRANSPORTABILITY CHANGES
!
!
!--
```

```
! EXTENDED FUCTIONAL DESCRIPTION:
!
!       SYNTAX:
!
!       LEXICAL LEVEL: ALL CHARACTERS WITH ASCII VALUE LEQ #040
!       ARE IGNORED.  THUS, BLANKS AND TABS AND <CR> AND <NL>
!       ARE IGNORED (AND MANY OTHERS).
!       UPPER AND LOWER CASE ALPHABETIC CHARACTERS ARE IDENTIFIED.
!
!       SINCE WE READ AHEAD ONE CHARACTER, THE USER MUST
!       TYPE SOMETHING AFTER THE LAST CHARACTER TO GET THE JOB DONE.
!       AFTER PROCESSING, THE REMAINDER OF THE INPUT IS ERASED.
!
!       THE UNARY MINUS ( <T1> ) MAY NOT IMMEDIATELY FOLLOW
!       ANY OPERATOR EXCEPT "(".  THUS -1+1;      (-1+1);
!       (-1+(-2));    ARE ALL CORRECT BUT  -1+-2;      IS NOT.
!
!       <FULL> ->       <EXPR> ;
!       <EXPR> ->       <ALPHA>=<EXPR> ! <T5>
!       <T5>   ->       <T5> + <T4> ! <T4>
!       <T4>   ->       <T4> - <T3> ! <T3>
!       <T3>   ->       <T3> / <T2> ! <T2>
!       <T2>   ->       <T2> * <T1> ! <T1>
!       <T1>   ->       - <T0> ! <T0>   (SEE COMMENT ABOVE ON USE
!                                                    UNARY MINUS.)
!       <T0>   ->       ( <EXPR> ) ! <ALPHA> ! <DECIMAL>
!       <ALPHA> ->      A ! B ! C ! ... ! Z
!       <DECIMAL> ->    <DECIMAL><DIGIT> ! <DIGIT>
!       <DIGIT> ->      0 ! 1 ! 2 ! ... ! 7 ! 8 ! 9
!
!       SEMANTICS:
!
!       THERE ARE 26 VARIABLES WITH <ALPHA> NAMES.  THEY
!       ARE INITIALLY ZERO.
!
!       ASSIGNMENT (THE "=" OPERATOR) IS ALLOWED ONLY TO
!       A VARIABLE AND HAS THE EFFECT OF REPLACING THE VALUE
!       OF THE VARIABLE WITH THE EVALUATED VALUE OF THE  <T5>.
!       THE VALUE OF AN ASSIGNMENT OPERATION IS THE VALUE
!       ASSIGNED.  THUS,  A=B=C=1;    ASSIGNS 1 TO ALL THREE
!       VARIABLES.  THE EXAMPLES:  A=<T5>;
!       A=B=C=<T5>;     B=1+(A=B=5+3);          ARE CORRECT
!       BUT     A+1=3;  1+A=3;  ARE NOT.
!
!       "A;"  MAY BE USED TO PRINT THE VALUE OF A.
!
!       THREE STACKS ARE MAINTAINED IN THIS PROGRAM.
!       THE "MAIN STACK" IS  MAIN_STK AND ITS POINTER IS
!       MAIN_STK_POINTER.
!       ALL VALUES AND OPERATORS END UP ON IT IN RIGHT ORDER.
!
!       THE DERAILING STACK FOR OPERATORS IS OPERATOR_STACK.
!       OP_STACK_PTR IS ITS POINTER.
```

```
!       THIS STACK HOLDS LOWER PRECEDENCE OPERATORS AS HIGHER
!       PRECEDENCE OPERATORS ACCUMULATE.  THIS STACK IS EMPTIED
!       WHEN THE ";" IS PROCESSED.
!
!       THE EVALUATION STACK IS EVAL_STK.  ITS POINTER IS
!       EVAL_STK_PTR.
!       IT HOLDS OPERANDS WHILE THE MAIN STACK IS SCANNED FOR
!       OPERATORS. THE RESULTS OF OPERATIONS PERFORMED
!       GO ON THE EVALUATION STACK.  THIS STACK IS MANAGED
!       BY EVAL_POLISH AND ITS FRIENDS.
```

```
!
! TABLE OF CONTENTS:
!
FORWARD ROUTINE
!       MAINLOOP,                       ! MAIN PROGRAM LOGIC
        EXPRESSION,                     ! READ, PARSE, AND EVALUATE
                                        ! AN EXPRESSION
        INPUT_CYCLE,                    ! PARSE AN EXPRESSION
        PROCESS_OPR,                    ! PROCESS AN OPERATOR
        READ_UNTIL_DEL,                 ! LEXEME BUILDING
        GET_CHARACTER:NOVALUE,          ! GET A CHARACTER FROM INPUT
                                        ! STREAM
        PUSH_OPERATOR:NOVALUE,          ! PUSH ONTO OPERATOR STACK
        POP_OPERATOR,                   ! POP OFF OF OPERATOR STACK
        PUSH_MAIN_STACK:NOVALUE,        ! PUSH ONTO MAIN STACK
        POP_MAIN_STACK,                 ! POP OFF OF MAIN STACK
        EVAL_POLISH:NOVALUE,            ! EVALUATE A POLISH-TYPE
                                        ! EXPRESSION
        EVAL_OPERATOR,                  ! EVALUATE AN OPERATOR
        EVAL_ADDRESS,                   ! EVALUATE AN ADDRESS
        EVAL_VALUE                      ! EVALUATE A VALUE
        PUSH_EVAL_STACK:NOVALUE,        ! PUSH ONTO EVALUATION STACK
        POP_EVAL_STACK,                 ! POP OFF OF EVALUATION STACK
        PRINT_STRING:NOVALUE,           ! PRINT AN EXPRESSION
        PRINT_STACK,                    ! PRINT THE CONTENTS OF THE
                                        ! MAIN STACK
        ERROR;                          ! PRINT AN ERROR MESSAGE
!
!
! INCLUDE FILES:
!
REQUIRE
        'BLI:COMIOG.REQ';
!
! MACROS:
!
MACRO
        LEXEME_TYPE=    LEXEME[0]%,
        LEXEME_VALUE=   LEXEME[1]%,
        MAIN_TYPE=      MAIN_STK[.MAIN_STK_PTR-2]%,
        MAIN_VALUE=     MAIN_STK[.MAIN_STK_PTR-1]%,
        TOPOP=          OPERATOR_STACK[.OP_STACK_PTR-1]%;
```

```
!
! EQUATED SYMBOLS:
!
LITERAL
!
!
!                       OPERATOR CODES
!
        FIRST=          0,              ! "FIRST" OPERATOR
        OPEN_PAREN=     1,              ! OPEN PARENTHESIS
        CLOSE_PAREN=    2,              ! CLOSE PARENTHESIS
        MULTIPLY=       3,              ! MULTIPLICATION "*"
        PLUS=           4,              ! ADDITION "+"
        MINUS=          5,              ! SUBTRACTION "-"
        DIVIDE=         6,              ! DIVISION "/"
        SEMI_COL=       7,              ! SEMI-COLON ";"
        EQUAL=          8,              ! ASSIGNMENT "="
        NEGATIVE=       9,              ! NEGATION (UNARY MINUS) "-"
        CUTOFF=         2,              ! OPEN PAREN AND FIRST FOR NEG
!
!                       MAIN STACK ELEMENT CODES
!
        IS_NAME=        1,              ! VARIABLE NAME
        IS_DECIMAL=     2,              ! INTEGER VALUE
        IS_OPERATOR=    3,              ! OPERATOR
        IS_NONE=        0;              ! "NOTHING" (SPECIAL ELEMENT)
!
!                       PRECEDENCE TABLES
!
!        PRCDENCE_1 IS THE PRECEDENCE OF THE CURRENT OPERATOR
!        PRCDENCE_2 IS THE PRECEDENCE OF THE OPERATOR ON TOP OF
!        OP-STACK.
!        THE TWO ARE COMPARED IN INPUT_CYCLE.
!        THE TWO LISTS ARE BASICALLY THE SAME, BUT:
!
!        NOTE: PRECEDENCE_2 IS THE SAME AS PRECEDENCE_1 FOR ALL
!        OPERATORS EXCEPT "(" WHERE IT IS REDUCED TO 1 AND "=" WHERE
!        IT IS REDUCED TO 2. CLOSE PAREN WILL FORCE "=" ONTO
!        STACK AND WILL FORCE ALL OTHER OPERATORS DOWN TO "("
!        WHICH, WITH ")", IS REMOVED BY THE ACTION OF ")".
!
BIND
        PRCDNCE_1= UPLIT(0,9,2,7,4,5,6,1,3,8):VECTOR[10],
!                                  ( ) * + - / ; = -
        PRCDNCE_2= UPLIT(0,1,2,7,4,5,6,1,2,8):VECTOR[10],
!
!                       ASCII VALUE OF OPERATORS
!
        OPNAMES=        PLIT(
                                PLIT (%ASCIZ 'FIRST'),
                                PLIT (%ASCIZ '('),
                                PLIT (%ASCIZ ')'),
                                PLIT (%ASCIZ '*'),
```

```
                                PLIT (%ASCIZ '+'),
                                PLIT (%ASCIZ '-'),
                                PLIT (%ASCIZ '/'),
                                PLIT (%ASCIZ ';'),
                                PLIT (%ASCIZ '='),
                                PLIT (%ASCIZ 'NEG')):VECTOR[50];
!
!                       SIZE PARAMETERS
!
LITERAL
        INPUT_SIZE=     133,                    ! INPUT AREA SIZE
        OUT_MSG_MAX=    132,                    ! MAX OUTLINE LENGTH
        STACK_SIZE=     400,                    ! SIZE OF STACKS
        STOR_LEN=       26,                     ! SIZE OF STORAGE
!
!                       MISC.
!
        NOTHING=        0,                      ! A CONDITION
        CAR_RETURN =    %O'15',                 !CARRIAGE RETURN
        CHARMASK =      (1^5)-1;                ! MASK LOW BITS

!
! OWN STORAGE
!
OWN
!                       STACKS
!
        MAIN_STK:       VECTOR[STACK_SIZE],     ! MAINSTACK
        OPERATOR_STACK: VECTOR[STACK_SIZE],     ! OPERATOR STACK
        EVAL_STK:       VECTOR[STACK_SIZE],     ! EVALUATION STACK
!
!                       STACK POINTERS
!
        MAIN_STK_PTR,
        OP_STACK_PTR,
        EVAL_STK_PTR,
!
!                       PARSING VARIABLES AND AREAS
!
        CHAR,                                   ! SINGLE ASCII
                                                ! CHARACTER INPUT
        STORAGE:        VECTOR[STOR_LEN],       ! IDENTIFIER VALUE
                                                ! STORAGE AREA
        DECVALUE,                               ! DECIMAL VALUE
        LEXEME:         VECTOR[2],              ! LEXICAL ELEMENT
                                                  ! LEXEME[0] = TYPE
                                                  ! LEXEME[1] = VALUE
        OPERATOR,                               ! OPERATOR CODE
                                                ! (SEE ABOVE)
        PAREN_LEVEL,                            ! PARENTHESES LEVEL
        INPUT:          VECTOR[INPUT_SIZE],     ! INPUT LINE
        INPUT_POINTER,                          ! INPUT LINE POINTER
        INPUT_LENGTH,                           ! LENGTH OF INPUT LINE
        ERRORV;
```

```
!
! EXTERNAL REFERANCES:
!
!       NONE
```

```
ROUTINE MAINLOOP  :NOVALUE =

!++
!
! FUCTIONAL DESCRIPTION:
!
!
!       THIS IS THE MAIN ROUTINE OF THIS MODULE.  IT CONTAINS THE
!       GROSS LOGIC OF THE MODULE.
!       THE USER IS REPEATEDLY ASKED TO "TYPE EXPRESSION".  UPON
!       DOING SO THE EXPRESSION IS PARSED AND EVALUATED BY A
!       CALL TO THE ROUTINE EXPRESSION.
!       EXECUTION OF THIS ROUTINE (AND THE MODULE) IS HALTED BY
!       HITTING CONTROL C.
!
! FORMAL PARAMETERS:
!
!       NONE
!
! IMPLICIT INPUTS:
!
!       NONE
!
! IMPLICIT OUTPUTS:
!
!       STORAGE, INPUT, INPUT_LENGTH, INPUT_POINTER
!
! ROUTINE VALUE:
!
!       NONE
!
! SIDE EFFECTS:
!
!       NONE
!
!--

        BEGIN
!
!        RESET STORAGE TO ZERO VALUES
!
        INCR I FROM 0 TO (STOR_LEN - 1) DO
            STORAGE[.I]= 0;
!
! READ NEXT LINE
!
        WHILE 1 DO
            BEGIN
            TTY_PUT_CRLF();
            TTY_PUT_CHAR( %C'*' );                  ! PROMPT
            INCR I FROM 0 TO INPUT_SIZE-1
            DO
                BEGIN
```

```
            INPUT[.I] = TTY_GET_CHAR();
            IF .INPUT[.I] EQL CAR_RETURN    !CARRIAGE RETURN
            THEN
                BEGIN
                INPUT[.I] = %C';';  ! ONE EXTRA SEMICOLON
                INPUT_LENGTH = .I;
                EXITLOOP;
                END;
            END;
        INPUT_POINTER = -1;
        IF EXPRESSION() THEN RETURN
        END;
    END;
```

```
ROUTINE EXPRESSION  =

!++
!
! FUCTIONAL DESCRIPTION:
!
!
!       LOGICALLY,
!
!       THIS ROUTINE REPEATEDLY CALLS THE ROUTINE INPUT_CYCLE
!       IN ORDER TO READ AND PARSE THE EXPRESSION, PRINTS THE
!       EXPRESSION, PRINTS THE CONTENTS OF THE STACK JUST BUILT, AND
!       THEN EVALUATES THE EXPRESSION VIA A CALL TO EVAL_POLISH.
!
! FORMAL PARAMETERS:
!
!       NONE
!
! IMPLICIT INPUTS:
!
!       NONE
!
! IMPLICIT OUTPUTS:
!
!       PAREN_LEVEL
!
! COMPLETION CODES:
!
!       RETURNED AS ROUTINE VALUE;
!               0 - NO ERRORS REPORTED
!               1 - ERROR ENCOUNTERED
!
! SIDE EFFECTS:
!
!       NONE
!
!--

        BEGIN
        LOCAL
                CONDITION;      ! VALUE RETURNED BY INPUT_CYCLE

        PAREN_LEVEL = 0;
        DO
            CONDITION = INPUT_CYCLE()
        UNTIL .CONDITION NEQ 0;
        IF .CONDITION  EQL 1 THEN  RETURN 1;            !ERROR
        PRINT_STRING();
        PRINT_STACK();
        EVAL_POLISH();
        RETURN 0
        END;
```

```
ROUTINE INPUT_CYCLE =

!++
!
! FUCTIONAL DESCRIPTION:
!
!
!       THIS ROUTINE MAKES CALLS TO ROUTINE READ_UNTIL_DELITER
!       ACCESSING LEXEMES AND DELIMITERS.  BASED ON THE TYPE
!       THE ROUTINE PERFORMS VARIOUS FUNCTIONS.
!       NOTE:
!               THERE IS AN INTERNAL ROUTINE CALLED PROCESS_OPR, WHICH
!       HANDLES OPERATOR DELIMITERS.
!
! FORMAL PARAMETERS:
!
!       NONE
!
! IMPLICIT INPUTS:
!
!       LEXEME_TYPE, LEXEME_VALUE
!
! IMPLICIT OUTPUTS:
!
!       NONE
!
! COMPLETION CODES:
!
!       RETURNED AS ROUTINE VALUE;
!               0 - NO ERRORS ENCOUNTERED
!               1 - ERROR ENCOUNTERED
!               2 - END OF EXPRESSION
!
! SIDE EFFECTS:
!
!       NONE
!
!--

        BEGIN
        LOCAL
                VALUE;                  ! VALUE TO BE RETURNED

        IF READ_UNTIL_DEL() THEN RETURN 1;
        IF .LEXEME_TYPE NEQ IS_NONE
        THEN
            BEGIN
            PUSH_MAIN_STACK(.LEXEME_TYPE);
            PUSH_MAIN_STACK(.LEXEME_VALUE)
            END
        ELSE                        !UNARY OPERATOR
            IF (.OPERATOR NEQ MINUS AND
                .OPERATOR NEQ OPEN_PAREN)
```

```
        THEN
            RETURN(ERROR(4))
        ELSE
            IF .OPERATOR EQL MINUS
            THEN
                IF .PRCDNCE_2[.TOPOP] LSS CUTOFF
                THEN
                    OPERATOR = NEGATIVE
                ELSE
                    RETURN(ERROR(5));
    PROCESS_OPR
    END;
```

```
ROUTINE PROCESS_OPR =

!++
!
! FUCTIONAL DESCRIPTION:
!
!       THIS ROUTINE HANDLES OPERATORS (DELIMITERS).  IT
!       KEEPS TRACK OF THE PARENTHESES COUNT AND THE PROPER
!       SYNTAX OF EXPRESSIONS.
!
! FORMAL PARAMETERS:
!
!       NONE
!
! IMPLICIT INPUTS:
!
!       OPERATOR, PAREN_LEVEL, TOPOP, LEXEME_TYPE,
!       PRECIDENCE_1, PRECIDENCE_2
!
! IMPLICIT OUTPUTS:
!
!       PAREN_LEVEL
!
! COMPLETION CODES:
!
!       RETURNED AS ROUTINE VALUE;
!               0 - NO ERRORS ENCOUNTERED
!               1 - ERROR ENCOUNTERED
!               2 - END OF EXPRESSION
!
! SIDE EFFECTS:
!
!       NONE
!
!--

        BEGIN
        LOCAL
                CONDITION;      ! VALUE RETURNED BY PROCESS_OPR

        IF .OPERATOR EQL OPEN_PAREN
        THEN
            PAREN_LEVEL = .PAREN_LEVEL+1;

        IF .OPERATOR EQL CLOSE_PAREN
        THEN
            PAREN_LEVEL = .PAREN_LEVEL-1;

        WHILE .PRCDNCE_1[.OPERATOR] LEQ .PRCDNCE_2[.TOPOP]
        DO
            BEGIN
            PUSH_MAIN_STACK(IS_OPERATOR);
            PUSH_MAIN_STACK(POP_OPERATOR());
```

```
    END;

IF .OPERATOR EQL SEMI_COL
THEN
    IF .PAREN_LEVEL EQL 0
    THEN
        RETURN 2
    ELSE
        RETURN (ERROR(9));

IF .OPERATOR EQL CLOSE_PAREN
THEN
    BEGIN
    IF .TOPOP NEQ OPEN_PAREN THEN  RETURN(ERROR(3));
    POP_OPERATOR();
    IF READ_UNTIL_DEL() THEN  RETURN 1;
    IF .LEXEME_TYPE NEQ IS_NONE THEN RETURN(ERROR(6));
    CONDITION=PROCESS_OPR();
    IF .CONDITION GTR 0
    THEN
        RETURN .CONDITION;
    END
ELSE
    PUSH_OPERATOR(.OPERATOR);
RETURN 0

END;
```

```
ROUTINE READ_UNTIL_DEL  =

!++
!
! FUCTIONAL DESCRIPTION:
!
!       THIS ROUTINE DOES THE ACTUAL PARSING OF THE INPUT EXPRESSION
!       LOOKING FOR SYMBOLS, NUMBERS AND OPERATORS(DELIMITERS).
!       IT ALWAYS ATTEMPTS TO RECOGNIZE AN OPERATOR AND
!       RETURN ITS CODE.
!       PRIOR TO SEARCHING FOR THE OPERATOR IT LOOKS FOR A SYMBOL
!       (IS_NAME) OR INTEGER(IS_DECIMAL).  IF NONE OF THESE
!       ARE FOUND THEN IS_NONE IS RETURNED IN THE GLOBAL VARIABLE
!       LEXEME_TYPE.
!
! FORMAL PARAMETERS:
!
!       NONE
!
! IMPLICIT INPUTS:
!
!        CHAR, OPERATOR
!
! IMPLICIT OUTPUTS:
!
!       OP_STACK_PTR, MAIN_STK_PTR, ERRORV, LEXEME_TYPE,
!       LEXEME_VALUE, DECVALUE, OPERATOR
!
! ROUTINE VALUE:
!
!       OPEN_PAREN, CLOSE_PAREN, MULTIPLY, PLUS, MINUS
!       DIVIDE, SEMI_COL, EQUAL, ERROR(1), ERROR(2)
!
! SIDE EFFECTS:
!
!       NONE
!
!--

        BEGIN
        IF .INPUT_POINTER EQL -1
        THEN
!
!           IF FIRST TIME THROUGH PLACE SPECIAL "FIRST" DELIMITER
!           ON THE MAIN_STK
!
            BEGIN
            GET_CHARACTER();
            OP_STACK_PTR = 0;
            MAIN_STK_PTR = 0;
            ERRORV = 0;
            PUSH_OPERATOR(FIRST);                !INITDEL
            END;
```

```
!       FIRST SEARCH FOR A SYMBOL OR INTEGER

        LEXEME_TYPE = IS_NONE;  !FOR THERE MAY NOT BE ONE
        IF (.CHAR GEQ %C'A' AND .CHAR LEQ %C'Z') OR
           (.CHAR GEQ %C'a' AND .CHAR LEQ %C'z')
        THEN
            BEGIN                                   !CONVERT CHAR TO AN
            LEXEME_TYPE = IS_NAME;                  !INDEX INTO STORAGE
                                                    !ARRAY
            LEXEME_VALUE = (.CHAR AND CHARMASK)-1;
            GET_CHARACTER()
            END
        ELSE
            IF (.CHAR GEQ %C'0' AND .CHAR LEQ %C'9')
            THEN
                BEGIN
                DECVALUE = 0;
                WHILE (.CHAR GEQ %C'0' AND .CHAR LEQ %C'9') DO
                    BEGIN
                    DECVALUE = 10*.DECVALUE+.CHAR-%C'0';
                    GET_CHARACTER();
                    END;
                LEXEME_TYPE = IS_DECIMAL;
                LEXEME_VALUE = .DECVALUE;           !DECIMAL INTEGER VALUE
                END;

!       NOW GET DELIMITER WHETHER OR NOT WE HAD AN IDENTIFIER OR NUMBER

        IF (.CHAR LSS %C'(' OR .CHAR GTR %C'=')
        THEN
                RETURN(ERROR(1))
        ELSE
            BEGIN
            OPERATOR=
                (CASE (.CHAR) FROM %C'(' TO %C'=' OF
                    SET
                    [%C'(']:
                        OPEN_PAREN;
                    [%C')']:
                        CLOSE_PAREN;
                    [%C'*']:
                        MULTIPLY;
                    [%C'+']:
                        PLUS;
                    [%C'-']:
                        MINUS;
                    [%C'/']:
                        DIVIDE;
                    [%C';']:
                        SEMI_COL;
                    [%C'=']:
                        EQUAL;
                    [INRANGE]:
```

```
                    ! ALL OTHER VALUES ARE IN ERROR
                    RETURN(ERROR(2))
                TES);
        GET_CHARACTER();
        IF .OPERATOR EQL 0 THEN RETURN(ERROR(2))
         END;
    END;
```

```
ROUTINE GET_CHARACTER  :NOVALUE =

!++
!
! FUCTIONAL DESCRIPTIION:
!
!       THIS ROUTINE ACCESES THE NEXT CHARACTER FROM THE INPUT STREAM
!       AND PLACES IT IN THE GLOBAL VARIABLE CHAR.
!       ALL CHARACTERS WITH AN OCTAL VALUE LESS THAN 40 ARE IGNORED.
!
! FORMAL PARAMETERS:
!
!       NONE
!
! IMPLICIT INPUTS:
!
!       INPUT_POINTER
!
! IMPLICIT OUTPUTS:
!
!       INPUT_POINTER, CHAR
!
! ROUTINE VALUE:
! COMPLETION CODES:
!
!       NONE
!
! SIDE EFFECTS:
!
!       NONE
!
!--

        BEGIN
        DO
            BEGIN
            INPUT_POINTER = .INPUT_POINTER + 1;
            CHAR = .INPUT[.INPUT_POINTER];
            END
        UNTIL (.CHAR GTR %C' ');
        END;
```

```
ROUTINE PUSH_OPERATOR(ELEMENT) :NOVALUE =

!++
!
! FUCTIONAL DESCRIPTION:
!
!       THIS ROUTINE "PUSHES" AN ELEMENT ONTO THE OPERATOR_STACK.
!
! FORMAL PARAMETERS:
!
!       ELEMENT - OPERATOR TO BE ADDED TO STACK
!
! IMPLICIT INPUTS:
!
!       OP_STACK_PTR
!
! IMPLICIT OUTPUTS:
!
!       OP_STACK_PTR, OPERATOR_STACK
!
! ROUTINE VALUE:
! COMPLETION CODES:
!
!       NONE
!
! SIDE EFFECTS:
!
!       NONE
!
!--

        BEGIN
        OPERATOR_STACK[.OP_STACK_PTR] = .ELEMENT;
        OP_STACK_PTR = .OP_STACK_PTR+1
        END;
```

```
ROUTINE POP_OPERATOR  =

!++
!
! FUCTIONAL DESCRIPTION:
!
!       THIS ROUTINE "POPS" A DATA ELEMENT OFF OF THE OPERATOR_STACK.
!
! FORMAL PARAMETERS:
!
!       NONE
!
! IMPLICIT INPUTS:
!
!       OP_STACK_PTR, OPERATOR_STACK
!
! IMPLICIT OUTPUTS:
!
!       OP_STACK_PTR
!
! ROUTINE VALUE:
!
!       VALUE OF ELEMENT POPPED FROM STACK
!
! SIDE EFFECTS:
!
!       NONE
!
!--

        BEGIN
        OP_STACK_PTR = .OP_STACK_PTR-1;
        .OPERATOR_STACK[.OP_STACK_PTR]
        END;
```

```
ROUTINE PUSH_MAIN_STACK(ELEMENT) :NOVALUE =

!++
!
! FUCTIONAL DESCRIPTION:
!
!       THIS ROUTINE WILL "PUSH" AN ELEMENT ONTO THE MAIN_STK.
!
! FORMAL PARAMETERS:
!
!       ELEMENT - DATA TO BE PUSHED ON MAIN_STK
!
! IMPLICIT INPUTS:
!
!       MAIN_STK_PTR
!
! IMPLICIT OUTPUTS:
!
!       MAIN_STK, MAIN_STK_PTR
!
! ROUTINE VALUE:
! COMPLETION CODES:
!
!       NONE
!
! SIDE EFFECTS:
!
!       NONE
!
!--

        BEGIN
        MAIN_STK[.MAIN_STK_PTR]= .ELEMENT;
        MAIN_STK_PTR= .MAIN_STK_PTR+1
        END;
```

```
ROUTINE POP_MAIN_STACK  =

!++
!
! FUCTIONAL DESCRIPTION:
!
!       THIS ROUTINE "POPS" AN ELEMENT OFF OF THE MAIN_STK
!
! FORMAL PARAMETERS:
!
!       NONE
!
! IMPLICIT INPUTS:
!
!       MAIN_STK_PTR, MAIN_STK
!
! IMPLICIT OUTPUTS:
!
!       MAIN_STK
!
! ROUTINE VALUE:
!
!       VALUE OF ELEMENT POPPED FROM THE STACK
!
! SIDE EFFECTS:
!
!       NONE
!
!--

        BEGIN
        MAIN_STK_PTR= .MAIN_STK_PTR-1;
        .MAIN_STK[.MAIN_STK_PTR]
        END;
```

```
ROUTINE EVAL_POLISH  :NOVALUE =

!++
!
! FUCTIONAL DESCRIPTIION:
!
!       THIS ROUTINE DOES THE ACTUAL EVALUATION OF EXPRESSION
!       WHICH HAS NOW BEEN PARSED AND RESIDES ON THE MAIN_STK.
!       OPERANDS (VARIABLES AND INTEGERS) ARE SHUNTED OFF AND PLACED
!       ONTO THE EVAL_STK.
!       OPERATORS ARE EVALUATED BY MAKING A CALL TO EVAL_OPERATOR.
!
! FORMAL PARAMETERS:
!
!       NONE
!
! IMPLICIT INPUTS:
!
!       MAIN_STK_PTR, MAIN_STK, EVAL_STK
!
! IMPLICIT OUTPUTS:
!
!       EVAL_STK_PTR, LEXEME_TYPE, LEXEME_VALUE, EVAL_STK
!
! ROUTINE VALUE:
! COMPLETION CODES:
!
!       NONE
!
! SIDE EFFECTS:
!
!       NONE
!
!--

        BEGIN
        EVAL_STK_PTR = 0;
        INCR I FROM 0 TO .MAIN_STK_PTR-1 BY 2 DO
            BEGIN
            LEXEME_TYPE = .MAIN_STK[.I];
            LEXEME_VALUE = .MAIN_STK[.I+1];
            IF .LEXEME_TYPE NEQ IS_OPERATOR
            THEN
                BEGIN
                PUSH_EVAL_STACK(.LEXEME_TYPE);
                PUSH_EVAL_STACK(.LEXEME_VALUE)
                END
            ELSE
                EVAL_OPERATOR(.LEXEME_VALUE);
            END;
        IF .MAIN_STK_PTR EQL 2
        THEN
            EVAL_STK[1] = EVAL_VALUE(); ! THE CASE "A;"
```

```
        TTY_PUT_CRLF();
        TTY_PUT_QUO('VAL:       ');
        TTY_PUT_INTEGER(.EVAL_STK[1],10,10);
        END;
```

```
ROUTINE EVAL_OPERATOR(STACK_OPERATOR) =

!++
!
! FUCTIONAL DESCRIPTION:
!
!       THIS ROUTINE EVALUATES THE OPERATOR STACK_OPERATOR.
!       THE PROPER NUMBER OF OPERANDS ARE ACCESSED FORM THE MAIN_STK.
!       AFTER EVALUATION THE VALUE IS PLACED ON THE EVAL_STK.
!
! FORMAL PARAMETERS:
!
!       STACK_OPERATOR - OPERATOR TO BE EVALUATED
!
! IMPLICIT INPUTS:
!
!       NONE
!
! IMPLICIT OUTPUTS:
!
!       STORAGE
!
! ROUTINE VALUE:
!
!       ERROR(3)
!
! SIDE EFFECTS:
!
!       NONE
!
!--

        BEGIN
        LOCAL
            VALUE_1,                                ! INTERMEDIATE
                                                    ! SAVE AREAS
            VALUE_2,                                ! ...
            VALUE_3;                                ! ...

        VALUE_3 =
            (SELECT .STACK_OPERATOR OF
                SET

                [ALWAYS]:

!                   DO THIS FIRST - DETERMINE THE NUMBER OF OPERANDS
!                   NEEDED BY THIS PARTICULAR OPERATOR        o

                    BEGIN
                    VALUE_2 =  EVAL_VALUE();
                    VALUE_1 =
                        (IF .STACK_OPERATOR EQL EQUAL THEN
                            EVAL_ADDRESS()
```

```
                        ELSE
                            IF .STACK_OPERATOR NEQ NEGATIVE THEN
                                EVAL_VALUE())
                    END;

                [NEGATIVE]:

!                   NEGATION - (UNARY MINUS)
                    -.VALUE_2;


                [MULTIPLY]:
                    .VALUE_1 * .VALUE_2;

                [DIVIDE]:
                    .VALUE_1 / .VALUE_2;

                [MINUS]:
                    .VALUE_1 - .VALUE_2;

                [PLUS]:
                    .VALUE_1 + .VALUE_2;

                [EQUAL]:
                    ! STORE THE VALUE IN VALUE_2
                    STORAGE[.VALUE_1] =  .VALUE_2;

                [OTHERWISE]:
                    RETURN(ERROR(8));

            TES);
        PUSH_EVAL_STACK(IS_DECIMAL);
        PUSH_EVAL_STACK(.VALUE_3);
        END;
```

```
ROUTINE EVAL_ADDRESS  =

!++
!
! FUCTIONAL DESCRIPTION:
!
!       THIS ROUTINE IS CALLED WHEN THE ASSIGNMENT OPERATOR IS TO BE
!       EVALUATED.  THE VALUE RETURNED IS THE ADDRESS (INDEX) OF THE
!       IDENTIFIER IN STORAGE.
!
! FORMAL PARAMETERS:
!
!       NONE
!
! IMPLICIT INPUTS:
!
!       NONE
!
! IMPLICIT OUTPUTS:
!
!       NONE
!
! ROUTINE VALUE:
!
!       ERROR(7), ADDRESS(INDEX) OF THE IDENTIFIER FROM THE
!       TOP OF EVAL_STK
!
! SIDE EFFECTS:
!
!       NONE
!
!--

        BEGIN
        LOCAL
            VAL,                                    ! TEMPORARY VALUE
            TYPE;                                   ! TEMPORARY TYPE

        VAL = POP_EVAL_STACK();
        TYPE = POP_EVAL_STACK();
        IF .TYPE NEQ IS_NAME THEN  RETURN(ERROR(7));
        .VAL
        END;
```

```
ROUTINE EVAL_VALUE =

!++
!
! FUCTIONAL DESCRIPTION:
!
!       THIS ROUTINE ACESSES THE VALUE OF THE IDENTIFIER.  THE
!       EVAL_STK VALUE IS USED TO INDEX THE IDENTIFIER VALUE STORAGE
!       AREA (STORAGE).
!
! FORMAL PARAMETERS:
!
!       NONE
!
! IMPLICIT INPUTS:
!
!       STORAGE
!
! IMPLICIT OUTPUTS:
!
!       NONE
!
! ROUTINE VALUE:
!
!       VALUE OF THE IDENTIFIER ON THE TOP OF EVAL_STK
!
! SIDE EFFECTS:
!
!       NONE
!
!--

        BEGIN
        LOCAL
            TYPE,                                   ! TEMPORARY TYPE
            VAL;                                    ! TEMPORARY VALUE
        VAL = POP_EVAL_STACK();
        TYPE = POP_EVAL_STACK();
        IF .TYPE EQL IS_NAME THEN  VAL = .STORAGE[.VAL];
        .VAL
        END;
```

```
ROUTINE PUSH_EVAL_STACK(ELEMENT) :NOVALUE =

!++
!
! FUCTIONAL DESCRIPTION
!
!       THIS ROUTINE "PUSHES" A DATA ELEMENT ONTO THE EVAL_STK.
!
! FORMAL PARAMETERS:
!
!       ELEMENT - DATA TO BE PLACED ON EVAL_STK
!
! IMPLICIT INPUTS:
!
!       EVAL_STK, EVAL_STK_PTR
!
! IMPLICIT OUTPUTS:
!
!       EVAL_STK_PTR
!
! ROUTINE VALUE:
! COMPLETION CODES:
!
!       NONE
!
! SIDE EFFECTS:
!
!       NONE
!
!--

        BEGIN
        EVAL_STK[.EVAL_STK_PTR] = .ELEMENT;
        EVAL_STK_PTR = .EVAL_STK_PTR+1
        END;
```

```
ROUTINE POP_EVAL_STACK =

!++
!
! FUCTIONAL DESCRIPTION:
!
!       THIS ROUTINE "POPS" AN ELEMENT OFF OF THE EVAL_STK.
!
! FORMAL PARAMETERS:
!
!       NONE
!
! IMPLICIT INPUTS:
!
!       EVAL_STK_PTR, EVAL_STK
!
! IMPLICIT OUTPUTS:
!
!       EVAL_STK_PTR
!
! ROUTINE VALUE:
!
!       VALUE POPPED FROM EVAL_STK
!
! SIDE EFFECTS:
!
!       NONE
!
!--

        BEGIN
        EVAL_STK_PTR = .EVAL_STK_PTR-1;
        .EVAL_STK[.EVAL_STK_PTR]
        END;
```

```
ROUTINE PRINT_STRING :NOVALUE =

!++
!
! FUCTIONAL DESCRIPTION
!
!       THIS ROUTINE PRINTS OUT THE EXPRESSION JUST READ IN.
!
! FORMAL PARAMETERS:
!
!       NONE
!
! IMPLICIT INPUTS:
!
!       INPUT
!
! IMPLICIT OUTPUTS:
!
!       NONE
!
! ROUTINE VALUE:
! COMPLETION CODES:
!
!       NONE
!
! SIDE EFFECTS:
!
!       NONE
!
!--

        BEGIN
        TTY_PUT_CRLF();
        INCR I FROM 0 TO .INPUT_LENGTH-1 DO
            BEGIN
            IF .INPUT[.I] EQL CAR_RETURN
            THEN
                EXITLOOP;
            TTY_PUT_CHAR(.INPUT[.I]);
            END;
        END;
```

```
ROUTINE PRINT_STACK =

!++
!
! FUCTIONAL DESCRIPTION:
!
!       THIS ROUTINE PRINTS OUT THE CONTENTS OF MAIN_STK IN SYMBOLIC
!       FORMAT.
!
! FORMAL PARAMETERS:
!
!       NONE
!
! IMPLICIT INPUTS:
!
!       MAIN_STK_PTR, MAIN_STK
!
! IMPLICIT OUTPUTS:
!
!       NONE
!
! ROUTINE VALUE:
! COMPLETION CODES:
!
!       NONE
!
! SIDE EFFECTS:
!
!       NONE
!
!--

        BEGIN
        INCR I FROM 0 TO .MAIN_STK_PTR-1 BY 2 DO
            BEGIN
            TTY_PUT_CRLF();
            SELECT .MAIN_STK[.I] OF
                SET

                [IS_NAME]:
                    TTY_PUT_CHAR(.MAIN_STK[.I+1] + %C'A');

                [IS_DECIMAL]:
                    TTY_PUT_INTEGER(.MAIN_STK[.I+1],10,10);

                [IS_OPERATOR]:
                    TTY_PUT_ASCIZ(.OPNAMES[.MAIN_STK[.I+1]]);

                TES;
            END;
        END;
```

```
ROUTINE ERROR(ERROR_NUMBER) =

!++
!
! FUCTIONAL DESCRIPTION:
!
!       THIS ROUTINE PRINTS OUT ERROR MESSAGES BASED ON THE
!       ERROR_NUMBER PASSED TO IT.  IT ALSO DUMPS THE CONTENTS OF THE
!       MAIN_STK AND PRINTS THE EXPRESSION IN ERROR.
!
! FORMAL PARAMETERS:
!
!       ERROR_NUMBER - INDEX INTO ERROR MESSAGE PLIT
!
! IMPLICIT INPUTS:
!
!       ERROR_MESSAGE
!
! IMPLICIT OUTPUTS:
!
!       NONE
!
! ROUTINE VALUE:
!
!       1
!
! SIDE EFFECTS:
!
!       NONE
!
!--

        BEGIN
        MACRO
                MESSAGE(ARGUMENT) = PLIT (%ASCIZ ARGUMENT)%;

        BIND
                ERROR_MESSAGE = PLIT(
                    MESSAGE('ERR:0      NONE'),
                    MESSAGE('ERR:1      ILLEGAL CHARACTER ON INPUT'),
                    MESSAGE('ERR:       OPR EXPECTED, NOT FOUND'),
                    MESSAGE('ERR:3      EXCESS CLOSE PAREN'),
                    MESSAGE('ERR:4      ILLEGAL UNARY OPERATOR'),
                    MESSAGE('ERR:5      ILLEGAL USE OF UNARY MINUS'),
                    MESSAGE('ERR:6      OPERATOR MUST FOLLOW ")"'),
                    MESSAGE('ERR:7      ASSIGNMENT TO NON VARIABLE'),
                    MESSAGE('ERR:8      BAD OPERATOR ON STACK'),
                    MESSAGE('ERR:9      EXCESS OPEN PAREN'),
                    MESSAGE('ERR:10     NONE')
                                ):VECTOR[50];

        TTY_PUT_CRLF();
        TTY_PUT_MSG(.ERROR_MESSAGE[.ERROR_NUMBER],OUT_MSG_MAX);
```

```
        PRINT_STACK();
        PRINT_STRING();
        RETURN 1
        END;
END
ELUDOM
```

[End of Appendix C]

INDEX

Draft

Oct. 15, 1983

THE DIGITAL COMPUTING ENVIRONMENT

Gordon Bell
Encore Computer Corporation
(formerly Digital Equipment Corporation)

Bill Strecker
Digital Equipment Corporation

ABSTRACT

The Digital "E" Enviroment for computing is the aggregate of a wide range of compatible computers and ways of indefinitely interconnecting them to provide its users with both generic (eg. word processing, electronic mail, database access, spread sheet, payroll) and basic (eg. languages, file sytems) computing facilities that he may use directly, or "program". The intent is to provide the widest range of choices, by having complete compatibility, for where and how computing is to be performed without having to make a priori commitments to a particular computer system class (i.e. mainframe, minicomputer, team computer, personal computer). The design of the Environment is substantially more than a single range of compatible computers because different styles of use are required depending on the machine class and all the computers must be interconnected and work together. The Environment is the first of what we define as a multi-level, homogeneous computing architecture.

BACKGROUND

BATCH MAINFRAMES FOR CENTRAL SERVICES

In the first two computer generations, 1950-1970, computers were used in batch processing under the name of mainframe computing. During the 70's the mainframe began to be used almost interactively from remote job entry terminals at 'glass key punches'. The general direction is to have larger mainframes and larger terminal networks that interconnect to a single computer by an array of front end computers. When more power is required, more switching computers are connected to several mainframes each of which perform a particular function. Attached, dual processors are used to provide increased power for what is fundamentally a single system. Over time, the evolution will be to small scale, multiprocessors for incremental performance and higher availability.

MINICOMPUTERS AND TIMESHARING FOR A GROUP

In the mid-60's both minicomputers and timesharing were developed at Digital around the PDP-8 and PDP-10 computers, respectively. Minicomputers were initially used as components of real time systems and for personal computing. The LINC minicomputer, developed at M.I.T.'s Lincoln Laboratory was the first personal computer, providing a personal filing system and the ability to write and run programs completely on line.

Timesharing started out as a centralized mainframe facility for a large group. Access was via individual Teletypes which were eventually replaced by cathode ray tube terminals, or 'glass Teletypes'. By the mid-70's low cost PDP-11 timeshared computers began to be used by seperate groups and departments to provide 'personal computing'. In the early 80's, low cost disks and large memories permitted two evolved computer structures: the 32-bit supermini, and the microprocessor based 'team computer'. The supermini had all the power of its mainframe ancestors, especially the critical 32 bits to access memory. The 'team computer' based on modern, powerful microprocessors is simply much lower priced, (eg. $15,000) providing 'personal computing' at a price below personal computers.

MICROPROCESSORS, PERSONAL COMPUTERS AND POWERFUL WORKSTATIONS

The fourth generation appeared in 1972 with the microprocessor. With the second 8-bit generation microprocessor, floppy disks and 16 Kilobit semiconductor memories, circa 1976, Personal Computers were practical and began to be manufactured by Apple, Commodore, Radio Shack, etc. With 16-bit microprocesor and 64 Kilobit rams, the second generation of ~~PCs~~ pc's appeared in the early '80's.

In 1979, Carnegie-Mellon University wrote a proposal for personal computer research, stating:

> 'The era of time-sharing is ending. Time-sharing evolved as a way to provide users with the power of a large interactive computer system at a time when such systems were too expensive to dedicate to a single individual...Recent advances in hardware open up new

possibilites...high resolution color graphics, 1 mip, 16 Kword microprogrammed memory, 1 Mbyte primary memory, 100 Mbyte secondary memory, special transducers,...We would expect that by the mid-1980's such systems could be priced around $10,000.'

Today's powerful Personal Workstation such as the Apollo or SUN Workstation connected with shared facilities on a Local Area Network characterize this type of machine.

Numerous information processing products are possible using the modern, high performance microprocessor. These include:

- terminals and smart terminals
- personal computers and special word processors
- high performance workstations
- PABXes for voice and data
- Smart telephones and telephone-based terminals
- Conventional, shared supermicros
- High availability supermicros using redundancy to form seperate computers or seperate processors within a single computer

PERSONAL COMPUTERS CLUSTERS AS AN ALTERNATIVE TO SHARED COMPUTERS

In the early 70's Xerox Research Park researchers developed and provided itself with a personal computing environment consisting of a powerful personal computers all linked together via the first Ethernet cable (3 Mbits), and created the notion of the Local Area Network. Their network had various specialized function servers, including a shared central computer that was compatible with the DECsystem 10, for archival memory and large scale computation.

Figure Evolution shows the hardware and software of a multiprogrammed computer used for timesharing, and the corresponding structure of a Personal Computer Cluster consisting of functional services and interconnected by a common interconnect which provides basically the same capability. The timeshared system has a central memory containing various jobs connected to terminals and operating system which attends to the users and handles the particular functions (eg. real time, files, printing, communication). Personal Computers are connected to timesharing systems as terminals. By comparing the shared system with the systems formed from functionally independent modules, one would expect two design approaches:

1. decomposing systems to provide shared LAN services; and
2. aggregating Personal Computer to Form PC Networks and Clusters.

Decomposing Systems to Provide Shared LAN Services

As shared computers become more complex and more centralized, it's desireable to decompose the functions for execution on smaller computers that can be distributed to be nearer the use. Thus, the decomposition of a shared system into various boxes, each of which perform a unique function permits the evolution of the parts independent of the whole, the physical distribution of a function and the ability of several computers to share a function. While we have scribed the evolution of LANs as a decomposition of a single system, LANs are generally an aggregate of heterogenous systems which access a shared service of some kind as described below.

LANs differ from Wide Area Networks (WANs) in that they assume a low . ency, high bandwidth interconnect. This permits file access as well as file transfer applications. With file access, it is possible to remotely locate part or all of a system's mass storage to a file serving computer. File access requires bandwidth and latency which are roughly equal to that of a disk (i.e. 10 Mhz rates) file transfer can be done at substantially slower rates (56Khz to 1 Mhz).

Using the reasoning which allowed the formation of the file server, we continue the decomposition of a large central system into servers or stations and then combine these servers into a LAN. The major servers are:

1. Person Server (personal computer or workstation) - local computation and human interface, possibly private storage of files
2. File Server - mass storage
3. Compute Server - batch computation or existance of particular programs
4. Print Server - printing
5. Communication Server - terminal, telephone and PABX, Wide Area Network access including international standards, other companies (eg. SNA)
6. Name/Authentication/Directory Server - naming the networks resources and controlling access to them.

LAN formed as a complete decomposition of a single system and ontaining no other incompatible servers would be defined as a homogeneous cluster of Personal Computers or Workstations.

Aggregating Personal Computers to Form PC Networks and Clusters
As personal computers require more facilities (e.g. printing, communication and files), and the number and type of PCs grow, the need to directly communicate for sending messages and sharing files. Furthermore, as a collection of computers in one place forms, economy is gained by sharing common facilities such as printers, phone lines, and disks. Applenet and Corvus Omninet are relatively short and low data rate Local Area Networks used to permit the construction of what might best be called a network of Personal Computers because of the heterogenity of type. The 3 Com system for interconnecting IBM PCs is more characteristic of the homogeneous network, or cluster.

For a PC Cluster, one would expect to have a single File Server which can supply records at random to any of its constituency. Table (of what timesharing, PC's and PC clusters provide) shows what timesharing, PC's and PC Clusters provide.

DISTRIBUTED PROCESSING USING CAMPUS AND WIDE AREA NETWORKS
The proliferation of timeshared computers required the development of networking in order for various systems to communicate with one another and to mainframes. Thus, dispersed computing became distributed computing. Store and forward wide area networks evolved om the ARPA-net, which was used to interconnect timeshared mainframe computers (mostly PDP-10's).

Campus Area Networks

When a collection of Local Area Networks are connected together in a single area which extends beyond a typical LAN, we call this a Campus. Universities clearly typify the campus as does a collection of buildings. Gateways are used to interconnect LANs of different type (eg. Omninet, Ethernet, 802 Rings, Applenet, Arcnet, PCnet), whereas bridges or repeaters are used to interconnect networks of the same type to form one larger network.

Wide Area Networks

WANs are characterized by low bandwidth, high latency, and autonomous operation of the nodes. The applications typically include: mail, file transfer, database query, and low interaction remote terminal access. Wide Area Networks can be constructed in several ways: direct dial up using conventional circuit switching using voice grade circuits, an intermediate store and forward network such as Telenet, or a hybrid approach where various worker computers do store and forward switching.

THE E

Although the specific design of the E began in December 1978 with the approval of the Board of Directors, it's origins include:

- the original VAX-11 goals for a 1000:1 range of computers,
- evolution of distributed processing minicomputer networks, in Wide Areas, 'Campuses', and Local Areas,
- the appearance of powerful Personal Computers and Local Area Networks, permitting the aggregation of tightly coupled 'PC networs and clusters' that provide some of the benefits of timeshared minicompters and mainframes,
- the ability to aggregate minicomputers and mainframes into multiprocessors and multicomputer clusters that appear to be a 'single' system in order to provide higher reliability, higher performance and incremental performance.

The December '78 statement of the Distributed Computing Environment, Fig. DCE 12/78, and subsequent evolution [shown in brackets] was:

> 'Provide a set of homogeneous distributed computing system products based on VAX-11 so a user can interface, store information and compute without re-programming or extra work from the following computers system sizes and styles:
>
> - via [a cluster of] large, central (mainframe) computers or network;
> - at a local, shared departmental/group/team (mini) computer, [and evolving to a minicomputer with shared network servers];
> - as a single user personal (micro) computer within a terminal [and evolving to PC Clusters];
> - with interfacing to other manufacturer and industry standard information processing systems; and
> - all interconnected via the local area Network Interconnect, NI (i.e. Ethernet) in a single area, and the ability of interconnecting the Local Area Networks (LANs) to form Campus Area and Wide Area Networks.'

Fig. DCE 12/7 shows the origin of the 'E' shape that characterizes the present Environment of Fig. E. The three horizontal segments of the E provide the different computing classes which roughly correspond to different priced computers; the functions are described in the Table of Computing Styles. In order to implement the environment, many requirements were initially posited, and several developments evolved from necessity:

- a range of VAX-11 and 11 compatible computers to meet the requirements of the various computing styles based on different classes of computers;
- interconnection schemes and the corresponding protocols for building multiprocessors, tightly coupled centralized VAX Clusters, LAN-based PC Clusters, LANs, Campus Area Networks and Wide Area Networks;

GOALS OF THE ENVIRONMENT

NGE GOALS

The important goals and constraints of the Environment are contained in the original statement about what the Environment, which is simply "to provide a very wide range of interconnectable VAX-11 computers". The original goal of VAX was to be able to implement the range [for what appears to be a single system] of a factor of 1000 price range... with no time limit given. Since a given implementation tends to provide at a maximum, a range of 2-4 in price and 10 in performance if performance is measured as the product of processor speed times memory size, then many models and ways of interconnection were required.

At the time the 780 was introduced, the total range of products for both the VAX-11 and 11 family was almost 500, beginning with $1,000, LSI-11 boards and going to a $500,000 VAX-11/780. If the LSI-11 is used as a Personal Computer, the price range is reduced to only a factor of 50! While the two ends of the system were "compatible" and could be interconnected via DECnet, they lacked the coherency necessary for a fully homogenous computing environment.

By introducing "VAX Clusters", the range can be extended by a factor of up to the number in the cluster. For VAX, we now have a price range of from about $50,000 for a 730 to about $7.5 million for a cluster of 12, 782 dual processors and a corresponding performance range of over 100. In the following section, we will show how the cluster provides what appears to the user as a single system.

Obviously both higher performance machines, and VAX on a chip microprocessors are necessary in order to attain the range and computing style goals.

STATIC AND DYNAMIC ASSIGNMENT OF PROGRAMS TO NODES

Ideally, a user can decide on how to compute on a completely variable basis at the following times:

- at sytem purchase or rent time ranging from outside facilities reached via gateways, to a central facility, to a shared department or team computer, to a users own personal computer
- at system use time, ranging from access via a terminal, or personal computer interconnected to the system LAN or a particular, shared computer. Here, work is bound statically to a particular set of system resources. Most likely, particular nodes would execute special programs on data located at the node.
- at task time on the basis of reliability. VAX clusters provide the complete dynamic
- at task use time on a completely dynamic basis, ranging from computing on his own local system to being able to collect any resources and move work dynamically while programs are in execution. With this ability, as a program goes though its various stages of development, it might be moved from small system to large system to take advantage of increased computational power at higher level nodes.
- at task time on a dynmic basis with the ability to acquire arbitrary resources to engage in parallel computation.

VAX clusters

1.0 INTRODUCTION

Typically, multi-computer systems fall into one of two classes: tightly coupled and loosely coupled. Tightly coupled systems are charcterized by close physical proximity of the processors, high bandwidth interprocessor communication through shared primary memory, and a single copy of the operating system. In contrast, loosely coupled systems are characterized by physical separation of the processors, lower bandwidth message oriented interprocessor communication, and independent operating systems.

Recent hardware and software engineering work at DEC has focused on building multicomputer systems whose goals are (1) high availability and (2) easy extensibility to a large number of processors and mass storage controllers. To support these goals, a loosely coupled rather than a tightly coupled system was selected.

From a software perspective, the independent operating systems of the loosely coupled system increase the survivability of the system in the presence of operating system failures.

From a hardware perspective, it is difficult to design the physical packaging and memory system of a shared primary memory multiprocessor which is both adequately low in cost for small numbers of processors and mass storage controllers and adequately high in performance for large numbers of processors and mass storage controllers. In contrast, a group of standard single processor computers linked by an external message oriented interconnect facilitates building cost effective systems over a wide range.

A key issue with the loosely coupled approach is system performance. The performance is gated by (1) the bandwidth of the computer interconnect and (2) the software overhead of the associated communications architecture. To address this issue, three steps were taken:

1. The development of a very high speed message oriented computer interconnect termed the CI (for Computer Interconnect).

2. The development of a simple, low overhead communications architecture whose functions are tailored to the needs of highly available, extensible systems. This architecture is termed SCA (for System Communication Architecture).

3. The development of an intelligent hardware interface to the CI which implements much of SCA. This interface is termed the CI Port.

The loosely coupled system concept has been applied to the VAX-11 computer family and the VAX/VMS operating system. The resulting structure has been given the name VAXcluster. The following discusses the VAXcluster hardware and software.

2.0 VAXCLUSTER HARDWARE

Fig. 1 shows the hardware structure of a VAXcluster. There are five major component types: the CI, the CI Port, the Computer, the HSC50 mass storage controller, and the terminal and communications subsystem.

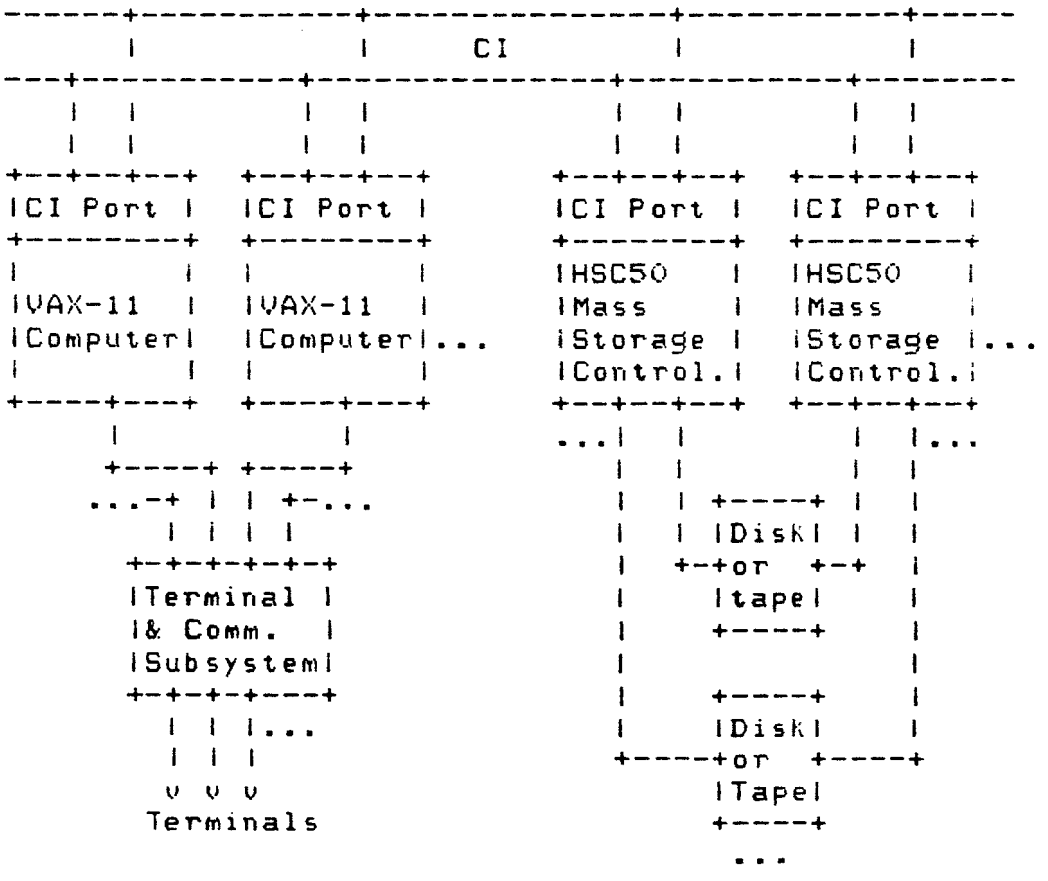


Fig. 1 Cluster Hardware Structure

2.1 CI

The CI is dual path serial bus with each path supporting a 70 megabit/sec transfer rate. The primary purpose of the dual paths is to provide a backup in case of path failure; but when both paths are available, they are used together to provide an increased transfer rate. Each path is implemented in two coaxial cables: one for transmitted and one for received signals.

The physical topology of the CI is that of a star: a central hub termed a Star Coupler with radial paths of up to 45 meters extending to each of the CI nodes. See Fig. 2. The Star Coupler is a passive electronic device (transformers and resistors) which supports up to 16 nodes which may be either host computers or HSC50 mass storage controllers.

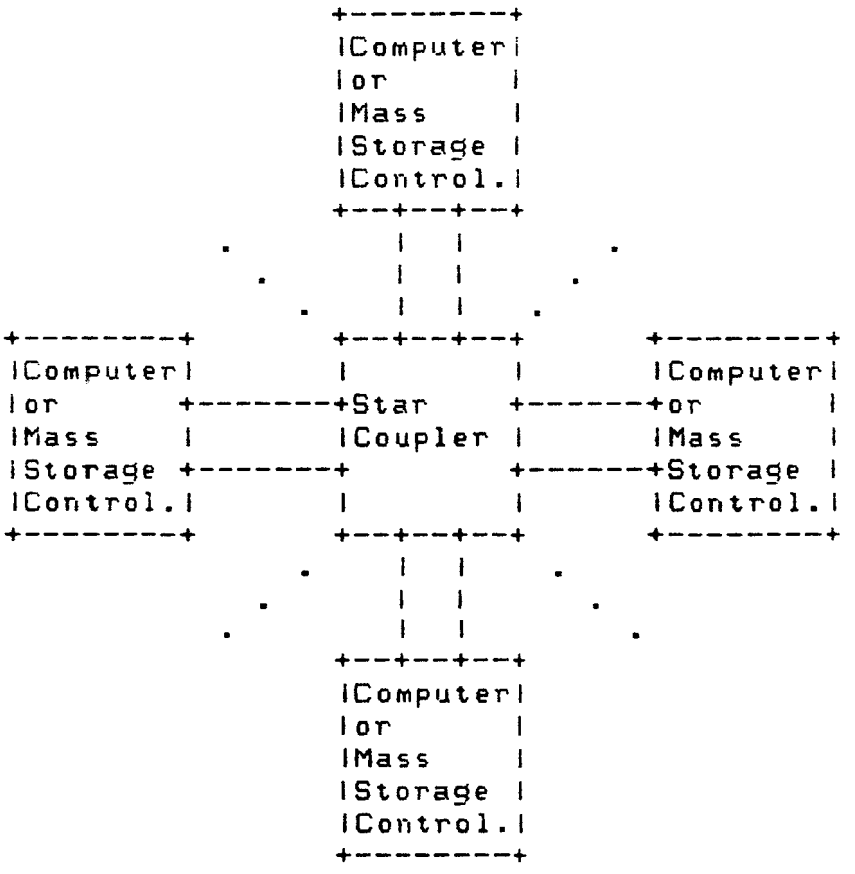


Fig. 2 CI Topology

A star topology was selected over a linear topology for three reasons:

1. The efficiency of arbitration of a serial bus is related to the transit time between the the most widely separated nodes. The star topology permits the nodes to be located anywhere within a 45 meter radius circle (about 6400 sq. meters) with a maximum node separation of 90 meters. In general, a linear bus threaded through 16 nodes in the same area would greatly exceed 90 meters in length.

2. The central star coupler permits the addition and removal of nodes with a minimum risk of electrical or mechanical disruption of the CI.

3. The star topology facilitates two possible future extensions to the CI: (1) an active Star Coupler permitting many more than 16 nodes and (2) replacement of the coaxial cables with fiber optic cables.

Data is transferred on the CI in variable length packets containing up to 512 data bytes. After each packet is sent and acknowledged, the CI is arbitrated. A deterministic round-robin arbitration scheme is employed guaranteeing all nodes equal access. All CI packets are protected by a 32-bit Cyclic Redundancy Check.

Interfaces to the CI are transformer coupled providing electrical isolation between the nodes. The CI has been engineered to rigid guidelines in both electromagnetic radiation and susceptibility.

2.2 CI Port

The CI Port is the interface between the CI and the host computer. The CI Port offloads much of the work in communicating among host computers and mass storage subsystems.

At the lowest level, the CI Port performs CI arbitration. After winning the arbitration, the sending Port -- Port 1 in Fig. 3 -- sends a data packet and waits for receipt of an acknowledgement packet. If the data packet is correctly received, the receiving Port -- Port 2 in Fig. 3 --immediately returns an acknowledgement packet without arbitrating the CI. If a positive acknowledgement is not received, the sending CI Port retries the operation up to a specified retry limit.

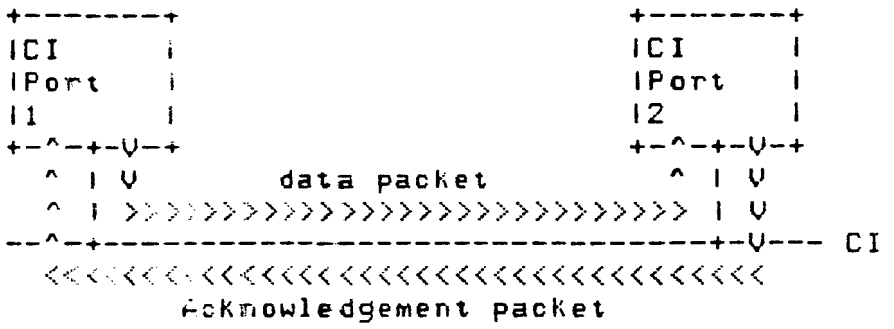


Fig. 3. Data and Acknowledgement Packets

When both CI paths are available, the CI Port statistically distributes transmissions across both paths. This insures load balancing and continual verification that both paths are operational. The notion of path availability is maintained on a per node basis in each CI Port in a data structure termed the path status table. Fig. 4 shows a hypothetical path status table for node 1. Note that is possible that path 1 might be bad to one node and path 2 be bad to another node.

Node	Path 1	Path 2
0	bad	good
1	good	good
2	good	good
=		=
15	good	bad

Fig. 4. Path Status Table for Node 1

The CI Port will only attempt to use good paths as indicated in the path status table. If the CI Port is unsuccessful in sending data on path marked good after the retry limit, it will change the path to bad in the path status table. If the other path is good, the CI Port will automatically try up to the retry limit on the other path. Only if both paths are bad will a CI Port operation fail.

A key contributor to the performance of the VAXcluster is the capability of the CI Ports to transfer data between virtual memory buffers in different nodes. By issuing a single command to its local CI port, a host computer can read the contents of an arbitrarily sized buffer in remote node into a local buffer or write the contents of an arbitrarily sized local buffer to a buffer in a remote node. See Fig. 5.

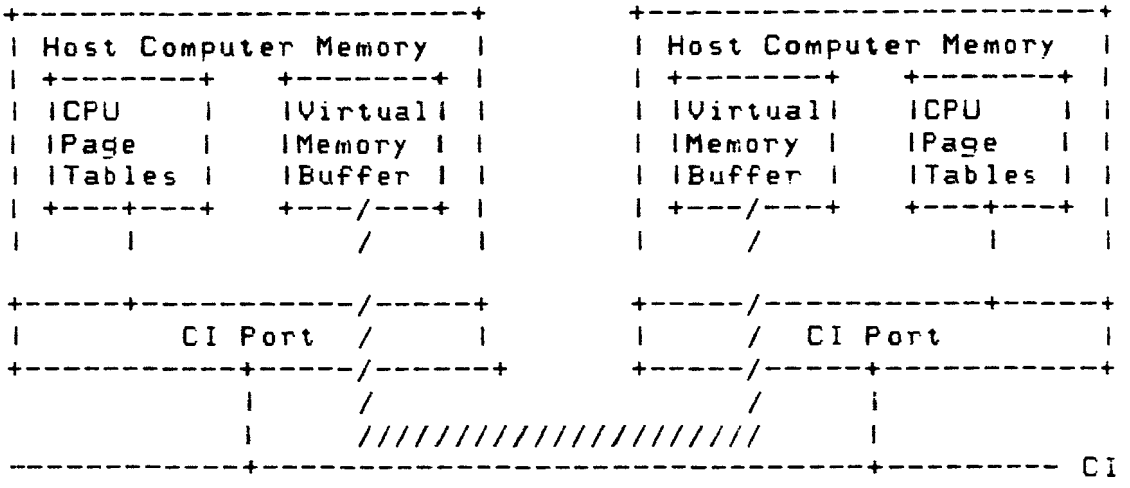


Fig. 5. Virtual Buffer Transfers

The local and remote CI Ports cooperate in this operation by:

1. Breaking the transfer up into maximum sized CI data packets.

2. Insuring that all packets of the transfer are correctly sent and received.

3. Referencing the virtual memory buffers specified in the transfer through the same page tables used by the host computer CPU.

2.3 Computer

A host computer is any VAX-11 family member which has available a CI Port. Currently, this includes the medium scale VAX-11/750 and large scale VAX-11/780 and VAX-11/782 computers. As new medium and large scale VAX-11 family members become available, they will have available CI Ports.

2.4 HSC50

The HSC50 is a controller subsystem supporting disks and tapes. Each HSC50 supports a combination of six disk or tape channnels. A disk channel supports up to four disks while a tape channnel supports up to 16 tapes.

The HSC50 allows direct access by all host computers to all mass storage devices. Thus a mass storage request by one host computer in the VAXcluster does not require the intervention of the other host computers.

All mass storage devices on an HSC50 can be dual ported to another HSC50. If the HSC50 controlling a dual ported device fails, the other HSC50 automatically assumes control.

To optimize the throughput of the VAXcluster, the HSC50 performs both seek and rotational optimization across outstanding disk operations. With these optimizations disk requests are not serviced in the order issued but rather in the order which will sustain the highest completion rate.

To protect against data loss due to disk failure, the HSC50 provides disk shadowing. Two disk drives can be combined into what is termed a disk shadow set. Whenever a computer issues a disk write operation to a drive in a shadow set, the HSC50 automatically writes to both drives in the disk shadow set. Thus both drives in the shadow set have identical data. If one of the drives in the shadow set fails, the other drive continues to provide operation with no data loss.

The HSC50 provide other capabilities to offload the host computers. The HSC50 presents an error free logical disk to the host. All details of disk geometry (e.g sector size, track size, number of surfaces) are hidden. All bad block handling, operation retry, and error correction is done by the HSC50. Disks can be copied and disks can be backed up to tape without host computer intervention.

2.5 Terminal And Communications Subsysytem

The needs of terminal and other external communications support varies widely with application. In some cases direct attachment to the individual computers is adequate. VAXcluster software allows any terminal to transparently communicate to any host computer regardless of the host to which the terminal is physically attached. However failure of the host computer to which a terminal is directly attached would deny access to the VAXcluster by that terminal.

The first alternative to directly attached terminals is the use of a standard communications switch. Terminals and host terminal ports are attached to the switch. On failure of a host computer, a terminal connected to that host computer is reconnected to another host computer through the switch.

The second alternative is to use Ethernet based terminal concentrators. All of the terminal concentrators are attached to all of the host computers through an Ethernet. Host computer failures do not isolate terminals because all terminals have access to all host computers.

3.0 VAXCLUSTER SOFTWARE

The VAXcluster software structure is a set of independent VAX/VMS operating systems which communicate using SCA services. To understand the evolution of VAX/VMS from a single computer operating system to the VAXcluster operating system, a simple example of file processing is considered for both the single computer and the VAXcluster environments. Fig. 6 shows the single computer environment.

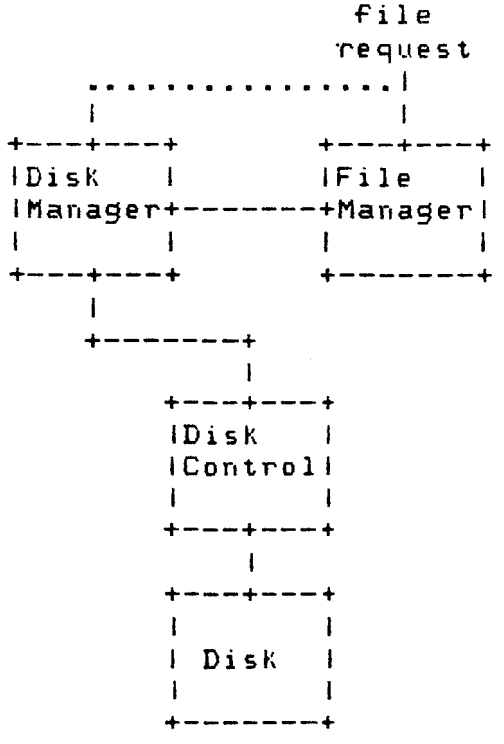


Fig. 6. File System Components in a Single Computer

File requests are passed to the file manager (the File ACP in VAX/VMS terminology) which manages the file system. The file manager maps the file operation into reads or writes of specific disk blocks and calls the disk manager (the disk class driver in VAX/VMS terminology). The disk manager performs the physical reading or writing of the disk. In actual VAX/VMS implementation the file manager and disk manager are structured such that the file manager can be bypassed for many file requests.

Fig. 7. shows the VAXcluster environment.

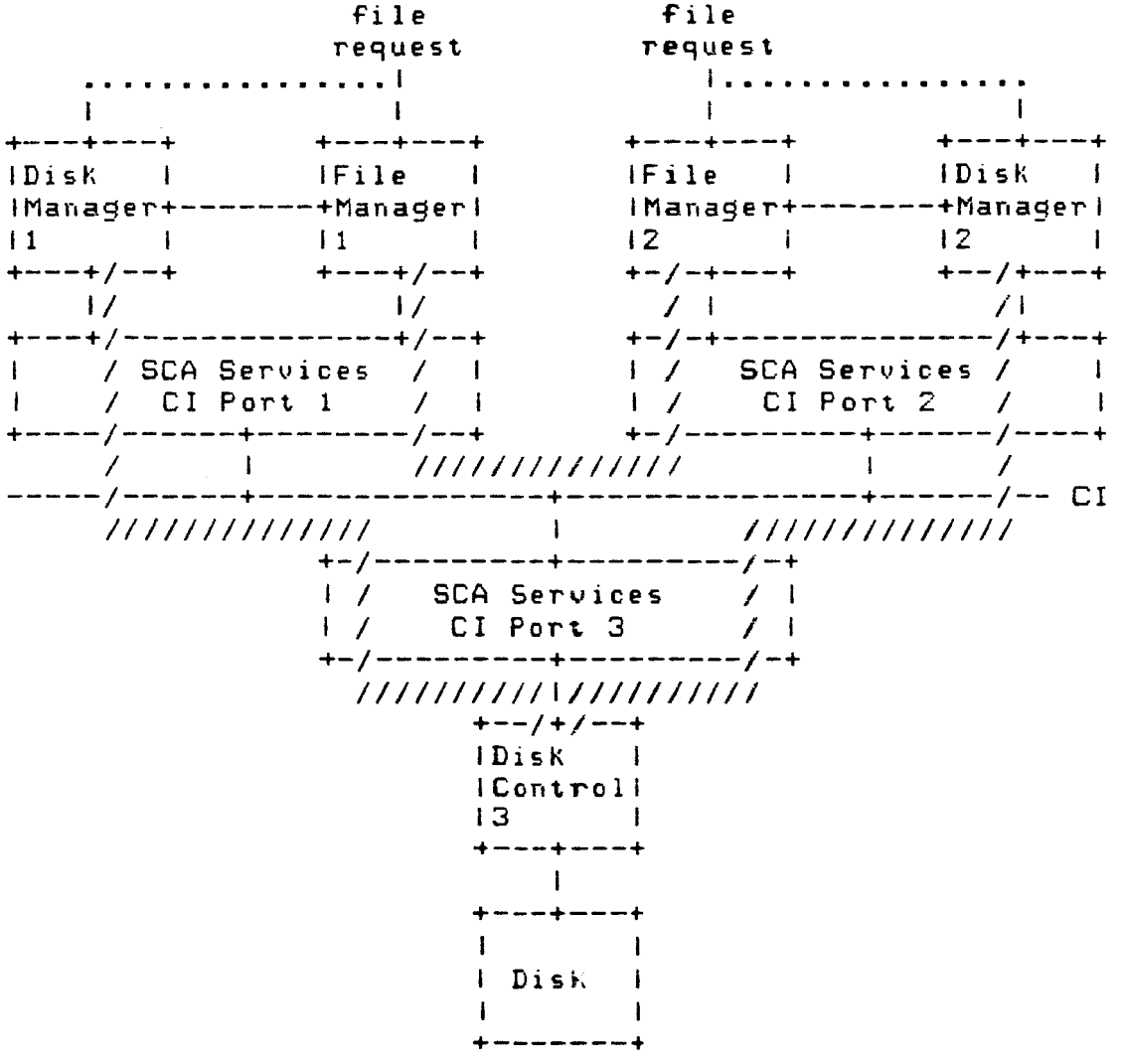


Fig. 7. File System Components in a VAXcluster

There are three nodes on the CI. Nodes 1 and 2 are VAX-11 host computers. Node 3 is an HSC50. The host computers have file managers and disk managers structured as

before. The SCA services software and CI Ports provide communication paths between the disk managers and the disk controller and between the file managers. The disk manager/disk controller path allows each host computer to directly access the disk. The path between the file managers is used to synchronize state changes in the file system insuring that each file manager has a consistent definition of the file system.

In general, each of the VAX/VMS functions which has been distributed in the VAXcluster environment is under the control of a manager which uses the SCA services to pass data and synchronize. The following describes SCA and the major VAX/VMS functions.

3.1 SCA

SCA provides the mechanism by which the software components of the VAXcluster communicate. An implementation of SCA provides four basic communication services:

1. Connection management - a connection is a logical communication path between two processes. In Fig. 7 there are three connections: disk manager 1 to disk controller 3, disk manager 2 to disk controller 3, and file manager 1 to file manager 2. The connection management service establishes these connections. Once a connection is established, the following three SCA services can be used.

2. Datagram transfer - a datagram is a small information unit (typically tens to hundreds of bytes) which is sent over a connection. Delivery of a datagram is not guaranteed. Datagrams, for example, are used for certain type of status information which is periodically generated and whose loss is not critical. Datagrams are also used by software components which have their own protocols for insuring reliable communication: the DECnet network manager is an example of this.

3. Message transfer - like a datagram a message is a small unit of information sent over a connection. Delivery of a message is guaranteed and requires a more elaborate protocol than for datagram transfer to insure the delivery. Messages, for example, are used to carry disk read and write requests.

4. Block data transfer - block data is arbitrarily sized data moved between virtual memory buffers. Delivery of block data is also guaranteed. The block data service, for example, is used to move the data associated with disk read or write

requests.

3.2 Distributed File System

The most important concept of the VAXcluster is the distributed file system in which all VAXcluster files appear local to each of the VMS operating systems in the VAXcluster. The distributed file system supports the VAXcluster goals of availability and extensibility as follows:

1. As additional disks are added to the VAXcluster, the associated files are immediately available to all computers in the the VAXcluster. Similarly, as additional computers are added to the VAXcluster, all file storage is immediately available to the new computers.

2. When a computer implementing an application fails, the application can be moved to another computer in the VAXcluster. The new computer has exactly the same access capabilities to the application's files as did the failed computer.

3.3 Distributed Lock Manager

An important function in any file system supporting sharing is the ability to selectively lock files and records. Other resources similarly need to be locked. Rather than create locking mechanisms specific to each resource to which locking might be applied, a common facility is provided in the VAXcluster.

The distributed lock manager provides a namespace in which processes can lock and unlock resource names. If a process requires access to a resource whose name is locked that process can be queued in a wait state until the resource becomes available.

The namespace provided by the distributed lock manager is tree structured allowing complex objects to be locked at different levels. For example, the tree structure reflects the correct relation between locks on a file and locks on the individual records of a file.

The distributed lock manager provides deadlock detection. The lock manager detects cases where a set of processes are all waiting for resources held by other processes in the set. As a simple example of this, consider

two processes 1 and 2 and two resources A and B. Process 1 has locked resource A and is blocked waiting for resource B. Process 2 has locked resource B and is blocked waiting for resource A. In the absence of deadlock detection, both processes would wait indefinitely.

The distributed lock manager also recovers from host computer failure. Whenever a host computer fails, the locks held by all processes running on that computer are released allowing any waiting processes on other host computers to continue.

3.4 Common Journaling Facility/Recovery Unit Facility

A journal records over time the state changes of a file. A journal can be used to 'rollback' a file from a later state to an earlier state or 'roll forward' a file from an earlier state to a later state. By recording the source of file state changes, a journal provides an audit trail. The Common Journaling facility provides journaling for VAXcluster distributed file system.

It often occurs that a database operation involves a series of changes to one or more files. To maintain the consistency of the database either all the changes must be made or none of them. The series of changes can be defined as a recovery unit. If a failure occurs within a recovery unit, the Recovery Unit facility uses the Common Journaling facility to restore the file system to the state prior to the start of the recovery unit.

3.5 Checkpoint/Restart Facility

The checkpoint/restart facility is valuable in protecting lengthy computations against host computer failure. An application can periodically issue checkpoint calls. The checkpoint/restart facility saves the complete state of the application process. If a host computer fails, the application can be restarted from the most recent checkpoint rather than from the beginning. Prior to restarting, the Recovery Unit facilty is used to restore files back to their state at time of the last checkpoint.

4.0 SUMMARY

The strength of VAXcluster approach is that it builds a sophisticated available and extensible computing structure using general purpose computer hardware and a general purpose operating system. To get availability and

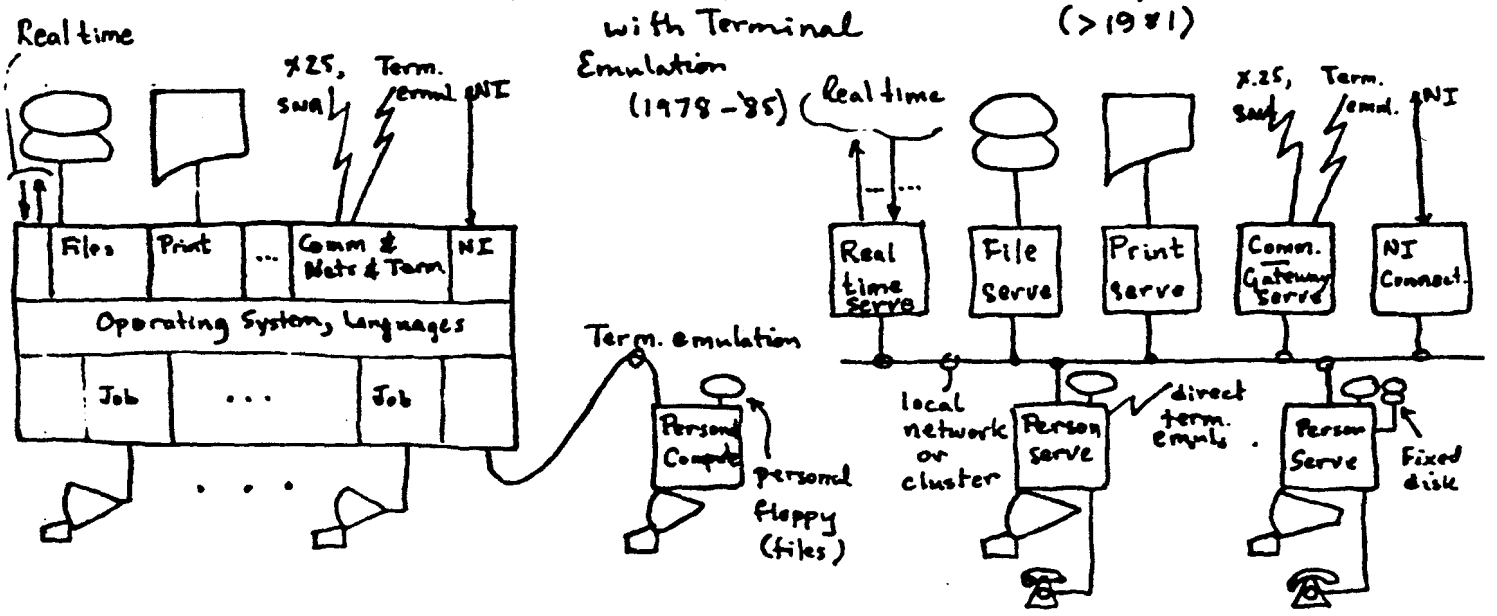
extensibility is no longer necessary to choose specialized hardware with limited peripheral options and choose a specialized operating system with limited software support. Available and extensible structures can now be built with a full range of VAX-11 hardware and the VAX/VMS operating system which offers interactive, batch, and realtime modes of operation and a broad range of layered software products.

Evolution from Timesharing to PC Clusters

Departmental Timesharing of the 70's ('65 ~ '85)

Personal Computers with Terminal Emulation (1978 - '85)

Personal Computer Clusters & Networks of the '80's (>1981)



Estimated (G. Bell) Percentage (in # terminals) of Computer Use Versus time (1950-1990)

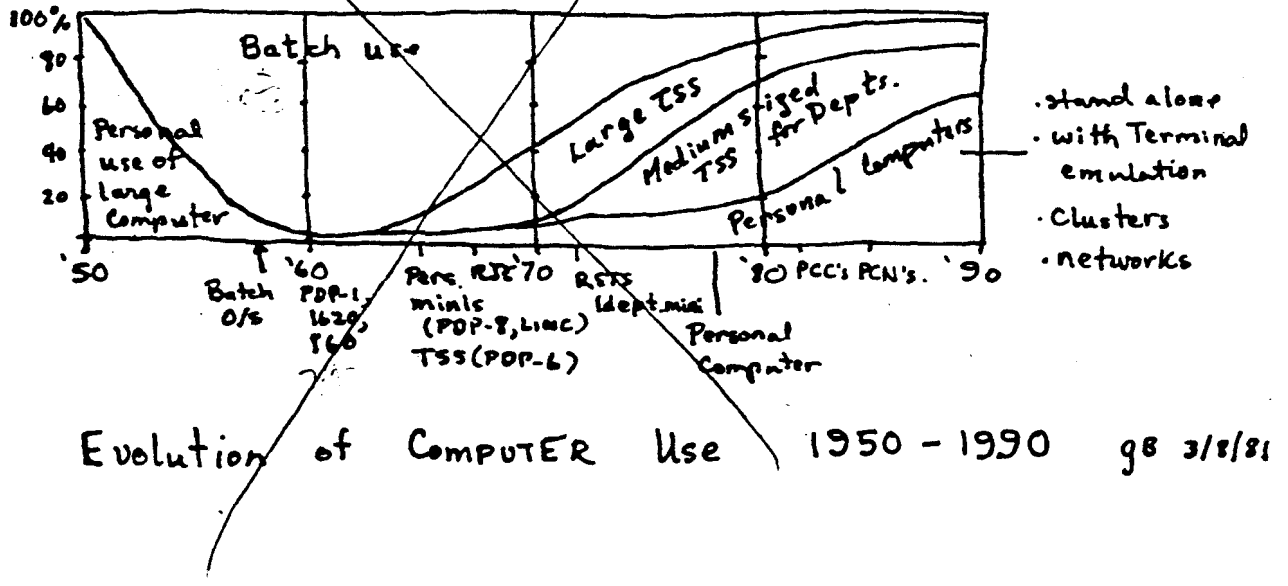
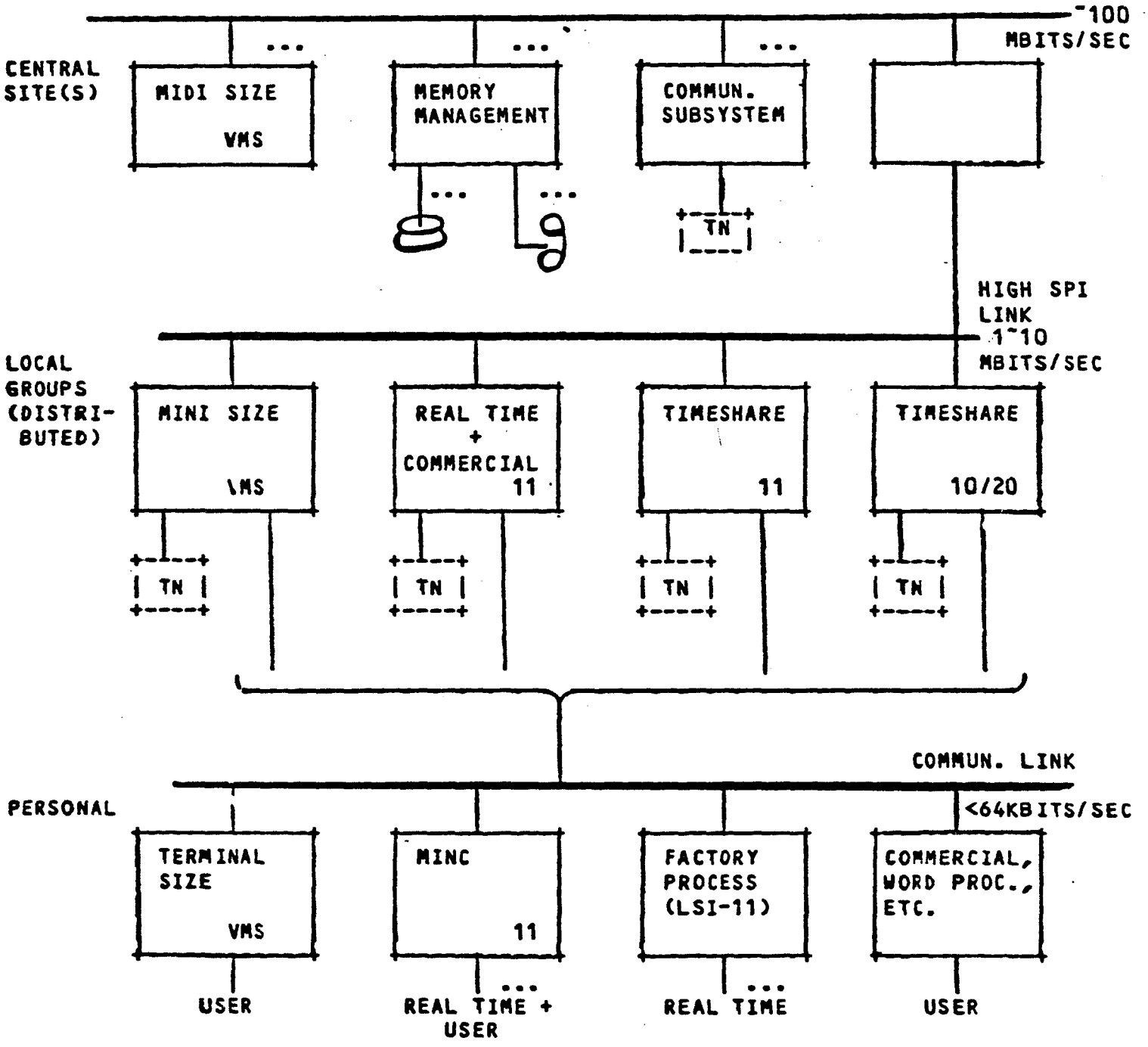


Fig. PC Evolution of Computer Use 1950 - 1990 gB 3/8/81

DISTRIBUTED COMPUTING

ENVIRONMENT

Copied from the original,
as presented to BOD 12/78
by Gordon Bell



TN = TERMINAL NETWORK -- CONNECTS SIMPLE TERMINALS, MOST PERIPHERALS,
AND PERSONAL COMPUTERS AND PROCESS COMPUTERS

The Environment "E"

Fig. Product Strategy

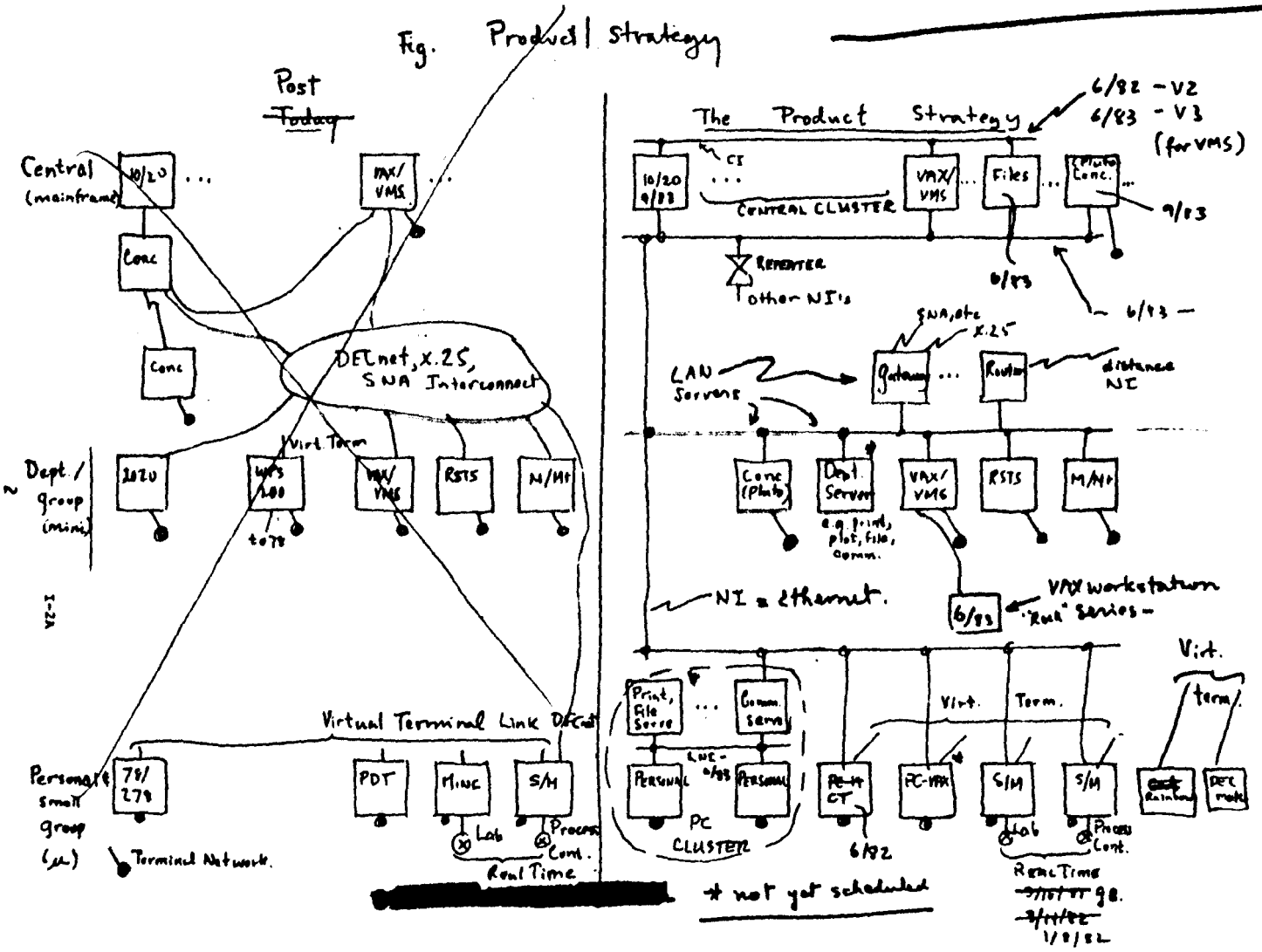


Table of Computing Styles

type*	central	departmental / group / team	personal
machine	mainframe	supermini / mini / supermicro (High availability transaction processor)	micro
size range	500K - 5M	10K - 500K	1K - 10K 10K - 50K**
communications	connected to terminals	to mainframes, peers and PC's	terminal emulation
database	organizations, vendors database	organization, project function	personal files
terminal access	glass keyboard	glass Teletype	direct crt
applications	corporate accounting, electronic mail,	project CAD, order entry, small business	word processing, financial analysis, CAD**
operation	a staff providing service	distributed among group	by user

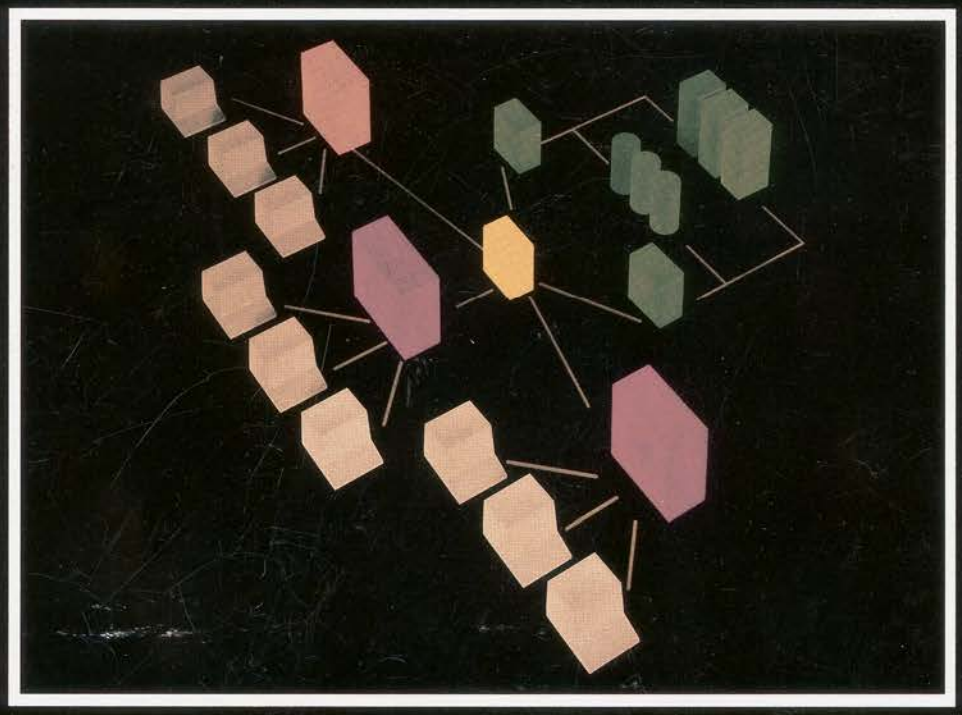
Not including required styles supercomputers
* Workstations used in a area at a time

Table of What Timeshared, PC's, PC Cluster an PC Networks Provide
(Circa 1981, GB)

~~TABLE: WHAT TSS, PC'S AND PC CLUSTERS OR NETWORKS PROVIDE~~

What	Timeshared System	Personal Computer	PC Cluster/ Networks.
processing	highest peak	lo-med, guaranteed	= PC
programs size	very high peak	small to medium	= PC
filing	large	small, guaranteed (+ off line)	= PC and TSS
communication	network	term. emulation	= PC and TSS
CRT	slow response "glass Teletype"	fast response, screen oriented	= PC = PC
cost	fixed, can go to lowest$/terminal	lowest entry	f(no. of PCs)
secure	shared, public access	totally private	contained/TSS
pros	explicit costs shared programs big jobs	low entry cost "owned" by indiv. security SW publishing = low cost	ability to expand shared facilities better match to org. structure
cons	shared poor response for terminals higher entry security	limited capability, but increasing	limited proc/prog. shared facilities

heterogeneity — one system — homogeneous. — n computers — one system / Can be heterogeneous.



VAX

Computer Interconnect Products

digital

The VAX Computer Interconnect Lets Your VAX/VMS System Grow With Your Application.

Computer applications can and often do outgrow their computing resources. Upgrading to larger or architecturally different systems is both expensive and time consuming. And what happens when you've outgrown the new system? There should be a better, easier way to have the right amount of computer power at the right time for an ever-changing application environment.

Now, from Digital, there is. A VAXcluster system.

By starting with VAX hardware and VMS (Virtual Memory System) software and by adding the new VAX Computer Interconnect (CI) products, your computer system's capabilities can keep pace with your application's requirements.

If more processing power is needed, just add another VAX processor to the cluster. If more mass storage is required, just add additional disks or tape devices. If a higher level of data integrity is demanded, the VMS operating system together with the CI provides it. If a more fault-tolerant system is called for, add as much redundancy as you want. And all of your existing hardware and software investments are preserved.

VAX Computer Interconnect products give you the capability to build and expand VAXcluster systems that grow with your applications.

Highlights

- 70 Mbit-per-second bandwidth, dual-path Computer Interconnect.
- VAXcluster systems of from one to sixteen nodes.
- Shared and/or redundant mass storage facilities.
- Passive node-to-bus interconnection scheme for online cluster reconfiguration without cluster downtime.
- Intelligent, microprogram-controlled, processor-to-CI interfaces.
- All CI elements can be redundantly configured to avoid single points of system failure.
- Fully supported by the VMS operating system.

Start With VAX...Stay With VAX.

If your computer application requirements are growing, soon you'll be faced with a difficult question. How to expand your computer capabilities without a lot of expensive reprogramming and retraining?

With a Digital VAXcluster system, the answer is easy.

With a VAX/VMS system as the starting point, and by gradually adding VAX Computer Interconnect products, the growth of your computing resources keeps parallel pace with your needs. Of course, if your application is already large or has particular requirements that can be met by a VAXcluster, your starting point can be the cluster itself.

The Digital System Interconnect

VAX Computer Interconnect products represent an implementation of the Digital System Interconnect (DSI). The DSI is a new concept in computer system technology that allows the integration of VAX/VMS processors, mass storage, and communications networks into VAXcluster configurations. The DSI provides the capabilities for integrating the VAX-11/750, VAX-11/751, VAX-11/780, and VAX-11/782 processors and the HSC50 (Hierarchical Storage Controller) intelligent, mass storage file servers into a system configuration that significantly increases system reliability, availability and processing power.

The high-speed interconnect function of the DSI is implemented by means of the CI and VMS software to allow users to share computer resources among cluster nodes, to increase system performance, and to insure the availability and integrity of system data.

When system expansion is necessary, the CI provides the most efficient and cost effective means of adding CPU power and mass storage devices. The expansion is accomplished easily and incrementally as the system requirements evolve.

The DSI operations are presently implemented by VMS Versions 3.3 and 3.4 and DECnet-VAX software. Additional functionality for the DSI will be provided by future releases of the VMS operating system.

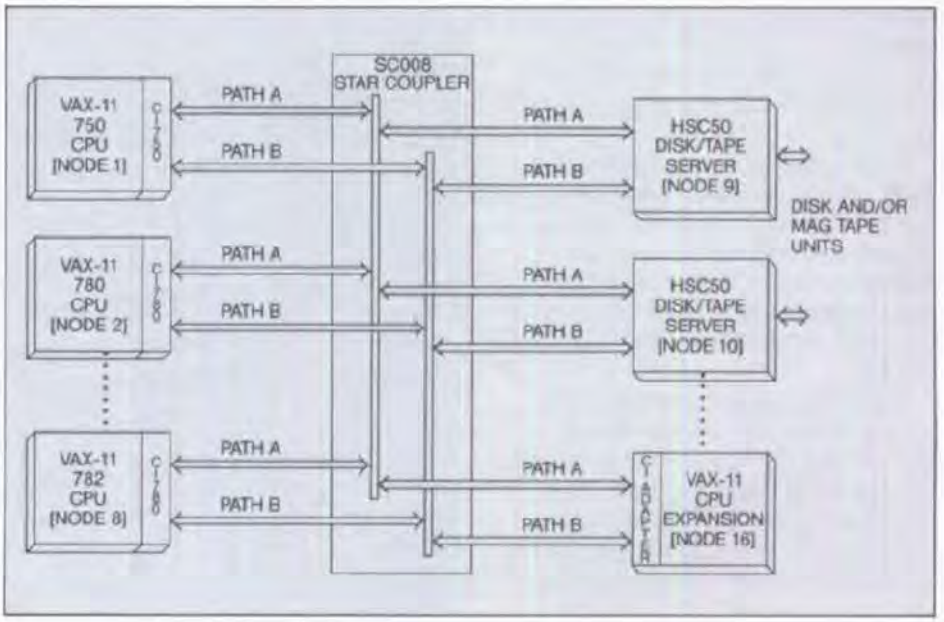
VAXcluster hardware and software support and maintenance is available at over 400 locations worldwide from Digital's expert Field Service organization.

High-Speed Computer Interconnect (CI)

The CI is a 70 Mbit-per-second bandwidth, dual path, logically multidropped link used to transfer serial data between the nodes of a VAXcluster. Up to 16 nodes can be connected to the CI and each node can be a VAX processor or an HSC50 mass storage server. Interprocessor communication is controlled by new system-level protocols. The physical connections are the CI750 and the CI780 computer interconnect adapters and the Star Coupler unit. The adapters are microprogram-controlled intelligent devices that perform the functions of fully buffered communication ports.

Messages are transferred in blocks of data between the CPU memory and other nodes within the VAXcluster by the VMS and DECnet-VAX software. By providing the necessary data buffering, address translation, and encoding and decoding, the CI750 and CI780 adapters significantly reduce the amount of overhead processing required to complete typical high-level intercomputer communication.

The Star Coupler is the common connection for all the nodes. It provides the termination and distribution points for the serial data coaxial cables to and from the individual nodes. The maximum cable length from each node to the Star Coupler is 45 meters (147.6 feet).



CI750 Adapter

The CI750 adapter is available as an option for the VAX-11/750 and VAX-11/751 computer systems. It is a microprocessor controlled, fully buffered, high-speed interface between the Computer Memory Interconnect bus (CMI) of the CPU and the dual path CI. The capability to reliably communicate over the CI in the event of failure of one of the data paths is insured with the dual CI paths.

The CI unit is mounted in a 101.6 cm (40.6 in) high cabinet. One CI750 adapter can be configured with each VAX-11/750 processor. The CI750 consists of three extended-length, hex-height modules, a backplane, and a power supply, all contained within a mounting enclosure 26.67 cm (10.5 in.) in height. A CI interface module is included and installs in an I/O Adapter slot of the CPU backplane. The interface module is connected to the CI adapter through a set of flat cables. Space is available within the CI750 cabinet for mounting an additional CI750 adapter. Although mounted in the same cabinet as the first CI750, the additional CI750 adapter is used to connect a second, separate VAX-11/750 or VAX-11/751 processor to the CI bus.

The CI components consist of the CI750 adapter, the interconnecting coaxial cables and the SC008 Star Coupler unit. Signals between the adapter and Star Coupler are transferred through four coaxial cables. The maximum length of each cable is 45 m (147.6 ft).

The CI750 cabinet attaches to the left side (preferred) or right side of VAX-11/750 CPU cabinets that comply with the latest Federal Communications Commission's (FCC) rulings.

For non-FCC-compliant systems, the CI750 unit must be mounted on the right side of the VAX-11/750 CPU cabinet. If a UNIBUS expansion cabinet is present in the existing VAX-11/750 system, the CI750 cabinet must be mounted between the CPU and UNIBUS cabinets. This configuration requires that longer UNIBUS cables be ordered because of the new position of the UNIBUS cabinet.

A minimum of 2 Mbytes of main memory is required in the VAX-11/750 CPU to support the installation and operation of the CI750 adapter.

The configuration options of the CI750 are listed as follows:

CI750-BA,-BB	CI750 adapter mounted in a 101.6 cm (40 in) height cabinet. The CI750-BA operates with 120 Vac input power and the CI750-BB operates with 240 Vac input power.
CI750-AA,-AB	An expansion CI750 adapter for mounting into the cabinet supplied with the CI750-BA, -BB option or into a customer supplied FCC-compliant cabinet for use with the VAX-11/751 CPU. The CI750-AA operates with 120 Vac input power and the CI750-AB operates with 240 Vac input power.
CI750-SA, -SB	Dual node starter kit contains two CI750-BA, -BB adapters, an SC008-AC star coupler unit, and two BNCIA-10 cables.

CI780 Adapter

The CI780 adapter is available as an option for the VAX-11/780 and VAX-11/782 systems. It is a microprocessor controlled, fully buffered, high-speed interface that connects the Synchronous Backplane Interconnect (SBI) of the CPU to the dual path CI bus. The capability to reliably communicate over the CI bus in the event of failure of one of the data paths, is insured with the dual CI paths.

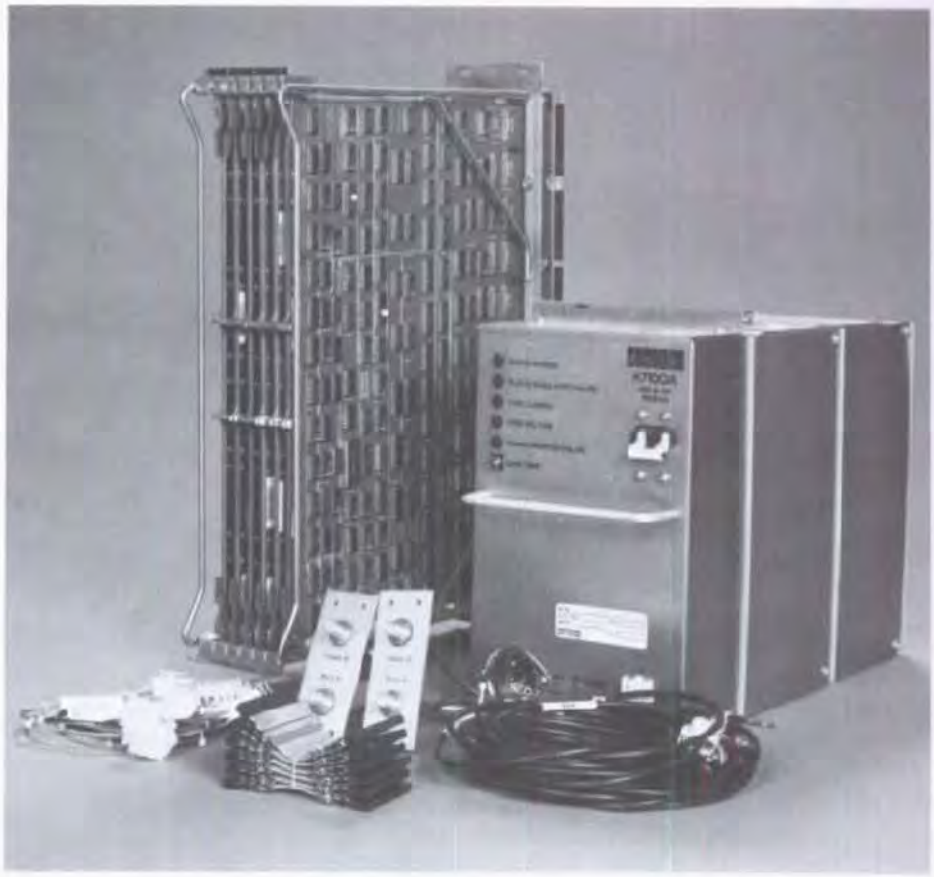
The CI780 consists of a backplane assembly that contains four hex-height, extended-length modules, a cable set and connector panel, and a power supply. The adapter can be mounted in any option panel space available within the CPU cabinet or the CPU expansion cabinet. Coaxial cables from the CI780 adapter are terminated at coaxial connector panels that are installed into the slots of the I/O connector panel at the rear of the CPU or expansion cabinets.

The CI components consist of the CI780 adapter, the interconnecting coaxial cables, and the SC008 Star Coupler unit.

A minimum of 2 Mbytes of main memory is required in the VAX-11/780 CPU to support the installation and operation of the CI780 adapter.

Signals to and from the CI780 adapter and Star Coupler are transferred through four coaxial cables. The maximum length of each cable is 45 m (147.6 ft).

The configuration of the CI780 is listed as follows:



CI780-AA, AB	CI780 adapter option including an H7100 power supply. The CI780-AA operates with 120 Vac input power operation and the CI780-AB operates with 240 Vac input power.
CI780-SA, SB	Dual node starter kit consisting of two CI780-AA, -AB adapters, one SC008-AC Star Coupler, and two BNCIA-10 coaxial cables.

SC008 Star Coupler

The SC008 Star Coupler serves as the common, central connecting point for all elements on the CI. The Star Coupler is an RF-transformer-coupled, dual-path, passive device that connects up to 16 CI nodes. The unit distributes the signals and provides signal isolation between paths and noise rejection.

The Star Coupler occupies a 101.6 cm (40.0 in) high, free-standing cabinet that has mounting space for up to four star-coupler connector panels. Each panel contains eight transmit and eight receive coaxial connectors, and six modularity coaxial connectors. The modularity connectors are used to connect the panels together for node expansion. The Star Coupler is supplied in an eight node, dual-path configuration (SC008-AC) with two connector panels. A Star Coupler expander option (SC008-AD) is available and consists of two additional panels for mounting within the cabinet. This option increases the capability to 16 nodes.

The transmit and receive cables enter the cabinet at the rear and are routed through the cable guides to the connector panels. Provisions are made for the cables to enter at either the top rear or bottom rear of the cabinet. A rear door and cable shield protect the installed cables.

The configurations of the SC008 options are described below and shown in the diagram on the following page.

SC008-AC	One A panel and one B panel. Eight node Star Coupler option with a dual-path connector panel in a 101.6 cm (40.0 in) high cabinet.
SC008-AD	Eight to 16 node dual-path expansion kit for mounting in the SC008-AC unit.
SC008-AC and SC008-AD (combined)	Two A panels and two B panels for up to 16 node dual-paths.

VMS Software Support

The VAXcluster functionality provided by the VMS Version 3.3 software allows any combination of VAX-11/780 processors, VAX-11/782 processors, and HSC50 mass storage servers to operate using the CI with shared disk support only. Each VAX-11/780 or VAX-11/782 on the CI must have its own system disk, connected either locally or to the HSC50, although no local mass storage is required. For VMS distribution purposes, there must either be an RA60 disk present or an appropriate local tape or disk drive. Up

to 16 nodes can be included, however only one CI780 adapter can be attached to each processor node.

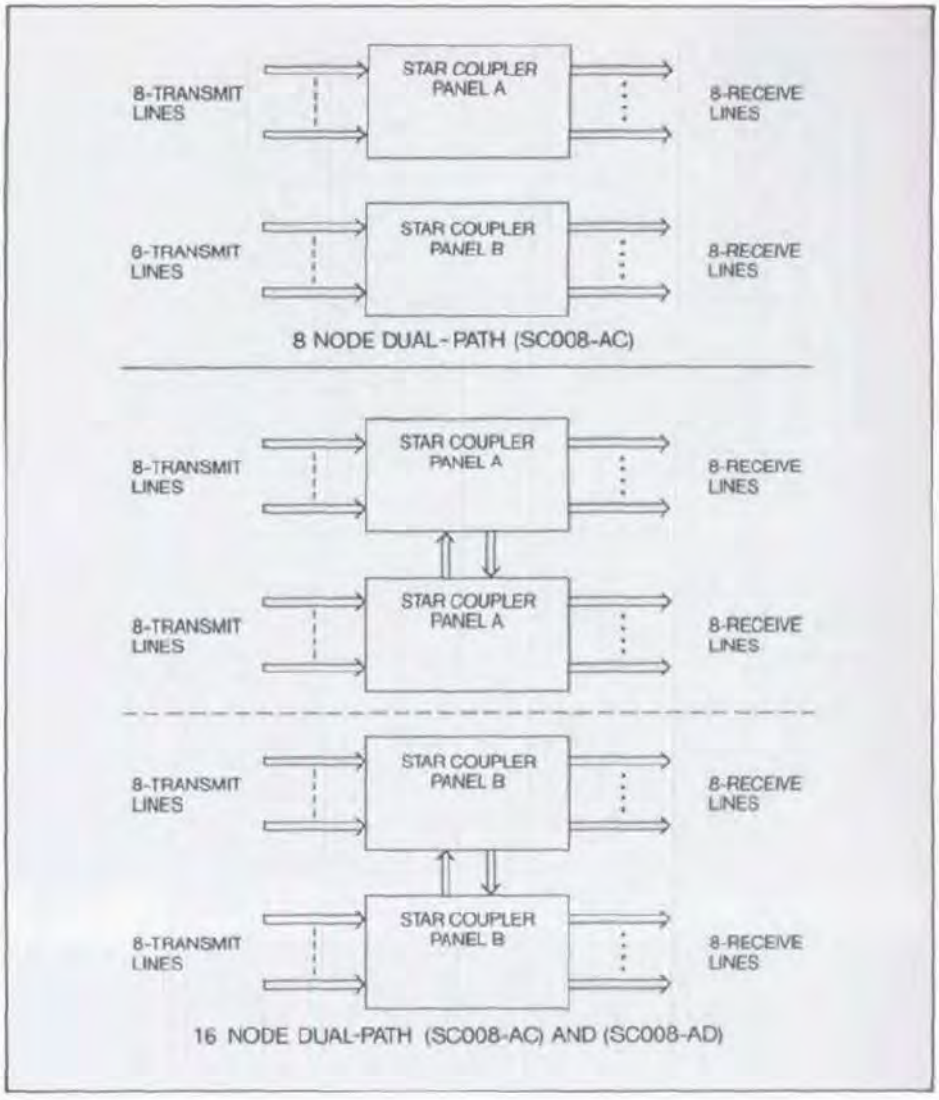
The dual-porting of the disk drives between two HSC50 file servers is possible, however the active path from the HSC50 to the disk drive must be specified when the disk media is installed. Limited sharing of disk volumes between processor nodes is also possible, however only one processor can be assigned to write to the disk and the remaining processors can only read from the disk volume. New files and the extensions of old files are not accessible to the read-only processors until the directory caches in the HSC50 have been written to the disk volume. VMS file system caching has to be disabled for the processors that are reading and writing on a specific disk volume to insure that "old-data" is not passed.

VMS version 3.3 does not permit volume shadowing on the HSC50 disk drives. Limited diagnostic capability of the HSC50 from the host processor is possible in a system environment.

DECnet-VAX software utilizes the CI and has comparable performance to a DMR11 high-speed, synchronous link. If a DECnet-VAX license has been purchased, the local disk drives on remote VAX processors can be accessed over the CI. Communication between a VAXcluster node and remote network nodes is also possible by the DECnet-VAX interface.

The VAXcluster functionality provided by the VMS Version 3.4 software allows the VAX-11/750 processors to operate with the VAX-11/780 and VAX-11/782 processors on the CI. A local disk drive is required on the VAX-11/750 processor for booting the VMS software. For VMS distribution purposes, there must be either an RA60 disk present on the HSC50 or an appropriate local disk or tape drive.

The first implementation of the VAXcluster Architecture is based on the Computer Interconnect which supports up to 16 nodes. Each node may be either a VAX-11/750, VAX-11/751, VAX-11/780, VAX-11/782 processor, or an HSC50 controller.



Initially Digital is offering VAXcluster configurations in which each HSC50 controller can provide storage access for as many as four processors. Later, we will offer VAXclusters where each HSC50 will provide storage access to additional processors. Future releases of the VMS software will also add further enhancements to the functionality of VAXcluster systems.

Field Service Support

Digital Customer Services provides full hardware and software support for VAXcluster systems. On-site service includes written agreements to respond to service needs within a specified period of time. This service can be tailored to meet needs with coverage up to 24 hours per day, seven days per week.

Remote diagnostic service is also available and provides fast and accurate diagnosis from one of Digital's Remote Diagnostic Centers.

Basic service agreements include next-day service, up to eight hours per day, five days per week.

For customers with a need for high availability and who commit to certain site and system requirements, Digital provides a guarantee of 96%, 98% or 99% system uptime, depending upon the agreement.

Specifications

GENERAL

CI750/CI780 Adapters

Data Format	Manchester-encoded serial data packets
Data Transfer Rate	70 Mbits per second maximum
Data Throughput	CI750—up to 2.4 Mbytes per second sustained. CI780—up to 3.0 Mbytes per second sustained.
Modes	Uninitialized Uninitialized/Maintenance Disabled Disabled/Maintenance Enabled Enabled/Maintenance
Priority Levels	
CI750	ARB 1,2,3 (CMI) BR4
CI780	TR14 (SBI) BR4
CI Cluster Cabling	Dual-path, Half-duplex BNCIA double shielded coaxial cables, 45 meters (147.6 feet) maximum

POWER REQUIREMENTS

CI750-BA	ac Power—120V, 60Hz (nominal) Line voltage—90-128 Vrms Frequency—47-63 Hz Phases—single Steady-state current—7.5A at 120 Vac Plug type—L5-30P Cable length—3m (9.84 ft)
CI750-BB	ac Power—240V, 50 Hz (nominal) Line voltage—180-256 Vrms Frequency—47-63Hz Phases—single Steady-state current—4.2A at 240Vac Plug type—(varies by country) Cable length—3m (9.84 ft)
CI750 (CCI) Module	5 Vdc at 11.0A
CI780-AA	ac Power—120 V, 60 Hz (nominal) Line voltage—90-128 Vrms Frequency—47-63 Hz Phases—single
CI780-AB	ac Power—240V, 50 Hz (nominal) Line voltage—180-256 Vrms Frequency—47-63 Hz Phases—single

OPERATING ENVIRONMENT

Operating Temperature	10° C to 40° C (50° F to 104° F) with a temperature gradient of 20° C/hour (36° F/hour)
Operating Relative Humidity	10% to 90% with a wet bulb temperature of 28° C (82°F), and a minimum dew point of 2° C (36 F)
Operating Altitude	Sea level to 2400 meters (8000 ft) Derate the maximum allowable operating temperature by 1.8 C/1000 meters (1°F/1000 feet) for operation above sea level

SC008 STAR COUPLER UNIT

Electrical Characteristics

I/O Connectors	Female TNC coaxial
Impedance	50 ohms
Bandwidth	1 to 100 Mhz
Isolation between I/O ports	30 db
Insertion loss (input to output)	20 db (input to output)
Nodes	
SC008-AC	Eight nodes, dual-path
SC008-AD	Provides eight additional nodes, dual-path, for SC008-AC

Physical Characteristics

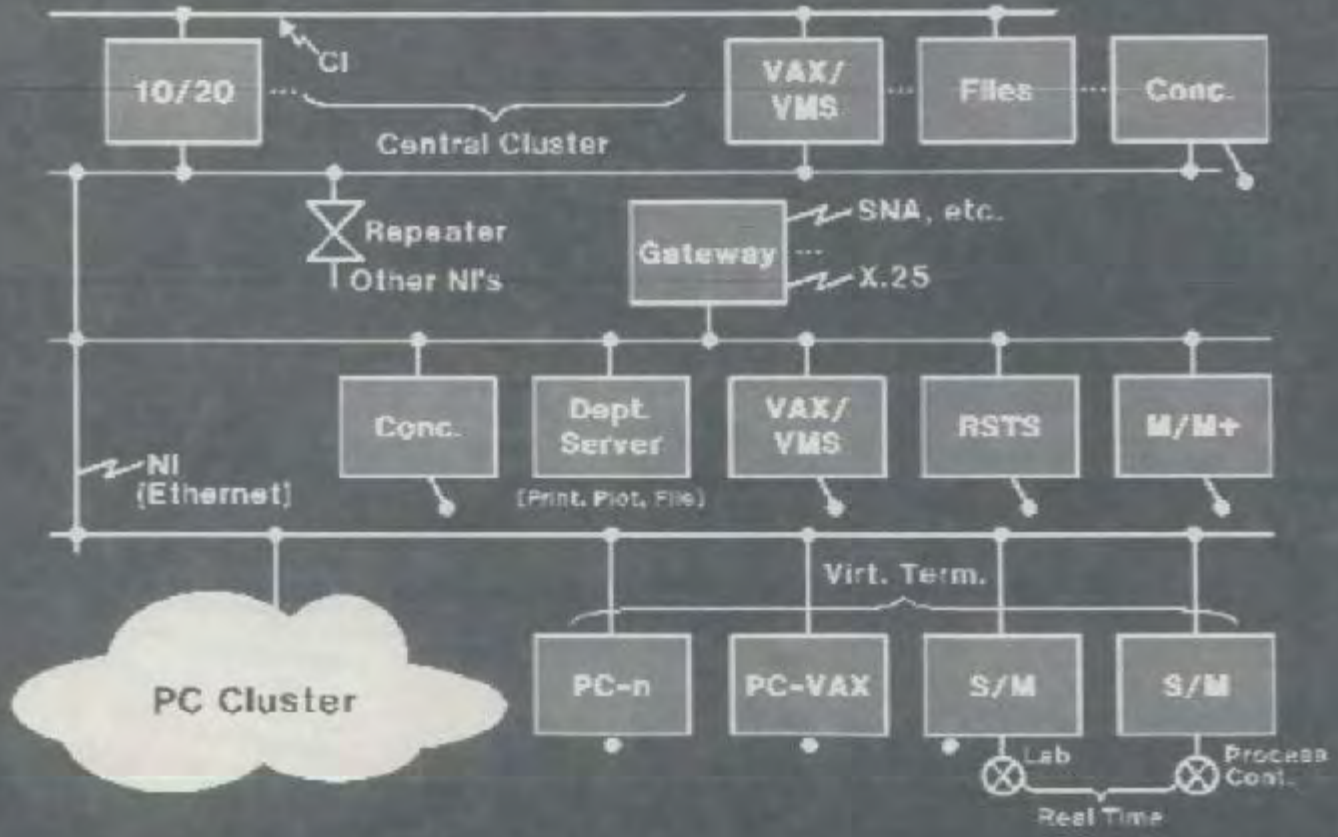
Cabinet size:	
Height:	40.0 in (101.6 cm)
Width:	21.25 in (54 cm)
Depth:	30.0 in (76.2 cm)
Connector panel size:	
Height:	8.75 in (22.2 cm)
Width:	16 in (40.64 cm)
Depth:	4.0 in (10.2 cm)
Weight:	9.6 lbs (4.3 kg)

Environmental (MIL STD 202E)

Operating and storage temperature	-55° to 100° C (-67° F to 212° F)
Operating humidity	95% relative

PRINTED IN USA EA 24790 18/83 03 08 50.0

The Product Strategy



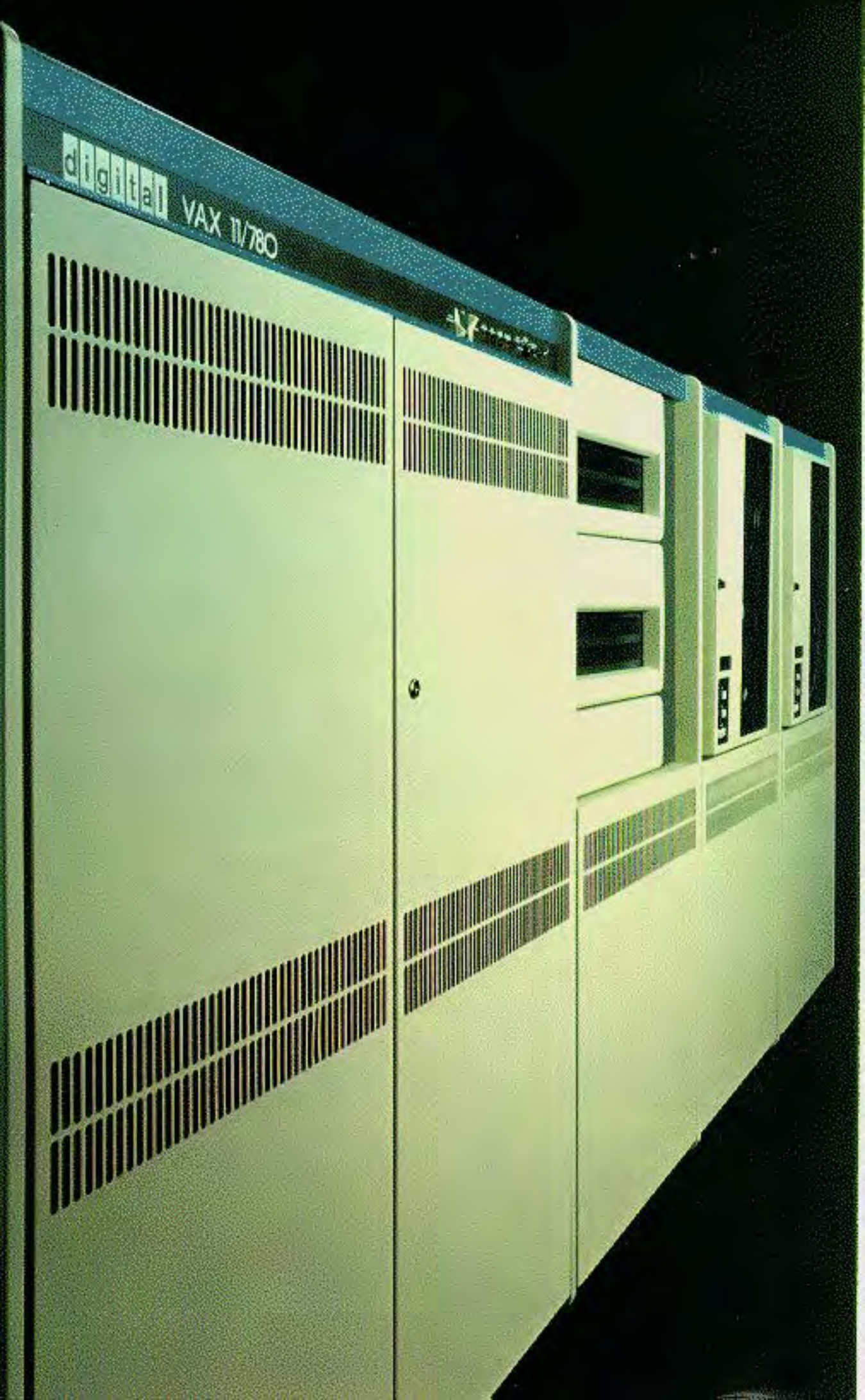

digital VAX 11/780


digital VAX 11/780


COMPUTER
HISTORY
MUSEUM

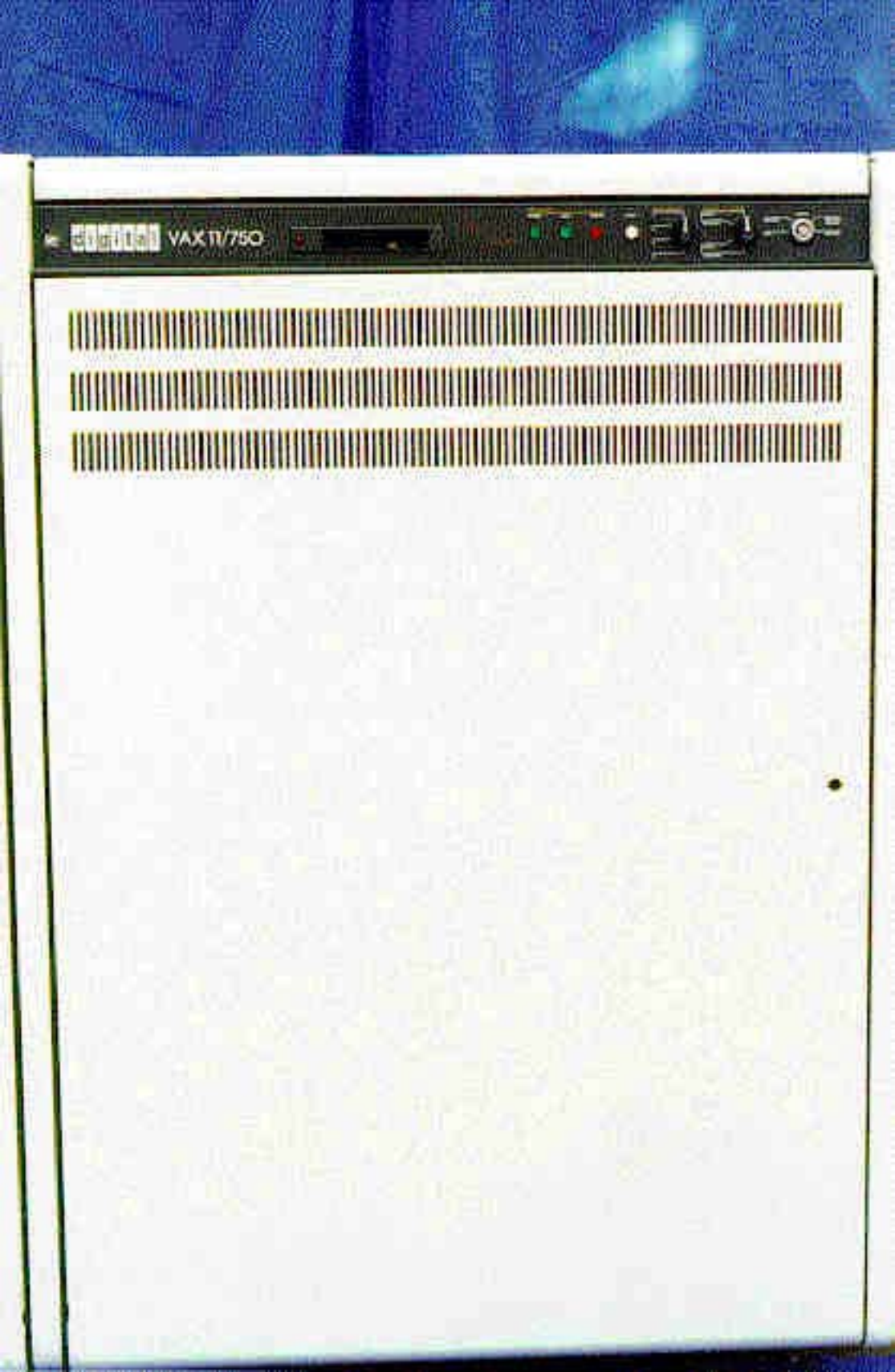

VAX 11/780
VAX BAR


digital VAX 11/780


digital VAX 11/780

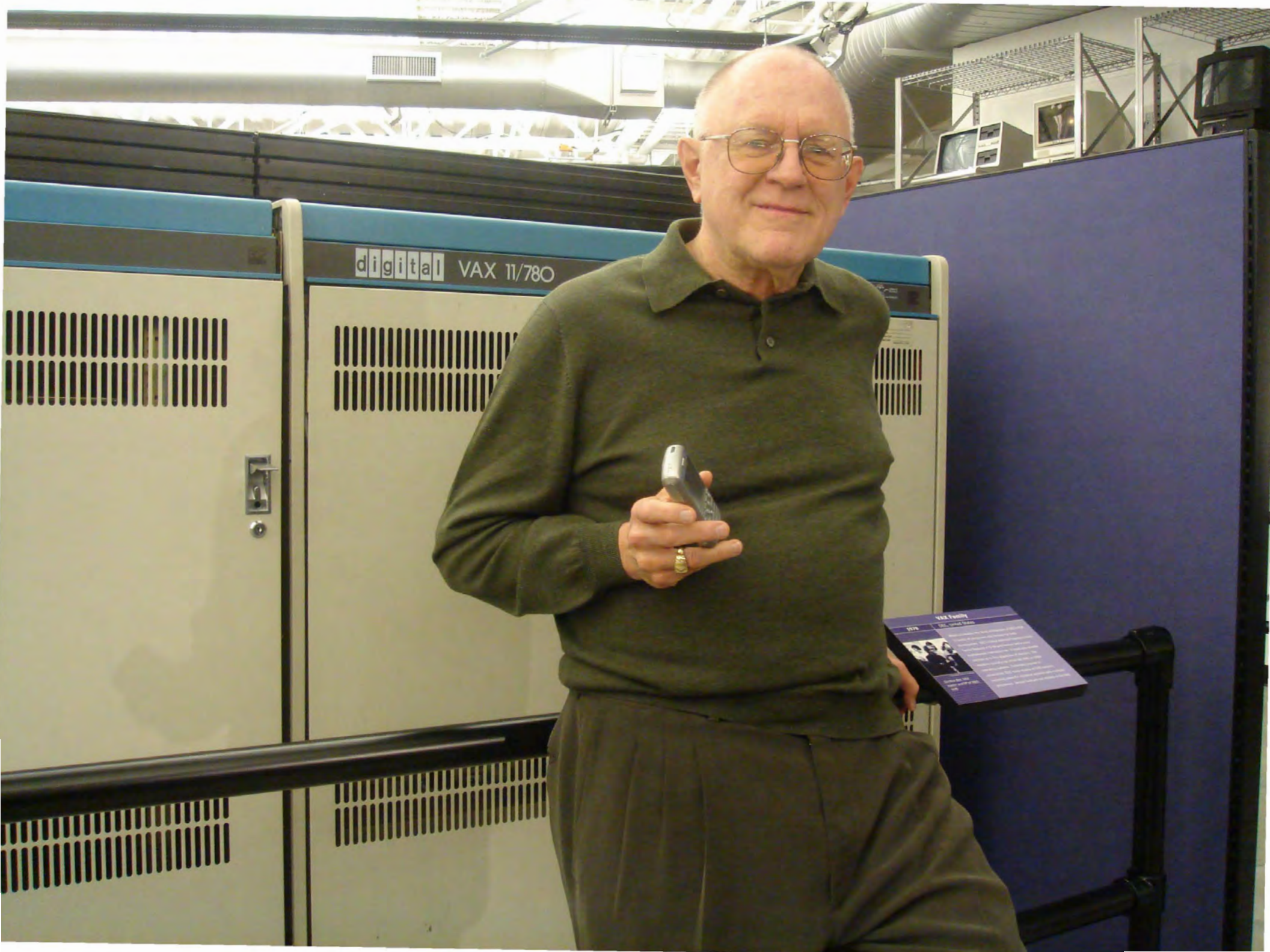

digital VAX 11/780

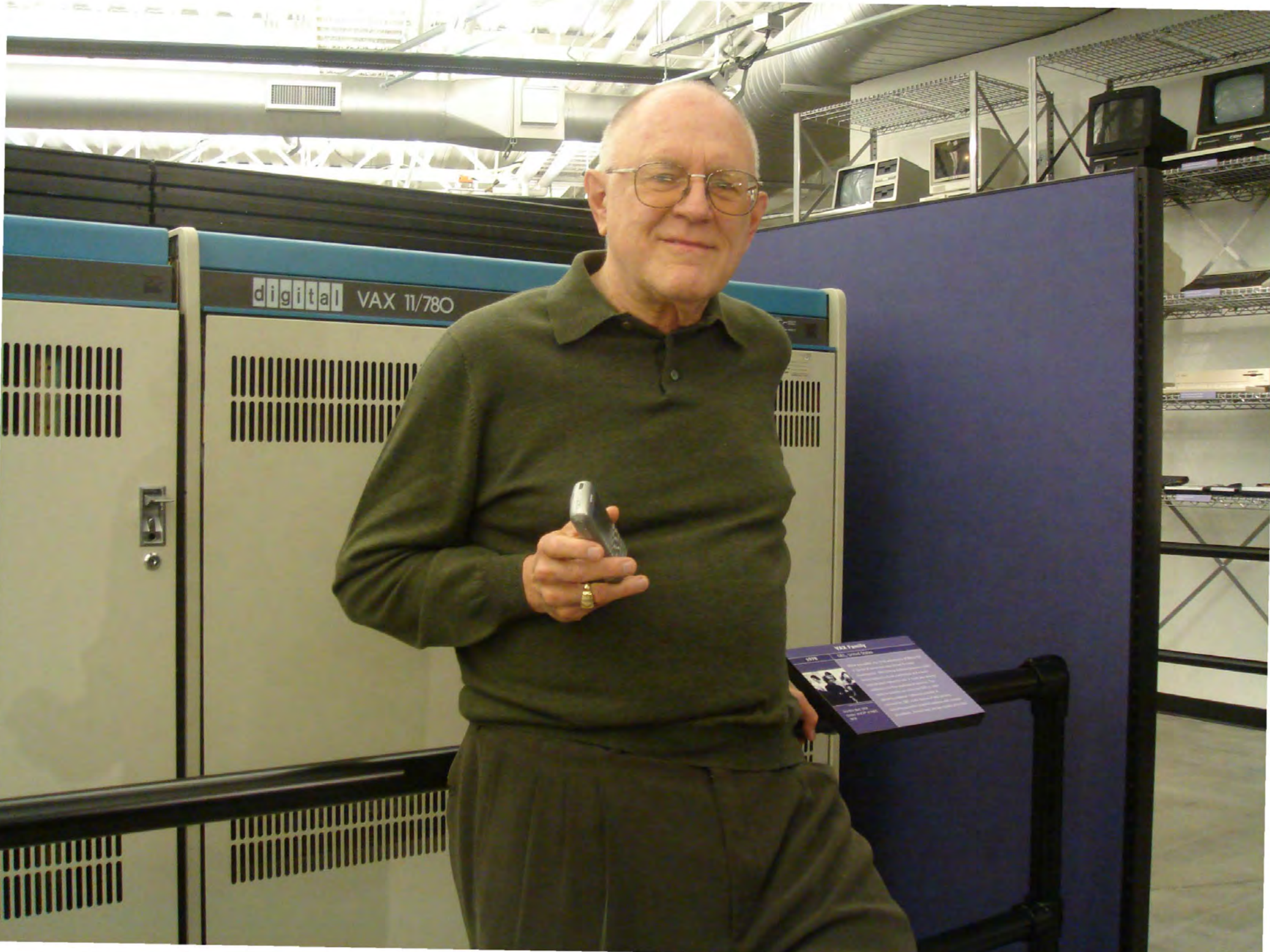

VAX 11/780


VAX 11/750


digital VAX 11/750


Micro VAX 1tm
Concept to Tapeout
Silicon Compilers Inc.


digital VAX 11/780


digital VAX 11/780

Gordon Bell

When Chester Gordon Bell was recovering from a near-fatal heart attack that occurred during a 1983 vacation in Snowmass, Colo., his doctor constantly monitored Bell's neurological functions. "Who is the president of the U.S.?" the doctor queried Bell after the brilliant computer engineer emerged from a day-long coma. Bell, so apolitical that he had never even voted, replied, "I don't remember, and it really doesn't matter."

To those who knew him, this signaled that Bell's brain was indeed intact. Bell's brain also told him that it was time to leave Digital Equipment Corp., where he had guided the creation of virtually every important computer system the company sold, including the best-selling PDP-11 and VAX series.

In the nine years since his departure from DEC, Bell has not slowed his pace; he has simply aimed a shotgun blast of energy and talent at the industry, spending time on various development projects and corporate boards and in advisory roles.

MY FOCUS HAS always been on products, but now it's more broad than it's ever been before. I like to get down in the details. You can only contribute to things if you really understand the technology and what all the constraints are.

When I consider my greatest accomplishments, certainly the VAX and then the VAX environment are at the top of the list. To me, the importance of VAX was the overall vision. IBM's computers all sat in a glass room. In the VAX environment, we were putting these computers everywhere, fully distributing them using Ethernet and all the DEC networking.

I don't mind being linked to the VAX. It's the most important thing I've done in that it touched more people than anything else. Given what I'm doing now, I'm unlikely to have anything else that far-reaching.

On the other hand, there are many accomplishments that rate highly in my career.

I set up the computing directorate at the National Science Foundation and co-authored the High Performance Computing and Communications Initiative. I was a founder of the Computer Museum, which is likely to outlive all the organizations I've worked with. [My wife] Gwen made that work.

I am currently working with Microsoft on several projects that are likely to be as important as VAX, and I've been involved in the formation and growth of a number of start-ups such as Mips, Wavetracer, Wolfsort and Chronologic. I will always be measured against VAX, however. People say, "You did VAX. Now what are you doing?"

In a funny way, I have always been my own harshest critic. It's become a matter of adjusting my level of expectations of what I should do and understanding what the trade-offs are. Do I want to give up any of these things I enjoy to try to get that second big hit?

I've also had the opportunity to mentor and support a long list of creative people such as Henry Burkhardt, founder of Kendall Square Research, Dick Clayton at Thinking Machines, Dave Cutler at Microsoft, Dave Nelson at Fluent Machines, Jeff Kalb at Maspar. I respect really bright people. That is one of my flaws. I have often bought a sales story from someone who is very bright without understanding his flaws.

I see Dave Cutler, the man who created VMS, every time I go to Seattle. He is working on Microsoft NT, which I think is going to be very far-reaching. It's going to grab the rug out from under Unix. I'm head of Microsoft's technical advisory board and consulting with them on these two key products.

I loved managing engineering at Digital, which is one thing I rarely get any credit for.

One of the things I'm happiest about now is the Gordon Bell Prize for Parallelism that I give each year. It's my personal gift of $3,000 to $5,000 a year to people who get the most out of large computers.

I was out at Los Alamos at a dinner. One of the guys who won the first prize came up to me all excited and said, "You've totally changed my life. Nothing like that ever happened. Winning that prize just totally changed our project." That felt really good.

My father was probably my greatest influence. He had an appliance store and a contracting business and did repair work. I was working as an electrician from the time I was about five or six. He retired when I went to MIT. He was a mentor and all that. I learned intuitively about handling people and customers. My mother was a school teacher — intellectual, inquisitive and, at 91, is very active mentally today. Both parents were straightforward, positive, nonjudgmental and good teachers.

I, on the other hand, can be very judgmental. My view of the industry is a good example. The thing that 99% of the computer industry doesn't understand yet is that technology is destroying the industry. In 10 years, you'll see 99% of the hardware and software systems sold through what are fundamentally retail stores.

Then there's the intermediate job, which for DEC, IBM, Unisys and HP is being systems integrators. We've got all this stuff coming out; now how do we put it all together? I don't see that as a long-lived phenomenon because the world can't stand that much adhocracy in computers.

Twenty-five years from now ... the computer disappears. Computers will be exactly like telephones. They are probably going to be communicating all the time so that no matter where I am, they are going to be attached to the network. I would hope by the year 2000 there is this big [networking] infrastructure, giving us arbitrary bandwidth on a pay-as-you-go basis.

I tend to be optimistic. So what I think of as happening in 10 years, I automatically double it. In projecting, I'm usually off by a factor of two. Somebody once said, "He's never wrong about the future, but he does tend to be wrong about how long it takes."

Interview by Glenn Rifkin, a freelance writer based in Sudbury, Mass.

Photography by Alan Levenson


C·A·L·I·F·O·R·N·I·A
VAX
SAN FRANCISCO

IPM forces have succeeded in securing part of HEC's mill...


VENUS
The VAX Mainframe
digital


VAX11
780
". . . most significant
interactive computer
of the last decade."
780
". . . most significant
interactive computer
of the last decade."
Ken Olsen

Magazine Layout

I am SUVAX . . .

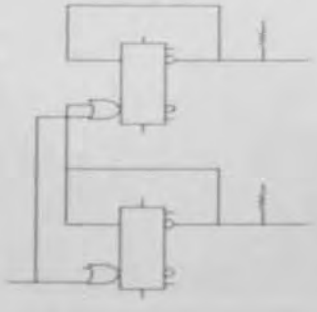
Fund me! Fund me! Fund me! Fund me! Fund me!
Fund me! Fund me! Fund me! Fund me! Fund me!
Fund me! Fund me! Fund me! Fund me! Fund me!
Fund me! Fund me! Fund me! Fund me! Fund me!
Fund me! Fund me! Fund me! Fund me! Fund me!

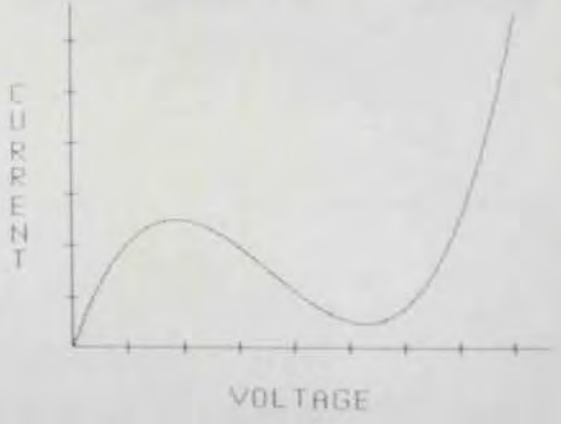
Fund me! Fund me! Fund me! Fund me! Fund me!
Fund me! Fund me! Fund me! Fund me! Fund me!
Fund me! Fund me! Fund me! Fund me! Fund me!

Fund me! Fund me! Fund me! Fund me! Fund me!
Fund me! Fund me! Fund me! Fund me! Fund me!
Fund me! Fund me! Fund me! Fund me! Fund me!
Fund me! Fund me! Fund me! Fund me! Fund me!
Fund me! Fund me! Fund me! Fund me! Fund me!

$$\frac{1}{2\pi}\int_{-\infty}^{\sqrt{2}}\left(\sum_{k=1}^{n}\sin^2 x_k(t)\right)dt$$

Fund me! Fund me! Fund me! Fund me! Fund me!
Fund me! Fund me! Fund me! Fund me! Fund me!
Fund me! Fund me! Fund me! Fund me! Fund me!





Fund me! Fund me! Fund me! Fund me! Fund me!
Fund me! Fund me! Fund me! Fund me! Fund me!
Fund me! Fund me! Fund me! Fund me! Fund me!
Fund me! Fund me! Fund me! Fund me! Fund me!
Fund me! Fund me! Fund me! Fund me! Fund me!

The VAX is one of the most successful computer-families of Digital Equipment Corporation (and perhaps the whole industry). The name VAX, which stands for "Virtual Address eXtension" refers to the large virtual address space, which was meant to be an extension to the PDP-11 architecture. In the first months of the project, the machine was called PDP-11/780. The hardware and the main operating system for it, VMS (Virtual Memory System) were developed parallelly.

The beginning...

Many thanks to Steve Rothman!

Digital started to think about a possible wide-word machine in 1973, but the early decision was to build a new 36-bit machine based on the successfull PDP-10. This new machine was to superceed the high-end of the PDP-11 line, so the strongest PDP-11, the 11/70 was developed as an interim machine until the new 36-bitter would be ready. One year later however a few people (Bill Demmer was among them) questioned whether this was the right way to go. They felt that Digital needed a machine that would made migration from the PDP-11 easy. Following that, the decision was made to consider an "extended PDP-11" alternative to the 36 bit machine. The 36-bit project, led by Len Hughes (engineering and Dave Rodgers (lead architect/project engineer/engineering supervisor) got the nickname "Unicorn", while the 32-bit machine was later called "Star". The latter took off on April 1st, 1975 (this is just an anecdote, it was a few days earlier).

The "extended PDP-11" project was led by Gordon C. Bell, and one of the main goals was PDP-11 compatibilty. The team consisting of five people (Gordon Bell, Bill Demmer, Richie Lary, Steve Rothman and Bill Strecker) created a proposed architecture, which was to be reviewed in June, 1975. The team (called VAX-A team) worked through April and May on the third floor of building 12 in the Maynard mill; the result was the so-called "VAX Blue Book" - the specification of a 32-bit machine, distributed in loose blue binders. The architecture we now know as VAX is a subset of what was originally planned: many features were dropped because they weren't economically realisable. There were two other commitees too besides team A: VAX-B and VAX-C; the first consisted of technology reviewers, the second worked on business issues. The name "VAX" was coined probably by Gordon Bell.

Pics courtesy of:

After the meeting in June, the "Unicorn" project got cancelled (that's one of the reasons why 36-bit fans hate VAXen; and it only got worse later in the mid-eighties, when the whole PDP-10/DECsystem-10/DECSYSTEM-20 line was "killed" in favor of the VAX), and the "32-bit engineers" went on implementing their architecture, with some of the engineers of "Unicorn". When the architecture work on paper seemed to be near its end, a few members of the team went on to build the VAXHS, the VAX Hardware Simulator: a modified PDP-11 for developing VMS and the FORTRAN compiler (four of these machines were made). After this was finished, the development group took off to design the second VAX computer ("Comet").

The VAX Timeline

Disclaimer: the pages don't focus on figures, numbers, and such. This is a history page, not the summarized edition of all technical manuals ;-)

1977. Introduction of the **VAX-11/780** "supermini" computer

1978. VMS1.0 shipped

1980. The second generation: **11/750**, the first LSI VAX

1981. VAX Information Architecture

1982.VAX-11/730, the "mini-VAX", **ALL-IN-1** (integrated office softwaresystem); **RA60** and **RA81** hard disk drives

1983. Computer Interconnect (CI) "clustering" technique for interconnecting VAX processors giving greater computing power; the **11/725; ULTRIX, VAXELN**

1984. VAX-11/785, 8600, MicroVAX I, VAXstation I: the first microprocessor VAX-implementation; **ULTRIX-32; Rdb/VMS** and **Rdb/ELN**

1985. MicroVAX II, MicroVAX II/GPX; VMS 4.2; VAX11 ACMS (transaction-processing)

1986. VAX 8800; 8200, 8300; VMS 4.5; LAVC (local area VAX-cluster)

1987. MicroVAX/VAXstation 2000: low-cost micro-workstation, **MicroVAX 3500, 3600**; **VAX 8874, 8878**: pre-packaged high-end clusters

1988. VAX 6200: the dawn of a new era; **VMS 5.0; DSSI, Dual-Hosting;**

Network Application Support

1989. **MicroVAX 3100**; **3800, 3900;VAX 6000-300**; **6000-400**; **9000** (the last non-microprocessor based Digital machine), **DECwindows**

1990. **MicroVAX 3300, 3400**; **VAX4000**; **VAX 6000-500**; **VAXft** (Fault Tolerancy), **OpenVMS**

1991. **VAX 6000-600**

1992. **VAX 7000, 10000**, the last of the "big" VAXen

About these pages, links, etc...

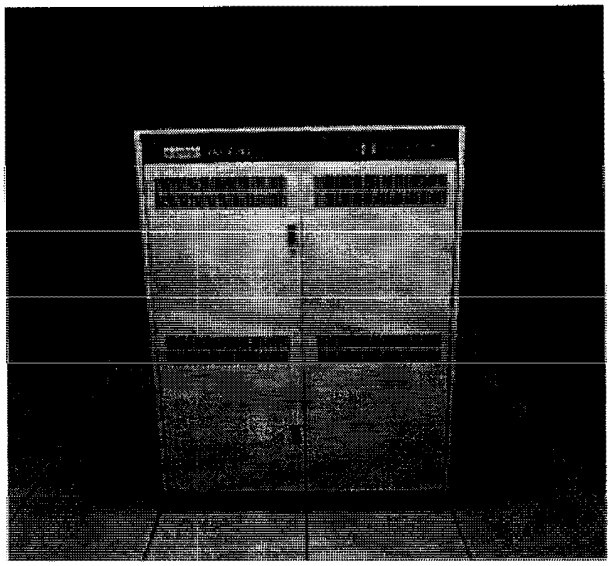
All comments are welcomed!

Hamster

1977

VAX 11-780

"Star"



The VAX11-780 was a one MIPS (one million instruction per second) machine that became an industry standard: DEC dubbed its performance one VUP for VAX Unit of Performance; but "standard" benchmarks as the SpecMark89, SpecInt92, SpecFp92 are also based on the 780.
The processor unit depicted to the left contained the SBI (synchronous backplane interconnect; said to be borrowed from a never-finished 36-bit machine, the "Unicorn"), CPU (KA780), FPP (FP780), memory, UNIBUS adapters (DW780), power supply and the console, an LSI-11 (PDP-11/03) processor. A usual configuration contained 4MB of ECC RAM. A commercial "hit", and a very reliable machine too: a few of them are actually in use even this day (2000), and they were "cloned" too!

780 opened up
A nice brochure photo
A tipical configuration
Oldies but goldies...
A closer look
Full house
Machine house (but you can't hear the noise)

Others

PDP-11/60: An It-could-have-been-a-VAX PDP-11

1978

VAX/VMS 1.0

The operating system designed specially for the VAX family was based on the conception of 4 GB's of virtual memory, a four levels of processor mode (Kernel, Executive, Supervisor, User), and one of the most significant PDP-11 operating system: RSX-11 (the utilities came mostly from the latter; whe first version of VMS to feature only native code was 3.0).

The development of the first version (V1), codenamed "Starlet" started in 1975, led by Roger Gourd, assisted by Dave Cutler, Dick Husvedt and Peter Lipman. The first machine that the code ran on was a PDP-11 with modified microcode, called the "VAX Hardware Simulator", then they changed to the first VAX-11/780 prototype. One of the main feature of the OS/hardware was the ability to run PDP-11 code, the next goal was to maintain compatibility with newer OS releases and hardware models: the newest release of VMS still runs on the oldest hardware...

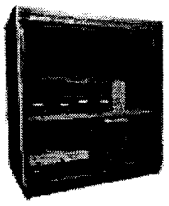
Main features of V1: 64 MB maximal physical memory, multi-user multitasking, DCL (Digital Command Language) shell, DECnet-support, FORTRAN IV compiler. The first VAX11/780-VMS configuration was shipped in 1978, the customer was the Carniege-Mellon University.

Misc

VT100: Digital's new CRT video terminal complied with ANSI regulations, and immediately became a de-facto standard. Main featrues: 24x80 or 14x132 characters, keyboard with 83 keys, support for special escape characters. It is interesting to note, that the vt100 features found in the termcap library of UNIX systems is based on the VT102, which was an enhanced version of the terminal.

DECsystem-2020: Digital's last 36-bit computer, a small-sized mainframe.

Digital VAX 11/780 (1978)



The first of Digital's famous VAX "supermini" family, the 11/780 became an industry standard for performance benchmarks -- a 1 MIPS machine. Several factors came together to make the VAX one of the most successful computers of all time, and its architecture one of the most pervasive.

Digital's strong marketing organization played a key role; and the company's huge installed base of PDP-11s was leveraged because the 32-bit VAX was software compatible with those 16-bit machines.

Strong operating system software also made the 11/780 a contender for many applications that until then had been the exclusive territory of mainframes. Unix, which was developed on a PDP-11, was one. The other was Digital's own VMS, developed by David Cutler, one of the most robust supermini operating systems of its day.

The 11/780 was part of a larger "VAX strategy" spearheaded by Gordon Bell, to install VAXes throughout a business, connected by a network. This strategy was worked on through 1985 and resulted in workstations (microVAX), minis and VAX clusters, all connected via Ethernet.

Price	$200,000 (base)
Units Shipped	100,000 by 1987
Technologies	32-bit architecture, Shottky TTL, 1 MIPS performance
Software	VMS, Unix
Inventor	Gordon Bell; implemented by Bill Strecker, William Demmer, David Rodgers & David Cutler

|Generations | Transistors | Integrated Circuits | Microprocessors | Intel Evolution | Servers |

| HOME |
Notices and General Information
Trademarks

digital

1977–1997... and beyond

Nothing Stops It!

Of all the winning attributes of the OpenVMS operating system, perhaps its key success factor is its evolutionary spirit. Some would say OpenVMS was revolutionary. But I would prefer to call it evolutionary because its transition has been peaceful and constructive.

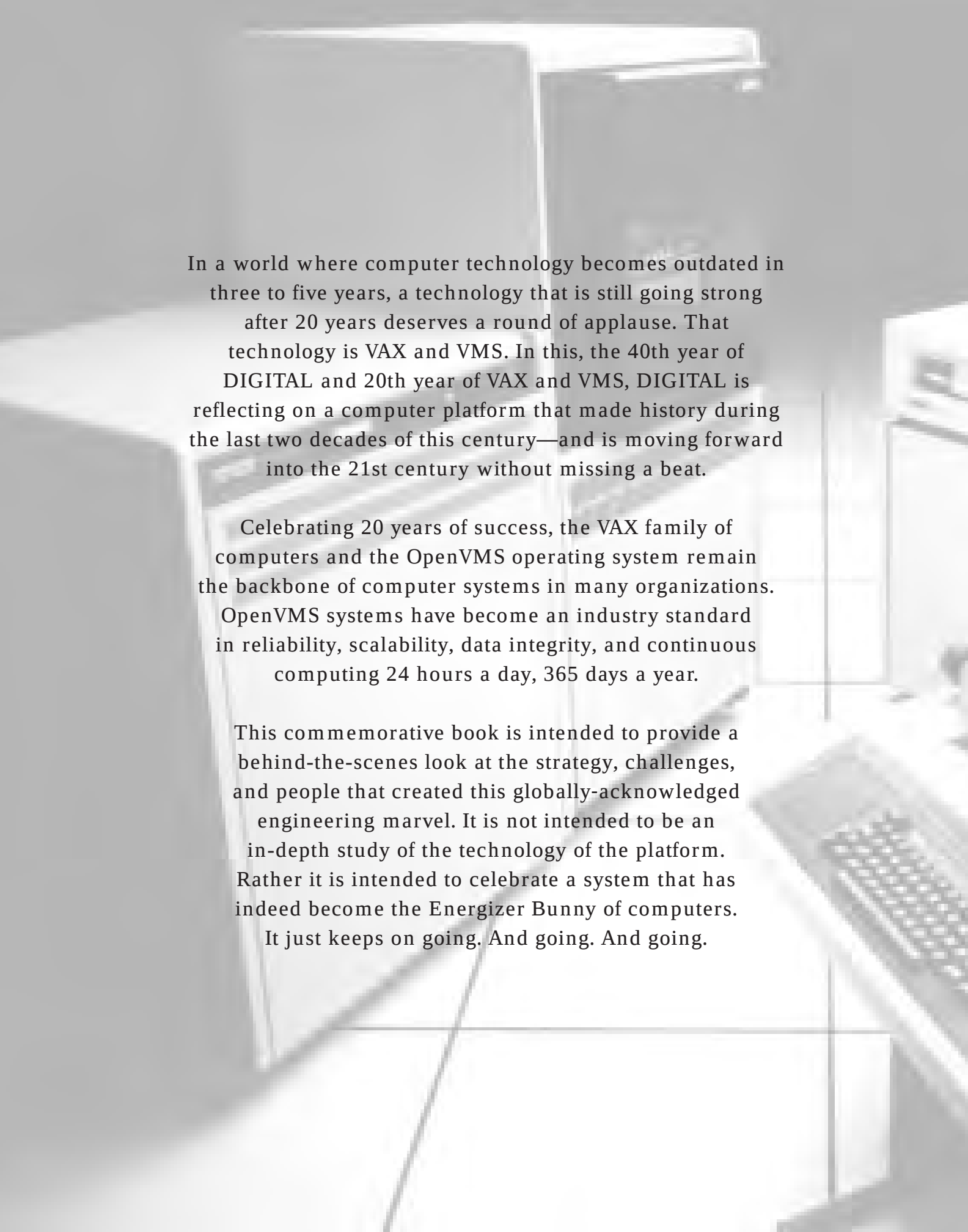
Over a 20-year period, OpenVMS has experienced evolution in five arenas. First, it evolved from a system running on some 20 printed circuit boards to a single chip. Second, it evolved from being proprietary to open. Third, it evolved from running on CISC-based VAX to RISC-based Alpha systems. Fourth, VMS evolved from being primarily a technical operating system, to a commercial operating system, to a high availability mission-critical commercial operating system. And fifth, VMS evolved from time-sharing to a workstation environment, to a client/server computing style environment.

The hardware has experienced a similar evolution. Just as the 16-bit PDP systems laid the groundwork for the VAX platform, VAX laid the groundwork for Alpha—the industry's leading 64-bit systems. While the platforms have grown and changed, the success continues.

Today, OpenVMS is the most flexible and adaptable operating system on the planet. What started out as the concept of 'Starlet' in 1975 is moving into 'Galaxy' for the 21st century. And like the universe, there is no end in sight.

—Jesse Lipcon
Vice President of UNIX and OpenVMS Systems Business Unit

Table of Contents



In a world where computer technology becomes outdated in three to five years, a technology that is still going strong after 20 years deserves a round of applause. That technology is VAX and VMS. In this, the 40th year of DIGITAL and 20th year of VAX and VMS, DIGITAL is reflecting on a computer platform that made history during the last two decades of this century—and is moving forward into the 21st century without missing a beat.

Celebrating 20 years of success, the VAX family of computers and the OpenVMS operating system remain the backbone of computer systems in many organizations. OpenVMS systems have become an industry standard in reliability, scalability, data integrity, and continuous computing 24 hours a day, 365 days a year.

This commemorative book is intended to provide a behind-the-scenes look at the strategy, challenges, and people that created this globally-acknowledged engineering marvel. It is not intended to be an in-depth study of the technology of the platform. Rather it is intended to celebrate a system that has indeed become the Energizer Bunny of computers. It just keeps on going. And going. And going.

Acknowledgements

Editors:
Jim Rainville
Karen Howard

Writer:
Kathie Peck

Design:
Kathy Nardini, Doherty Communications

Research:
Jim Rainville
Cynthia Carlson
Ed Yee
Ann Hewitt

Review Team:
Paul Bergeron
Peter Bernard
Jeff Borkowski
Mary Ellen Fortier
David Foulcher
Andy Goldstein
Mark Gorham
Jane Heaney
Jackie Jones
Karen Leonard
Steve Stebulis
Ed Yee

Photography:
Nancy Strader, Corporate Photo Library
Dick Willett, *Forefront* Magazine

Web Production:
Jim Keenan

Content Providers:
Brian Allison
Patty Anklam
Paul Beck
Gordon Bell
Richard Bishop
Tom Cafarella
Rick Casabona
Bruce Claflin
Wally Cole
Harry Copperman
Chris Christiansen
Marion Dancy
Jan Darden
Stu Davidson
Scott Davis
Bill Demmer
Sas Dervasula
Richard Doucette
Elliot Drayton
Jim Evans
Dave Fenwick
William Fischer
Mary Ellen Fortier
Sue Gault
Andy Goldstein
Roger Gourd
Clair Grant
Heather Kane
Forrest Kenney
Veli Kozkko
Jim Krycka
Kim Leavitt
Jud Leonard
Rich Lewis
Steve Lionel
Jesse Lipcon
Daryl Long
Ernie Lyford
Rich Marcello
Dan Marshall
Wes Melling
Kathy Morse
Larry Narhi
Ken Olsen
Matti Patavi

Mark Plant
Robert Porras
Jean Proulx
John Rando
Robert Ryan
Joe Scala
Terry Shannon
Maurice Steinman
Bob Stewart
Mark Stiles
Bill Strecker
Bob Supnick
Wendy Vogel
Bob Willard
Ed Yee
Steve Zalewski

CHAPTER I Changing the Face of Computing

After more than 300 man-years of intensive development, DIGITAL announced its first 32-bit computer system—the VAX-11/780 and its companion operating system, VMS—at the Annual Meeting of Shareholders on October 25, 1977. Because of its 32-bit technology, the VAX system represented a new milestone for DIGITAL and was heralded as a major breakthrough in the computer industry.

The VAX platform and VMS operating system were unveiled by the President and founder of DIGITAL, Ken Olsen. The new product was showcased with a clear plastic front so that the audience could see the CPU, cache, translation buffer, and other integral parts of the machine. The VAX system was demonstrated running a Scrabble program—astonishing spectators by winning the match against a human. The winning play was the word "sensibly," taking the 50-point, seven-letter bonus and scoring a total of 127 points.

"The best of what we've learned about interactive computers in our first 20 years has gone into this machine. We have spent more than 300 man-years of intensive engineering effort in its development, and during that time I have sensed more excitement and enthusiasm among the developers of VAX than I remember seeing at any other time in the short history of DIGITAL."

—Ken Olsen
Founder, Digital Equipment Corporation
October 25, 1977

Over the next decade, VAX and VMS products were destined to change the way people used computers—and catapulted DIGITAL into a position as one of the world's top computer manufacturers.

Setting the sights high

The VAX system was designed to meet several key objectives. First, it was based on a revolutionary 32-bit architecture. Second, it solved many of the problems associated with earlier computer technologies. Third, it was designed to be useful to the largest possible number of users in diverse markets, and to offer DIGITAL customers a seamless transition from earlier product architectures. And finally—in direct opposition to the questionable strategy of planned obsolescence—the VAX system was designed to last between 15 and 20 years. An impressive list of requirements, to be sure. But VAX and VMS met them all.

"Jesse Lipcon continually reiterated, 'Our top three goals are time to market, time to market, time to market.' At one point I said, 'Wait a minute, Jesse, what about quality?' Without missing a beat, Jesse replied, 'Quality isn't a goal, it's a given.'"

—Jay Nichols
Computer Special Systems, Manager of Engineering

Moving from 16-bit to 32-bit computing.

A total systems focus

From its inception, the VAX development program had a total system focus, encompassing groups from hardware and software engineering, support, product management, documentation, and manufacturing. More than 1,000 people in the corporation worked on the first VAX and VMS system, in some capacity, on an extremely aggressive schedule. Without question, the work produced within the limited time frame exceeded all expectations.

Hardware meets software at the drawing table

VAX and VMS made engineering history by being the first interactive computer architecture in which the hardware system and software system were designed together from the ground up. This was a novel approach to designing a computer architecture, where hardware and software teams worked jointly and altered their designs in consideration of each other's requirements. The result of this united engineering effort was a tightly integrated system that provided unprecedented reliability, flexibility, scalability, and data integrity. In short, bullet-proof computing.

The first VAX system demonstrated a major industry breakthrough by providing the functionality, capacity, and performance of a mainframe—coupled with the interactive capabilities, flexibility, and price/performance of a minicomputer.

"In our spare time, Stan Rabinowitz and I wrote a Scrabble program for the PDP-11. As soon as the VAX was available, we ported it over to the VAX. Ken insisted that we demonstrate the Scrabble program at the announcement. I ran the Scrabble program—pitting it against a human being—and the VAX demolished him. Then Ken stood up and said, 'It's time for the big league games. This isn't Tic Tac Toe, this is Scrabble!'"

—Richie Lary
Corporate Engineering Consultant

CHAPTER II Setting the Stage

The pre-VAX years

The first VAX computer was introduced as DIGITAL celebrated its 20th anniversary. The company—founded by Ken Olsen, Harlan Anderson, and Stan Olsen in 1957 with an initial capital investment of $70,000—began as a small module manufacturer in a corner of a sprawling mill complex in Maynard, Massachusetts, a small town 30 miles west of Boston.

Front cover of Fortune *magazine, October 1986. Features story on Ken Olsen and Digital Equipment Corporation.*

Moving from printed circuit modules to computers

While DIGITAL initially produced printed circuit logic modules, the company's real mission was to bring computing to the people. In its second year, DIGITAL made the transition to computers and in 1959 introduced its first computer—the PDP-1. During the 1960s, the company rolled out a family of PDP computers, each more powerful than its predecessor. Early on, innovation and engineering excellence were hallmarks that have characterized DIGITAL products throughout its entire history.

Peer-to-peer networking—the birth of distributed computing

The company's first computers were stand-alone systems. But in the early 1970s, DIGITAL pioneered peer-to-peer computer networking with the introduction of its first successful networking software product, DECnet. Networking allowed customers to connect many minicomputers and share a common database of information. This approach to computing launched the concept of distributed computing. It was a novel approach because at the time mainframes and stand-alone minicomputers were the only computing game in town.

Up to that point, computers did not talk to each other. Moving information from system to system involved using slow magtape and sneaker net. Distributed computing offered the advantage of flexibility and connectivity. Now information could be moved across computer rooms and later across the country almost instantly—making the DIGITAL goal of bringing computing power to the people who needed the information a reality. Technical customers embraced the interactive, accessible nature of these new minicomputers, and DIGITAL began to flourish with PDP systems selling in the tens of thousands.

"Gordon Bell's vision was the primary driver behind the entire VAX family. And I think its success was due to his vision."

—Bill Demmer
Former VP, Computer Systems Group

Digital Equipment Corporation headquarters in Maynard, Massachusetts, with the famous clocktower.

The beloved Mill

The physical environment in which VAX and VMS was created was very much in keeping with its New England heritage. "The Mill"—a set of brick buildings in the center of Maynard, Massachusetts—exemplifies two old New England traditions. One is the classic mill town pattern with the development of an industry and the growth of a community around it. The other is the thrifty Yankee "make do" principle—it's better to "make do" with what you have if it's still useful, rather than abandon it and buy something new and expensive.

The original mill site on the Assabet River was once part of the town of Sudbury; the opposite bank belonged to the town of Stow. The present town, incorporated in 1871, was named for the man most responsible for its development, Amory Maynard. At the age of 16, Maynard ran his own sawmill business and later went into partnership with a carpet manufacturer. They dammed the river to form a millpond to provide power for a new mill, which opened in 1847.

The clock tower

After Amory Maynard died in 1890, his son Lorenzo built the Mill's famous clock tower in memory of his father. The clock's four faces, each nine feet in diameter, are mechanically controlled by a small timer inside the tower. DIGITAL never electrified the timer nor the bell mechanism. To this day, someone has to climb the 120 steps once a week to wind the clock: 90 turns for the timer and 330 turns for the striker.

In 1899, the American Woolen Company, an industrial giant, bought the Assabet Mills and added most of the existing structures. The biggest section was Building 5, which was 610 feet long and contained more looms than any other woolen mill in the world.

Over the next 50 years, the Assabet Mills survived two world wars and the Depression. But when peace returned, the Assabet Mills were shut down entirely in 1950. Like many New England mills, it succumbed to a combination of Southern and foreign competition and the growing use of synthetic fibers.

From textiles to computers

In 1953, ten businessmen from nearby Worcester bought the mill and leased space to tenants. One of the companies attracted by the affordable space was Digital Equipment Corporation, which started operations in 8,680 square feet in the mill in 1957.

DIGITAL grew so fast that within 17 years, it bought and expanded into the whole mill complex. Inside, old paint was removed from the walls, exposing large areas of the original brickwork. Pipes were painted in bright colors, in contrast to the massive beams and columns. The large interior once used for textile machinery became filled with modular office cubicles—offering flexibility to meet the company's changing requirements. DIGITAL left the exterior of the buildings largely unaltered, but cleaned up the Assabet River, which once was colored with the residue from the mill's dyeing plant.

When the employees first moved in, the floors were wavy and soaked with lanolin from the wool processing days—which would eat right through crepe-soled shoes.

While the engineers started designing the VAX and VMS architecture, the floors were being refinished, so they had to live with the ambient music of hammering on the floors above. Then the floors were sanded and polyurethaned. During the sanding, the engineers were given plastic sheets to cover their desks and equipment each night. Before the roof was replaced, rivers of water flowed through parts of the mill when it rained.

Industry trends of the time

During the mid 1970s, industry trends included interactive computing and networking. Corporations were discovering that distributed computing was a viable alternative to the mainframe batch environment. Distributed computing allowed a company to decentralize information and put it in the hands of decision makers—an idea which dovetailed perfectly with Ken Olsen's original goal of giving computer power to the people.

PDP-11/70—moving from 16-bit to 32-bit computing.

Due to industry-wide technological advances, more computing power could be packaged in every square centimeter of space for less money. As a result, computer systems could be made smaller and more affordable, and provide functionality previously found only in large mainframes.

DIGITAL drove the industry trend of building increasingly more powerful—but less expensive and physically smaller—computer systems.

DIGITAL had developed tools to meet the opportunities created by these industry trends: the interactive minicomputer, DECnet, more powerful and easier-to-use software, a volume manufacturing capability, and financial applications. However, corporations were being constrained by the limitation in the addressing range of 16-bit computer architectures—a bottleneck the industry needed to address.

Outgrowing 16-bits

As early as 1974, DIGITAL recognized the limitations of its 16-bit PDP architecture—especially for such tasks as writing large programs and manipulating large amounts of data often needed in scientific, engineering, and business data processing applications. The company realized there was a need for a new architecture that would be compatible with PDP systems, but would have larger addressing capabilities and enough power to meet the computing needs of the future.

"I would say probably the most significant thing that DIGITAL has done is make computing available to the masses. Instead of the high priests in the white robes behind the glass walls, DIGITAL brought computing out of the glass house and made it affordable and acceptable to the mainstream."

—Terry Shannon
Publisher, Shannon Knows DEC

32-bit computing: the next logical step

Some users were also beginning to feel hampered by the limitations of 16-bit computing. They were becoming frustrated that large programs had to be broken up into smaller pieces in order to run on their computers. The 16-bit addressing was hindering progress, and finding a solution became eminent. Extending the addressing architecture to 32 bits could supply the power needed to solve this problem. Other computer companies also recognized the shortcomings of 16-bit addressing and had begun working on 32-bit systems, which seemed to be the next logical step in computer development.

Considering the alternatives

DIGITAL was intent on maintaining its lead in the minicomputer industry and knew that extending the addressing architecture was critical. The company considered many approaches to extending the addressing that built on its current line of products, the PDP family and the DECsystem-10. The company also knew that customers not only required more system power and memory, but wanted more economical systems that would be compatible with their PDP-11 family of processors, peripherals, and software.

In late 1973, DIGITAL began development of the PDP-11/70—an extension of the basic architecture of the PDP-11 family—in an attempt to solve the memory-addressing problem. The PDP-11/70 had a larger memory capacity (up to 4MB), but using it with the 16-bit PDP-11 software architecture was cumbersome. The development team had the choice of finding a way to continue to "brute force extend" the PDP-11 family architecture, or to create something new.

VAX-11/750

CHAPTER III VAX Hardware Development

"It's the addressing capability that has probably been, over history, the major driving force behind new computer architectures. It's what initially brought about the whole notion of the VAX computer."

—Bill Demmer

Former VP, Computer Systems Group

Creating a whole new architecture

In March of 1975, a small aggressive development task force was formed to propose a 32-bit PDP-11 architecture. The team included representation from marketing, systems architecture, software, and hardware.

The project included three phases. During Phase I, the team produced a document that encompassed a business plan, system structure, build plan, project evaluation criteria, the relation to longer term product development, software, and the alternatives considered. In Phase II, they produced the project schedule. Phase III was the implementation of the program.

The planned ship date for the new hardware system was for 18 to 20 months from the start date. The overriding concern was getting to market quickly with a 32-bit system to satisfy customers' need for more computing power.

Setting celestial sights

Initially, the VAX and VMS development team used the code name "Star" for the hardware and "Starlet" for the operating system. Thus began the celestial code naming of hardware and software. Plans for a new family of 32-bit systems had already been drawn. Over time, about 40 engineers worked on the hardware development team. Everyone involved expected this new class of computers to propel DIGITAL to the forefront of technology—and the air was charged with excitement and enthusiasm.

Above, VAX-11/780 Announcement, from left to right: Gordon Bell, Richie Lary, Steve Rothman, Bill Strecker, Dave Rogers, Dave Cutler, and Bill Demmer.

Planning the project

Gordon Bell, VP of Engineering, was the primary driver behind the new direction for DIGITAL. Bell drew up plans for a new system that would extend the addressing scheme used on the PDP-11.

While the initial code name for the new system was Star, it soon became known internally as VAX, an acronym for Virtual Address eXtension. When the product was announced, the company added the number 11 to the name VAX to show customers that the new system was compatible with the PDP-11.

Ken Olsen powers up first VAX. The first time the original VAX breadboard (prototype) was powered up, Ken Olsen almost burned his hand on the power supply.

Bill Demmer was the project manager for the VAX project and assembled three design teams to work on the new architecture. VAX A Team developed the concept design plan; VAX B Team initiated some of the architectural extensions and provided ongoing review of the design specification and project plan; VAX C Team reviewed and approved the final project plan and design specification.

Working the plan

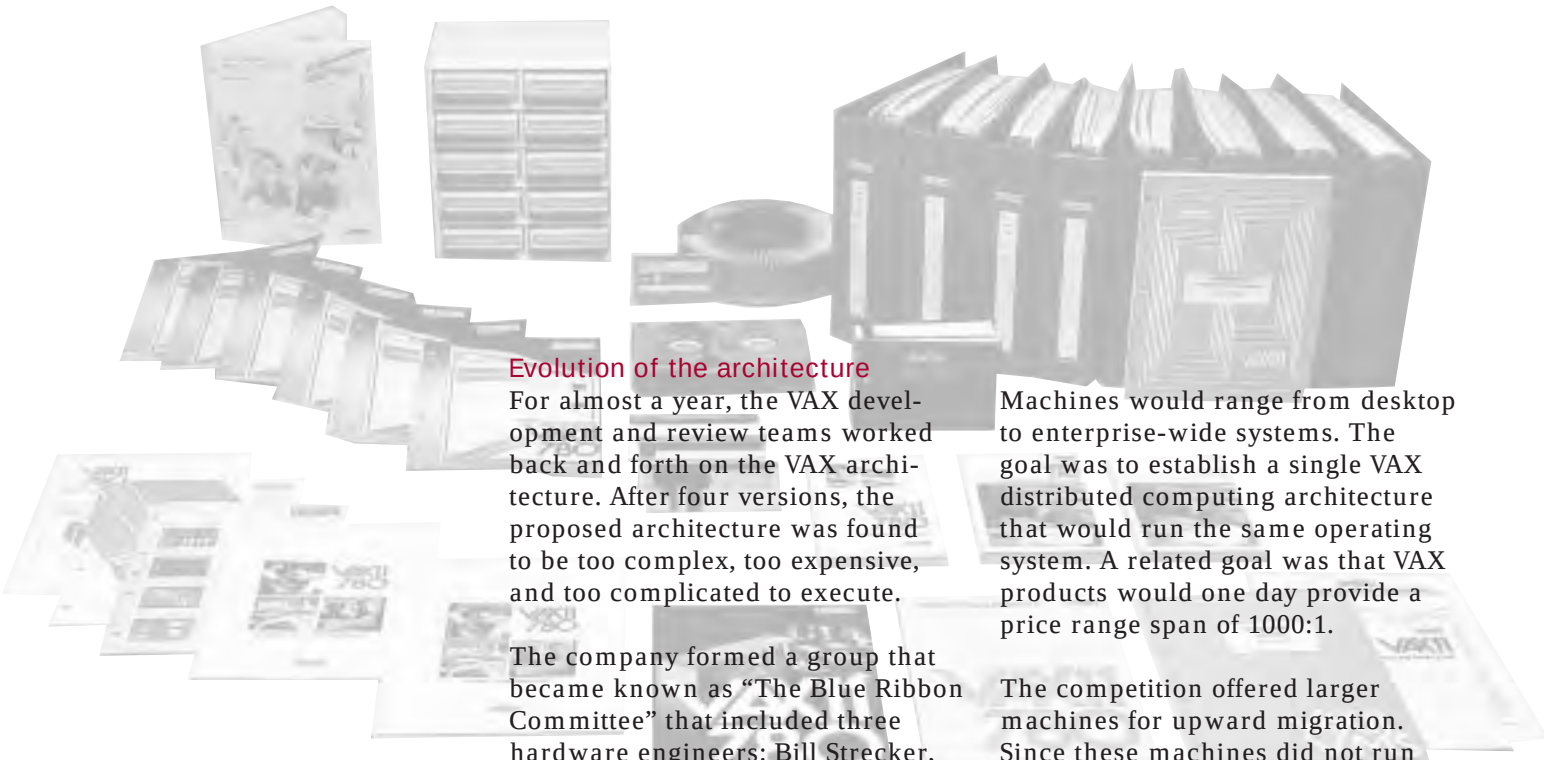
The VAX and VMS kick-off meeting took place in April 1975. The task force—addressing such items as instruction set extensions, multiprocessing, and process structures—closeted themselves away to discuss their options. Their goal was to make the least possible changes in the PDP-11, and still extend it to have a larger virtual address base.

The fundamental questions were, “Can the PDP-11 architecture have its addressing structure expanded to achieve the goals of transparency to the user? Can this expanded architecture provide a long-term competitive cost/performance implementation similar in style and structure to the base architecture?” It was not long before the team realized that both these goals could not be met completely, and so they made the decision to develop a completely new architecture.

All team members agreed that the new system would have to be culturally compatible with the PDP-11 and it would have to maintain the same look and feel as its successful predecessor.

An architecture evolved that cleanly solved the fundamental limitations of the PDP-11. The team developed an implementation plan to overlay both the extended architecture and the basic PDP-11 architecture, thus permitting the new system to appear to be an extended model of the PDP-11 family. This would help to achieve one of the major goals—allowing customers to capitalize on their investment in PDP-11 and grow their systems.

In addition to expanding the address space and ensuring PDP-11 compatibility, another goal was to create an architecture that would support user requirements for 15 to 20 years.



"VAX was the project name—Virtual Address eXtension—but it was never meant to be the product name. When it came time to choose a name, we thought PDP-what? Then some marketing specialist said there are two attributes that are really important in a name, if you want it to be memorable. One is that it be short and pronounceable and that it have an X in it, because Xs are rare letters, so they catch your eye. According to that theory, we had the best name sitting right in front of us: VAX."

—Peter Conklin,
VMS Engineering Manager

Evolution of the architecture

For almost a year, the VAX development and review teams worked back and forth on the VAX architecture. After four versions, the proposed architecture was found to be too complex, too expensive, and too complicated to execute.

The company formed a group that became known as "The Blue Ribbon Committee" that included three hardware engineers: Bill Strecker, Richie Lary, and Steve Rothman, and three software engineers: Dave Cutler, Dick Hustvedt, and Peter Lipman. They simplified the earlier design and created a plan that would be possible to execute. Key modifications included drastic simplifications to the highly complex memory management design and the process scheduling of the proposed system. The simplified architecture, the fifth design evolution of the VAX system, was perfected and accepted in April of 1976—exactly a year after the design work began.

The VAX strategy

Simplicity was the essence of the VAX strategy. The VAX strategy provided for a set of homogeneous, distributed computing system products that would allow users to interface, store information, and compute on any of the products—without having to reprogram their applications. Machines would range from desktop to enterprise-wide systems. The goal was to establish a single VAX distributed computing architecture that would run the same operating system. A related goal was that VAX products would one day provide a price range span of 1000:1.

The competition offered larger machines for upward migration. Since these machines did not run the same code, the code had to be recompiled from the smaller machine in order to run. The DIGITAL single-architecture strategy equated to cost savings for customers and simplified their computing environments.

In addition, a single architecture enabled the building of network and distributed processing structures.

Implementing the VAX hardware

Once the plan was accepted, the VAX and VMS project entailed several months of laying out the basic design for the architecture, followed by nine months of filling it in. The plan was executed by two separate hardware engineering teams. One used existing technology to design what eventually became the VAX-11/780; the other team developed a new VAX chip technology through the DIGITAL fledgling semiconductor group

"I don't think many people ever get the kind of opportunity we did. We had good people, and we grew into a great team. We had lots of differences, but we sorted them out and built what was expected."

—Roger Gourd
Software Engineering Manager

which became the VAX-11/750.

Memory and CPU design

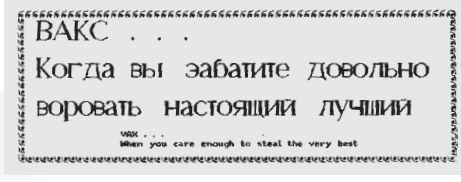
In planning the memory design, there was a question of what size memory and how many bits were needed. Trade-off decisions were made between achieving the best performance and optimizing the number of bits used from a cost perspective. The VAX-11/780 memory design was the first in which error-correction and detection code (ECC) was designed into the system. The semiconductor DRAM (Dynamic Random Access Memory) was susceptible to soft errors. In order to protect the system from memory loss or changing information, it was necessary to store the information in the memory using some code with additional bits of memory. In the unlikely event that one of the bits changed, the memory system could reconstruct the code, know which one changed, and correct the problem.

With virtual memory, memory space no longer had to be in the system's internal memory all at once. Instead, the whole program sat on a disk, while the operating system moved pieces in and out of internal memory as needed. Because the first VAX systems had very little internal memory, this was important. Relative to internal memory, disk memory was economical. Virtual memory also allowed programs that were too large to fit completely into memory to run parts at a time—which was not possible on systems that did not do virtual addressing.

VAX-11/780 – moving from 16-bit to 32-bit computing.

VAX...when you care enough to steal the very best

During the cold war, VAX systems could not be sold behind the Iron Curtain. Recognizing superior technology, technical people cloned VAX systems in Russia, Hungary, and China. After learning that VAX systems were being cloned, DIGITAL had the following words etched on the CVAX chip, "VAX...when you care enough to steal the very best."



Actual Russian words translated: VAX... when you care enough to steal the very best.

"In the early 1980s, we were designing computers so complex, our engineering processes couldn't keep up with them. We discovered we had to use the latest VAX to simulate the new one we were building. Building VAXes on VAXes—our first computers became tools for building the next generation of VAXes."

—Bill Strecker

Chief Technical Officer, VP, CST

CHAPTER IV VMS Software Development

With the VAX hardware development underway, the software development—code named Starlet—began a few months later in June of 1975. Roger Gourd led the project and software engineers Dave Cutler, Dick Hustvedt, and Peter Lipman were technical project leaders, each responsible for a different part of the operating system.

VMS project plan

The Starlet project plan was to create a totally new operating system for the Star family of processors. The plans called for a high-performance multiprocessing system that could be extended to support many different environments. Just as the hardware was designed to be culturally compatible with the PDP-11, Starlet was designed to augment the hardware compatibility by providing compatibility with the existing operating system, RSX-11M.

The short-term goal was to build an operating system nucleus for the first customer shipment of VAX systems. It would have sufficient functionality to be competitive, but would also provide a base that could be extended and subsetted over time for a variety of DIGITAL markets. Long-term goals for the project included quality, performance, reliability, availability, serviceability, reduced support costs, and lower development and maintenance costs. The main focus was to support high-performance applications, such as real-time and transaction processing.

VMS V1 software development team.

Putting it in writing

From the beginning, the software team considered documentation to be a significant part of the project. The first technical writer, Sue Gault, attended design meetings with the software development team and helped them write the Starlet Working Design Paper. This document contained an in-depth technical description of the operating system. Since this project was defined as building a system of hardware and software together, it was more complex in scope than any DIGITAL project to date.

Through the exercise of writing, the engineers received input from the documentation writers and were able to troubleshoot potential problems. Ideas had to be expressed clearly enough to be written in the specs. This method helped to resolve assumptions and potential differences of opinion. The design document also served to keep the rest of the company informed about the VMS project as it was made public—contributing to the overwhelming support and enthusiasm throughout the company for the new project.

Working in tandem

In order to ensure tight integration between the software and hardware, several software programmers attended the VAX design committee meetings and contributed to the hardware design from a software perspective.

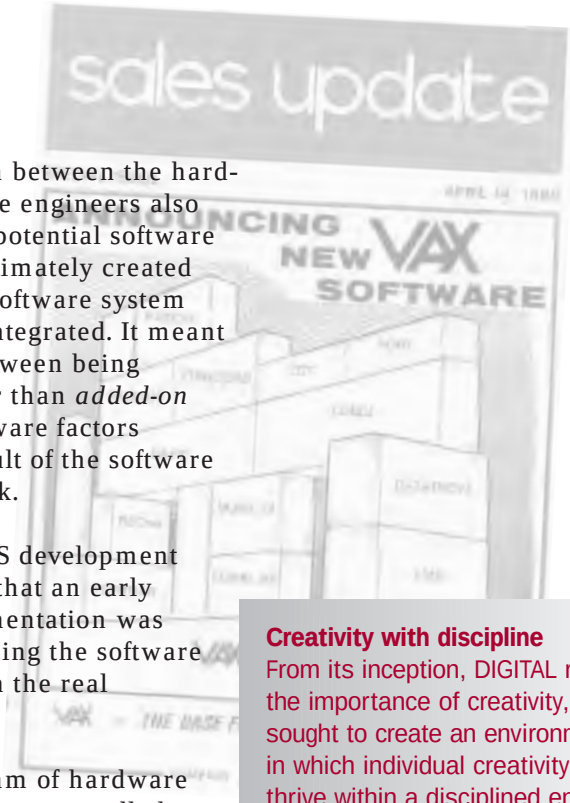
Close cooperation between the hardware and software engineers also helped work out potential software problems, and ultimately created a hardware and software system that was tightly integrated. It meant the difference between being *designed-in* rather than *added-on* later. Many hardware factors changed as a result of the software development work.

The VAX and VMS development team recognized that an early hardware implementation was critical to developing the software concurrently with the real hardware.

Accordingly, a team of hardware engineers built a system called the "hardware simulator." Constructed of PDP-11/70 components, some custom logic boards, and a lot of firmware, it provided a quick first implementation for the VAX platform. The VMS team designed and tested all the operating system software on that simulator. It ran 10 to 20 times slower than the actual system was to run, but it enabled the development team to work and develop the software on the system as it was being designed.

Breadboard on wheels

The VAX-11/780 engineering team first built a machine called the breadboard and placed it on a large metal cart. All the circuit boards were created with wire wrap; while



Creativity with discipline

From its inception, DIGITAL realized the importance of creativity, and sought to create an environment in which individual creativity would thrive within a disciplined environment. The strategy was to hire talented people and empower them to develop plans, have those plans reviewed and approved, and to accept ownership of the project. The company believed that people would be most productive when they had to meet milestones and stay within a budget—this is where the discipline factor came in. Each product line group was responsible for meeting its plan within time and budget constraints.

"Software development is very creative, very individual. We want to give the engineers the freedom to work independently, to work together, and to do the things they want to do."

—Bill Heffner
VP of Software Engineering

the power supplies sat loose on the lower shelf of the cart. The software development team ran a time-shared VMS system on the breadboard for a short time, because it became available around the time VMS had evolved sufficiently to support multiple users. However, the breadboard was not entirely reliable because the operating speed was pushing the limits of what could be handled with wire-wrap construction.

The breadboard was replaced by the first VAX-11/780 etched prototype. The prototype was the first machine built with "real" parts—the real frame, power supplies, etch circuit boards, etc. The only thing lacking was the external cabinet. This prototype was not replaced with a production machine until well after VMS and the VAX-11/780 shipped. The prototype continued to be used for years for stand-alone testing. Systems developed after the VAX-11/780 never went through the breadboard stage, but rather went directly to real etched circuit boards, after extensive simulation.

The software developers used the prototype to do their work—which provided a closed loop of using what they were building. This strategy of using the software system to do the design work helped them to pinpoint potential problems as they progressed.

VMS engineers Dick Hustvedt and Ben Schreiber.

All work and no play? Not at DIGITAL!

Over the years, the VMS engineers laughed together as well as worked together. And so there were a whole series of practical jokes that were played. There were some guidelines: You couldn't prevent people from getting work done. You weren't allowed to do anything that would harm the system or lose a day's work. But anything else was fair game.

Dave Cutler started the first VMS April Fool's jokes. One year, Andy Goldstein replaced the line printer driver so that everything printed out backwards. On another April 1, the entire system message file was replaced with joke messages—including ones like "File not found. Where did you leave it?"

Once VMS engineer Trevor Porter went back home to Australia on vacation. When he returned, fellow engineer Andy Goldstein had bolted a panel in place where the cube "door" was. Trevor walked to his office, observed the situation, turned to Andy and said, "All right, where's the spanner?"

"One of the VMS group's philosophies was that we lived on the software that we were writing. Because if it wasn't good enough for us, then it wasn't good enough yet for our customers."

—Kathy Morse
VMS Engineer

"Roger Gourd passed around the book The Mythical Man-Month by Fred Brooks and almost all the team members read it. Most of us already had one operating system under our belt, so Brooks' discussion of the 'second system effect' struck home. The 'second system effect' results from each engineer wanting to fix all the mistakes and shortcomings of their first system. Left unchecked, the second system effect can cause runaway complexity that can be disastrous for software quality and schedule. A new term entered the programmers' lexicon—'Creeping elegance'—a process in which a design is successively refined to be increasingly complete, eventually yielding a result that collapses because of its size and complexity. The entire software team was very conscious of maintaining the balance between producing a functional, high quality product and staying on schedule."

—Andy Goldstein
VMS Engineer on original development team

Who's got the red flag?

The software build environment process, which is what transforms the software source code into a runnable system, allowed only one person at a time to do a build. If two people tried to do a build at the same time, they would overwrite each other and produce nothing useable. The engineers—who often worked in an intense, heads-down mode—had no way of knowing if another engineer was working on a build. It was inevitable that early on, two engineers in adjacent offices would try to do builds concurrently, thus destroying each other's work.

Being a creative team, they came up with a creative solution. As a mechanism for determining who was working on builds, a red flag with a magnetic holder was put up in the cubicle of the person using the simulator.

It was usually referred to as "the mutex," in reference to a commonly used software synchronization mechanism. If an engineer wanted to do a build, he or she found the flag and asked its current owner "Can I have the mutex?" and it would be theirs as long as the flag holder wasn't in the middle of a build.

Developing tools on the fly

Out of necessity, the team developed many of their own tools as the project progressed. For performance evaluation, the VMS engineers built the performance monitor tool and then used the tool to measure system performance. One part of the monitor was a separate computer system running on a PDP-11 that could act as a time-sharing workload. Using that, the engineers measured VMS on a number of different multi-user workloads to see how it performed for time-sharing.

Using what you're building

There was a lot of back and forth communication between the hardware and the software engineers. The writers were also using the software, which provided a good closed loop process. And that was the philosophy behind it—to use it and debug it as the project moved forward.

Ensuring compatibility

The development systems for VMS were housed in one large computer room; most of it was taken up by a huge dual-processor DECsystem-10 and a PDP-11/70 was shoehorned into one side of the room. Much of the first version of VMS was written in Macro and the rest in Bliss. Macro development was done strictly on the PDP-11, using a cross-assembler. The assembler object modules were then linked into executables on the PDP-11 and written to a disk which the VMS engineers would then carry over to the VAX system in the next room.

VMS documentation set.

The DEC-10 was used to compile the VMS modules written in Bliss because at the time the Bliss compiler only ran on a DEC-10. The Bliss code had to be transported by tape to the PDP-11 to be linked.

This process of writing programs initially required a great deal of time and effort. However, the new virtual memory operating system was built in a relatively short time by any current standards.

The VMS system kernel and related critical function were written in "native mode" using the new VAX instruction set. However, many utility functions were simply ported from the RSX-11 operating system and so ran in "compatibility mode"—the PDP-11 emulation mode. Besides speeding up the implementation of these functions on VMS, this approach provided an effective live test of the VAX platform and VMS operating system compatibility features.

The virtual memory system software provided greater functionality than had ever been seen before in a minicomputer. VAX and VMS also supported networking capabilities as well as compatibility with PDP-11 thus enabling customers running PDP-11 programs to migrate their applications to the new VAX and VMS systems quickly and easily.

The VMS strategy

The VMS software strategy was based on developing a single VMS operating system that would span the product range from low-end to high-end. VMS would offer full mainframe capabilities allowing concurrent batch processing, transaction processing, time-sharing, and limited real-time processing.

This *single operating system* strategy behind VMS was a reaction to the multiple operating systems of the PDP-11:

- RT-11 for real-time and laboratory work
- RSTS-11 for educational and small commercial time-sharing
- RSX-11 for industrial and manufacturing control
- MUMPS-11 for the medical systems market
- DOS-11, the original PDP-11 operating system, largely superseded by the above.

That's not an abandoned car—it's a VMS engineer's car

"People worked a lot of overtime during the creation of VMS. At one point, we hired an engineer from California, Ralph Weber. For the first week he had a rental car and was living in a hotel. He got there so early that he parked in exactly the same spot every morning, and he stayed late. After a week, a security guard thought the car had been abandoned and called the car rental place to come and collect it. That night Ralph went to leave, and his car was gone. So he ran into the security room shouting, 'My rental car's been stolen!' They started to call the police and then, luckily, another security guard came in and said, 'No, no, we had that one towed today because it's been there a week and we thought it had been abandoned.'"

—Kathy Morse
VMS Engineer

VMS Version 3 release party on Cape Cod.

While each of the PDP-11 operating systems was targeted to a particular market segment, there were a lot of cross-over sales. At the same time, the multiple operating systems with incompatible interfaces diluted the system base for applications. Any application might have to be implemented in multiple versions to run on a large number of systems.

Therefore, the strategy with VMS was to have a single operating system that would be sufficiently flexible, powerful, and efficient to address most of the PDP-11 target markets.

Betting the business on VAX and VMS

Prior to developing the VAX system and VMS operating system, DIGITAL operated according to a multi-product line environment. However, in 1978, DIGITAL adopted a vision called The VAX Strategy which would guide the company through the next decade. Although DIGITAL would continue development on the PDP-11 and DECsystem-10, the company's main direction would be on VAX development.

The VAX and VMS strategy led to a consistent message from DIGITAL: "One platform, one operating system, one network." Simply put, DIGITAL decided to bet the business on VAX and VMS—and VAX and VMS business began to skyrocket.

"As the technical writer, my belief was that the technical writer is the advocate for the customer. So I always put myself in the shoes of someone who is trying to learn how to use the system, and wrote the documentation accordingly.

"The VMS Documentation Group grew from five people in 1977 to 45 in 1987, and the documentation set grew from 9,000 to 20,000 pages. It was a massive effort."

—Patti Anklam
Technical Documentation Writer

Debugging in the Blizzard of '78

On the first evening of the blizzard, Andy Goldstein was working late on the new VMS file structure. If he couldn't make it home, he wasn't worried. Hank Levy lived across the road from the Mill, so anybody from VMS who was really stuck would just pound on his door and sleep on his couch.

"I hit a bad directory error and said, Oh my God, I've got a bug in the file system. I was trying to collect data on this, but the snow was getting deeper and we lost power. The whole state was closed for the next week, but I drove to the Mill and talked my way inside. I powered up the machine, got dumps of the failed directory, and took them home with me.

"I called Richie Lary—who lived across town from me—and said, 'Richie, I think there's a bug in the microcode.' And he said, 'Why don't you come over. I've got the microcode listings here.'I walked through the snow over to his house. Richie fished a six-inch binder out from under his bed and we went through it, and sure enough, we found the bug and fixed it."

—Andy Goldstein
VMS Engineer

CHAPTER V Market Acceptance—Beyond Expectations

Rolling out the first VAX and VMS systems

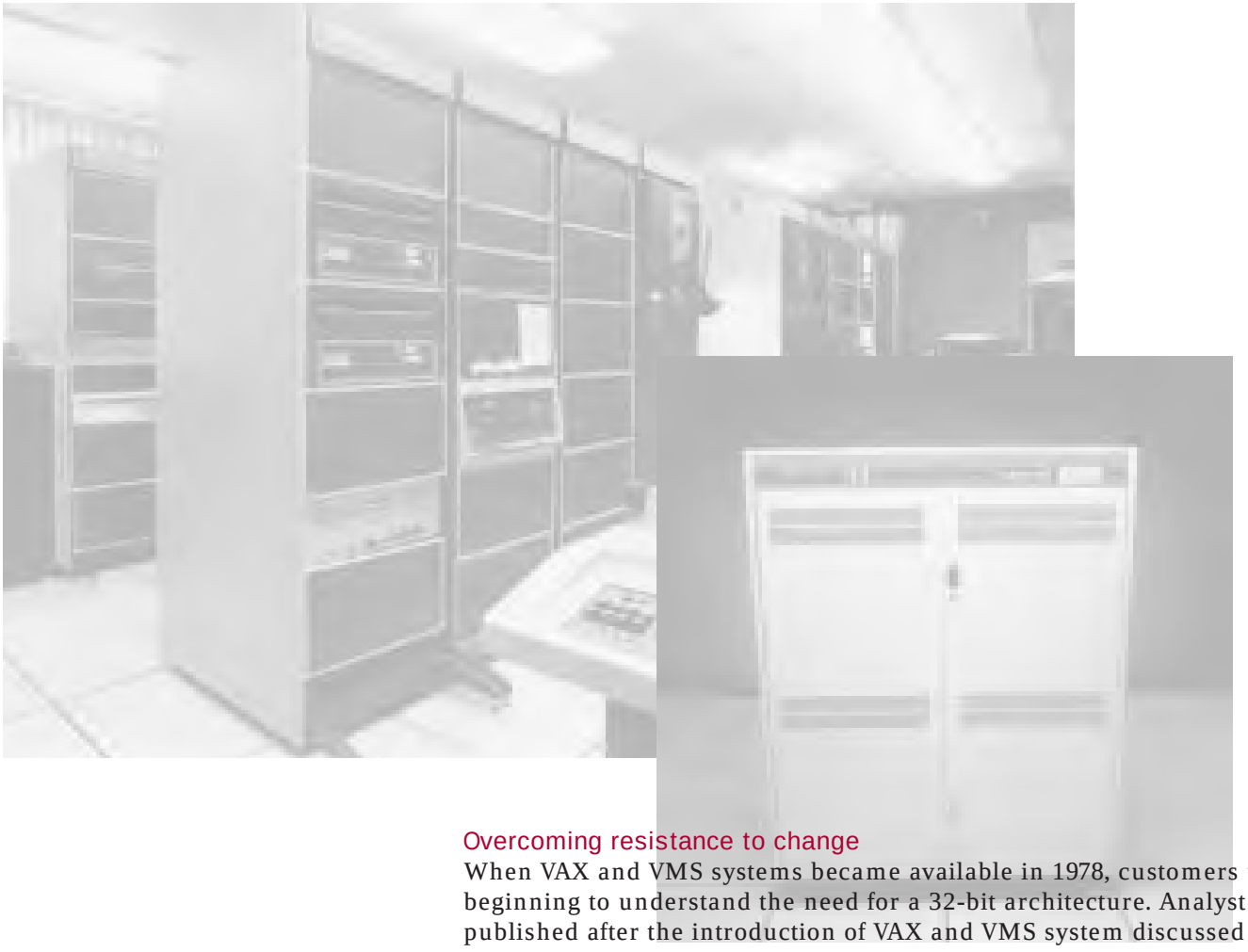
Roughly 18 months after the design team sat down to execute their plan for the new interactive architecture, the first machine rolled off the manufacturing floor and into a customer site.

The first VAX-11/780 was installed at Carnegie Mellon University and was released to more than 50 customers. In 1978, the VAX-11/780 became accepted internationally with installations at CERN in Switzerland and the Max Planck Institute in Germany.

A major industry contribution

In October of 1977, DIGITAL made a significant contribution to the industry by announcing both a new architecture hardware product and a new architecture-based operating system. One of the primary advances that the VAX architecture brought to computing was that it had a plan for the intercommunication of computers at the architectural level. DIGITAL had not only engineered the capability for computers to talk to computers with homogeneous existing architectures, but had planned for a complete range of computer systems—from the personal workstation level up to the high-performance systems—all having a homogeneous architecture.

President Ronald Reagan visits a DIGITAL VAX manufacturing facility with DIGITAL President/CEO, Ken Olsen.



Overcoming resistance to change

When VAX and VMS systems became available in 1978, customers were just beginning to understand the need for a 32-bit architecture. Analyst reports published after the introduction of VAX and VMS system discussed the significance of the 32-bit architecture.

Woods meetings

In 1983, DIGITAL began to hold day-long, off-site meetings. Initially these meetings were held at Ken Olsen's cottage deep in the woods of Maine. Soon, these off-site strategy meetings became known throughout the company as woods meetings—regardless of where they were held.

While some forward-thinking customers embraced the advantages of the 32-bit architecture—especially in specialized scientific applications—many were still satisfied with their current 16-bit architectures and didn't think the larger addressing space was necessary. Resistance to change is always an obstacle in introducing new ideas, and the VAX platform certainly represented a change for customers.

Migrating from PDP to VAX

As customers saw how efficiently VAX and VMS worked in their environments, acceptance for the new system grew overwhelmingly positive. Organizations suddenly proclaimed, "We are a VAX and VMS company" and focused all their efforts in that direction.

The VAX system drew on years of DIGITAL engineering experience in developing the PDP family of computers. Wherever possible, the VAX architecture took advantage of existing PDP-11 technology such as the UNIBUS—thus allowing existing PDP-11 I/O technology and products to be used on the VAX systems.

Thus, the new VAX system appealed to the installed base of PDP customers because of the built-in *compatibility mode*—which provided an easy migration path for moving up to the new 32-bit architecture, while still protecting their existing PDP investment.

Key success factors

32-bits at an affordable price

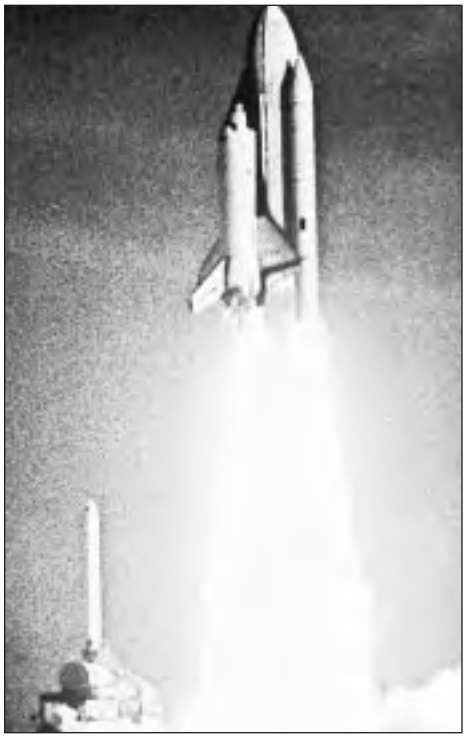
Although the VAX-11/780 was not the first 32-bit system on the market, it was the earliest computer that was capable of taking on large-scale problems at a reasonable price.

FORTRAN for the scientific world

The DIGITAL investment in VAX FORTRAN is credited with some of the VAX and VMS architecture's early success in the marketplace by gaining a leadership role in the world of scientific and technical computing.

Two aspects of VAX FORTRAN contributed to the early success of the VAX. First, the compiler produced excellent quality, fast performing code. Since it was a very complete implementation of FORTRAN, a FORTRAN program written for a competitor's machine could easily be brought over and run on a VAX system. Second, the interactive, source-level debugging allowed the programmer to interact with the program in FORTRAN, rather than machine language.

VAX systems became the first workhorse for numerical and scientific computing, supporting such power-hungry applications as computer-aided design, flight, operator training for nuclear and conventional fuel power plants, power monitoring and control systems for electric utilities, and seismic data reduction.



VAX and VMS play a part in the Space Shuttle development.

Scalability

By design, the VAX architecture was scalable, meaning that code written on small machines would run unchanged on larger machines. This made software development affordable, because the concept could be tested on a small machine before making a major hardware investment. Applications were not limited to a particular machine. Once an application was written, it could run on any size VAX system without changes. Scalability allowed customers to grow their VAX systems as they needed—without worrying about their software investments. It also minimized maintenance and support costs for software.

Connectivity

Another significant success factor was the connectivity strategy—interconnecting computers via networks. DIGITAL had developed DECnet in 1973, and support for this networking software was an integral part of the VAX strategy. The ability to connect computers gave minicomputers the power of mainframes. Distributed computing was an emerging concept and DIGITAL was in the leadership position.

The VAX and VMS architecture allowed for networking that was more efficient than in other systems at the time. Networking allowed DIGITAL to expand the application base for VAX platform and broaden its market base beyond the scientific and into the commercial world.

Software capabilities
With the increased breadth of software offerings of VMS V2.0, DIGITAL moved into the business marketplace.

These factors—along with the extensive software library and other interactive features of VMS—made it the best software development environment in the industry. In addition, the robustness and reliability of even the early VMS versions ensured that a customer's programming staff spent their time working on their programs rather than figuring out what had gone wrong with the operating system.

In short, the expanded address space, sophistication of the operating system, inherent networking capabilities, and affordable price were the integral factors in the success of this new technology.

CHAPTER VI Moving into Commercial Markets

DIGITAL President/CEO, Ken Olsen addresses customers to deliver new product announcement.

By 1979, the company's sales revenues topped the $2 billion mark for the first time. DIGITAL was a major player in the worldwide minicomputer market, and was marketing its systems, peripherals, and software in 35 countries around the world.

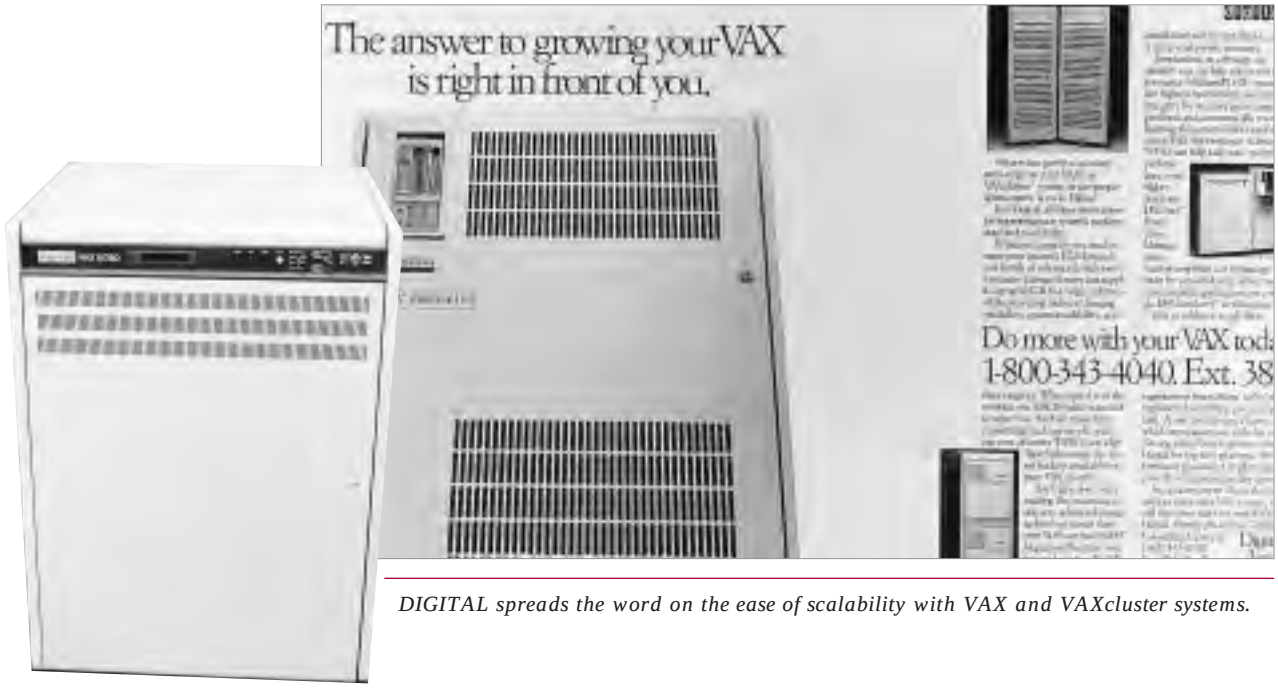
This success was attributed to Ken Olsen's original strategy of selling small, easy-to-use computers to a wide range of customers. Research scientists, accountants, banks, and manufacturers alike could use these systems.

By the 1980s, DIGITAL had established itself as the number-two computer company behind IBM. At this time, VMS Version 2.0 commercial software was introduced. This second generation of VMS provided to the commercial marketplace the same leadership that FORTRAN did for the scientific world. By the spring release in 1980, Version 2 had users at 1,400 sites.

With this announcement, DIGITAL made a commitment to the commercial marketplace and positioned the VAX system as the flagship product for new commercial applications. The company also emphasized networking and distributed data processing concepts as ongoing efforts.

Expanding the family

The initial evolution of the VAX family was downwards in size and price. This was a very deliberate strategy that was established when the VAX architecture was first conceived; the slogan was "$250K and DOWN." Even though the DEC-10 and DEC-20 systems were still going strong , the intent was that the VAX system would provide a replacement for high-end PDP-11 systems—without encroaching on the DEC-10/20 business.



DIGITAL spreads the word on the ease of scalability with VAX and VAXcluster systems.

Thus the first two successors to the VAX-11/780 were smaller, less expensive machines. The first successor to the 780 was the VAX-11/750. The VAX-11/750 was built of semi-custom LSI logic known as gate arrays. Each gate array chip consisted of about 400 standard logic functions. By interconnecting the basic functions, each chip was specialized to provide the needed functions of the 750 CPU.

The VAX-11/730 was the third member of the VAX family, introduced in 1982. The VAX-11/730 was built from off-the-shelf bit slice microprocessor and programmed array technology.

More power, please!

Meanwhile, some customers were beginning to clamor for more powerful VAX systems. In an effort to meet this demand, DIGITAL produced the VAX-11/782. This system was built with two standard VAX-11/780 processors using a shared memory. By supporting the VAX-11/782, VMS took its first step into multiprocessing—foreshadowing the symmetric multiprocessing capabilities of the VAX 6000 series years later.

This system was followed by the VAX-11/785—a re-engineered VAX-11/780 that used the same design with upgraded components—which allowed the CPU to be run at a 50% faster clock rate. Both the VAX-11/782 and VAX-11/785 were designed to bridge the long gap between the 780 and the 8600.

These new members of the VAX family—combined with ever-improving networking capabilities—provided significant and varied configuration possibilities for DIGITAL customers.

Increased software capabilities

The backbone of the new VMS commercial software capabilities was represented by six new products:

- COBOL was the flagship product
- BASIC was interactive and fast
- Multikey ISAM provided effective data management and was usable from all languages
- Integrated DECnet enabled multi-system communication
- DATATRIEVE V2 provided online inquiry and retrieval
- Form Management System (FMS) allowed for data entry and transaction-oriented applications.

This extended the range of capability and power—enabling commercial customers to distribute their data processing more easily and efficiently.

CHAPTER VII Networking

DIGITAL recognized early on that its customers needed a means of connecting various systems and coordinating their capabilities. To address this need, the company began research in this area as early as 1972, when it developed a multiprocessor that would combine a number of minicomputers to obtain the power of a large mainframe. This was accomplished through networking.

By 1973, DIGITAL formed a group to direct the design and implementation of a networking project; its goal was to achieve absolute compatibility and interconnectivity across all computer families. The company's initial efforts resulted in the DIGITAL Network Architecture that implemented a layered protocol approach. This method of connecting systems was recognized as state-of-the-art technology and put DIGITAL in the leadership position in the industry.

DECnet

In 1974, DIGITAL introduced DECnet, the industry's first general-purpose networking product for distributed computing. One goal was to make networks affordable so that customers could implement them more widely.

DECnet for VAX and VMS V1.0 was available for the first customer shipment of the VMS operating system, and significantly contributed to the success of the VAX-11/780 system.

Unlike earlier networking products—which focused on connecting terminals to hosts—DECnet provided peer-to-peer networking for the first time. This was a major step toward the client/server computing model. DIGITAL had developed the best and least expensive distributed computing solution and became an industry leader with this technology.

DECnet linked DIGITAL systems together in a flexible network that could be adapted to changing requirements. It provided direct communication among computers at the same organizational level, and had no hierarchical requirements or prerequisite host processors. DIGITAL networks were modular and flexible—as opposed to IBM's rigid, hierarchical products—and could connect computers from other vendors, providing a degree of compatibility among different computer systems that was unmatched in the industry.

Five phases of DECnet

Phase I: Supported point-to-point (directly wired connections) and task-to-task (customer applications could be coded to talk to each other over the networking protocols).

Phase II: Added remote file access and general task access (i.e., an application could invoke general command procedures on a remote system). This version of DECnet was supported by VMS V1.0, thus VMS had remote file access built into the base file system from day one.

Phase III: Added routing, which meant you no longer had to have a directly wired connection between two systems to allow them to interact via DECnet. Rather, network traffic could be forwarded between two systems by one or more intervening routing nodes. It also provided SET HOST (the ability to log into a remote system interactively) and MAIL—the beginnings of corporate electronic mail.

Phase IV: Added Ethernet support. Ethernet eliminated the requirement for point-to-point wiring, allowing many systems to be connected to a single wire in a Local Area Network. Phase IV also provided a larger address and the concept of areas (analogous to telephone area codes), thus allowing a network to grow to as large as 65,000 nodes.

Phase V: Incorporated OSI standard networking into DECnet. Supported unlimited address space/nodes when using OSI addressing; supported 100,000 nodes if using large local files.

Over the years, DECnet evolved through five releases, each designed to work with the next and previous phase. DIGITAL also contributed to the major networking standards, incorporating key standards such as OSI and TCP/IP into DECnet.

Enter Ethernet

Ethernet communications capabilities were incorporated into DECnet Phase IV, allowing DECnet users to extend their networks with local area capabilities of Ethernet.

The era of the Ethernet brought an entirely new concept to networking. DIGITAL set the standards with Xerox and Intel by establishing Ethernet as the industry choice for local area networks. The three companies jointly defined the Ethernet standard, which led to the deployment of local area networks. Ethernet became the medium-speed but long-distance network, connecting components as far apart as a kilometer.

CI, NI, and BI interconnects

DIGITAL coined the terms CI, NI, and BI as part of an effort to rationalize the company's strategy for interconnecting the components of computer systems at different levels of implementation.

NI—Network Interconnect. This was the highest level interconnect, connecting computer systems in a network. NI quickly became synonymous with Ethernet. Ethernet allowed the construction of local area networks of up to a thousand connections and a mile and half in size.

CI—Cluster Interconnect. The CI also connected individual computer systems. In contrast to the NI, it allowed much smaller configurations: up to 16 systems spread over a 90-foot radius. What the CI lacked in scale it made up for in speed—allowing communications over 10 times as fast as the NI. DIGITAL developed storage controllers that connected to the CI, providing the basis for clustered VMS systems (see next section).

BI—Backplane Interconnect. The backplane was used to connect components of a computer system within a single cabinet. The BI was built to be a faster replacement for the UNIBUS, used by all PDP-11s and the initial VAX systems. The VAX 8200 and 8300 used the BI as their "native" interconnect (i.e., both I/O and main memory). Later VAXes (other 8000 and 6000 series) used the BI strictly to connect to I/O controllers.

SI—Storage Interconnect. A standardized connection between a storage device (disk or tape) and its controller.

XI—Everything Interconnect. A future interconnect that would replace NI and CI, being both faster and larger than either. Something like the XI was ultimately realized with FDDI, but displaced neither the CI nor NI.

The next logical step of networking was computer clustering—a concept that DIGITAL pioneered. Today, the company continues its position as the industry leader in clustering.

CHAPTER VIII The Second VAX Generation

The VAX 8600

In October 1984, DIGITAL announced the VAX 8600. This system marked the beginning of the second generation of VAX machines—and a new milestone in the VAX strategy. The VAX 8600 offered up to 4.2 times the performance of the VAX-11/780 and increased I/O capability while maintaining I/O subsystem compatibility with the VAX-11/780 and the VAX-11/785 Synchronous Backplane Interconnect (SBI).

It was the first VAX implementation in ECL (Emitted Coupled Logic) technology and the first to include macropipelining. The VAX 8600 represented the confluence of many new concepts and further refined the solid engineering of earlier systems. It was packaged with an extensive portfolio of VMS software products that could run on the VAX 8600, as well on all the earlier models.

The VAX 8600 team.

One platform, one operating system, one network

While DIGITAL had considered as many as eight different approaches to networking, the company crystallized its approach to networking in 1983 and announced its networking strategy at DECworld '83. That strategy was one platform (VAX), one operating system (VMS), and one networking product (Ethernet).

VAX production line and test station.

Happy 10th Birthday, VAX and VMS

The 10th Anniversary of the VAX platform and VMS operating system in 1987 was celebrated at DECUS with a VAX-at-10 Dinner Speech. The company discussed VAX architecture goals and presented an overview of the development of VMS.

DIGITAL noted VAX architecture had achieved one of its initial goals—that of providing a price range span of 1000:1. The company achieved that goal in February of 1987 with the announcement of the VAXstation 2000, priced at $4,600; while the VAX 8978 was available for $5,240,000. The company also discussed how VAX and VMS grew from FORTRAN-only in 1977 to "101" layered products in one system in 1987.

A new high end: the VAX 8800

In January 1986, DIGITAL introduced its top-of-the-line VAX 8800 and the midrange VAX 8300 and VAX 8200. These VAX systems were the first VAX systems to support dual processors. Each machine incorporated a new high-performance I/O bus, the VAXBI. The high-performance VAX 8800 achieved application throughput that was two to three times faster than the VAX 8600.

A year later, the company introduced the VAX 8978 and 8974, the most powerful systems from DIGITAL to date, offering up to 50 times the power of the VAX-11/780. Both machines included the new 2.5 Gbyte SA582 Storage Array from DIGITAL. Combined with the HSC70 I/O processor and the VAXBI bus, the SA482 delivered mainframe-class I/O subsystem performance array and large storage capacity.

"From the late 1970s to the late 1980s, DIGITAL moved from being what I would call a niche mini-computer company to the second largest computer company in the world. And that growth was entirely driven by our VAX and VMS business. From that standpoint, VAX and VMS is one of the really great success stories in the history of computing, in terms of totally transforming a company and totally transforming an industry and playing a major role as one of the truly major computer architectures. Certainly VAX and VMS has been a driving architecture for ten years, and is still a very important architecture at age 20."

—Bill Strecker

Chief Technical Officer, VP, CST

Don Harbert and Pauline Nist accept The PC Week *corporate satisfaction award for the VAX 6000 from Susan Pasieka for the second year in a row. – Summer 1992*

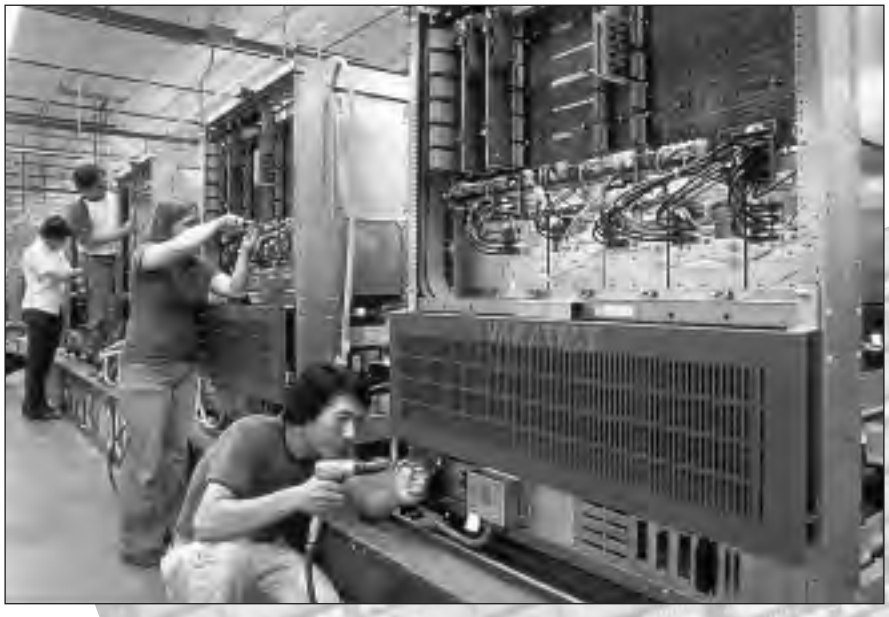
Ongoing engineering challenge: Evolving the architecture

By all standards, the VAX and VMS architecture was very stable from the late 1970s to the 1980s. The reasons for this stability were two-fold. First, because the architecture was so extensively engineered, it didn't require any architectural changes over that ten-year period of time. Second, it offered virtually everything a customer might want. The architecture had been designed for longevity, and it succeeded in that goal.

Over a 10-year time period, the product line broadened from a single VAX-11/780 to a whole family of products that offered continually improved performance at a lower cost. The next engineering challenge was to make successively faster implementations of the architecture at a lower cost.

While some development teams were working on the traditional VAX systems, others were developing a new breed of chip-based systems, which eventually became the main line of DIGITAL's products.

Bill Demmer with 2nd generation VAX family members.



VAX assembly line.

"When we announced the new VAX and showcased it at DECUS, customers would come up and ask, "What is that?" We told them that this is a new VAX. They were pleasantly surprised.

Our customers loved their VAX systems. They just put them in a closet and forgot about them— they're that reliable. It's a tribute to the hardware, the architecture, and the software, because it's bullet-proof. The Catamount was proof that DIGITAL continues to support its Installed Base customers."

– Ed Yee
Senior VAX Product Manager

VAX 6000

The VAX 6000 was the first volume SMP VAX. In the first six weeks of production, there were 500 units shipped. The VAX system was the so-called tornado of that time frame—the market just sucked them up. The shipments grew from a rate of zero to 6,000 units a year in about five months, which continued for a couple of years.

VAX 8000

VAX Family

VAX 8600

ALPHA AXP 2100

VAX-11/780

VAX 6230

VAX-11/730

1977-1997 ...

VAX 6200

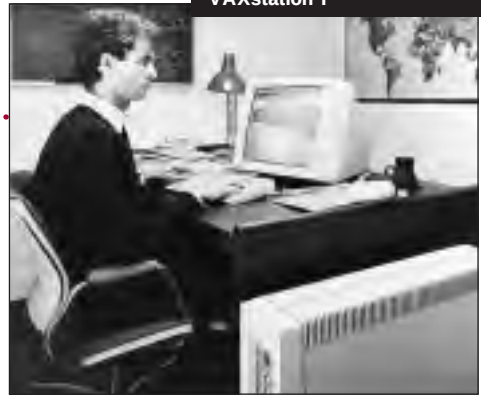
AlphaServer 4100

MicroVAX II & MicroVAX 2000

VAX-11/782

VAX 8650

VAXstation I



VAXstation 2000

VAX-11/750

VAX 4000 200/300

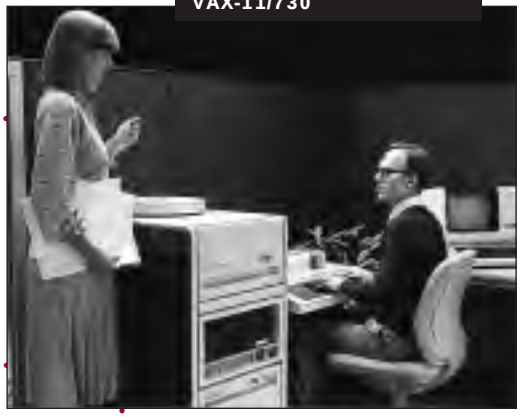
AlphaServer 8200

and beyond

VAXft 3000

VAX-11/785

VAX-11/730



Bill Demmer and the VAX Family

VAX-11/750 Family

VAX Family

CHAPTER IX Putting the VAX on a Chip

Roots of the DIGITAL semiconductor group

The inception of the DIGITAL semiconductor group occurred in the early 1970s with the development of the LSI-11 for the PDP family of computers. DIGITAL designed the chip and partnered with other companies for fabrication. By the late 1970s, technological advances in semiconductors had made chips more powerful and less costly to produce. It became clear that semiconductor technology was imperative in order to remain competitive in the computer industry.

VAX quality control inspection of VAX 8600 CPU board.

Developing the V-11: The first VAX chip

In 1981, an advanced development team explored ways to bootstrap capabilities in semiconductors and design a full-scale VAX on a chip. This project, V-11, was intended to be a full-scale VAX CPU, implemented with state-of-the-art semiconductor technology—N Channel or NMOS. As such, it required four different chips in the implementation.

As the project moved forward, it became clear that microcomputer systems were going to be built very differently from the way the V-11 was being built. Microcomputer systems were going to be based on single chip microprocessors aimed at a dramatically lower price.

DIGITAL addressed the question of whether the VAX design could be turned from being a minicomputer architecture implemented in silicon into a true microprocessor architecture that could be competitive with industry microprocessors. The company decided to do the latter.

VAX systems earn their stripes

VAX systems—due to their performance, network capabilities, and scalability—found their way into many military and Department of Defense applications. Developers developed military/DoD programs for Command, Control, Communication, and Intelligence (C^3I) applications. The highly scalable VAX and OpenVMS architecture performed well in the computer rooms and back lines. But there was a need to bring the VAX technology closer to the harsh environment of the battle front.

United Technologies' Norden Systems, a prime contractor located in New Hampshire, licensed the VAX architecture and developed a militarized version of the VAX on a chip called the MIL VAX II. The system's cost was five times that of a commercial VAX system, but ran significantly faster than its civilian brother. This system met military environmental testing standards, including temperature, vibration, shock, salt, fog, dust, explosive atmosphere, and humidity. MIL VAX II was suited for database management, command, control, and intelligence operations aboard ships and airplanes and in-land installations.

Over the years, other VAX systems have been "ruggedized" by many third-party DoD contractors for use in less severe military applications. These systems were used on shipboard and mobile applications where they had to withstand the rigors of shock and vibration.

The V-11 resulted in the VLSI VAX chip, which was shipped in the VAX 8200 and 8300 series systems. The V-11 was replaced by the MicroVAX chip, but it provided the design technology, basic architecture, and many of the building blocks that made up MicroVAX.

Designing the MicroVAX: The first chip-based VAX

The V-11 and the MicroVAX I were developed more or less concurrently. The VAX-11/750 was the first DIGITAL system to be designed with LSI semiconductor technology, using gate arrays. After the 750, DIGITAL designed the MicroVAX I—one of the first DIGITAL projects to include silicon compilers—with the consulting help of Carver Meade, a pioneer in integrated circuit design. Building on the experience of the MicroVAX I, the company soon followed with the more powerful MicroVAX II.

While the V-11 was designed as a full VAX implementation, the MicroVAX I was designed as a VAX subset. The MicroVAX I system was developed in the company's Seattle facility, headed by Dave Cutler. Because the MicroVAX I was a much simpler design than the V-11, and because of the use of the silicon compiler tools, it was completed before the V-11.

DIGITAL explored the option of having one of the industry's semiconductor companies produce the chip, but decided to do the work internally because of the complexity of the task and the aggressive schedule. This proposal was considered radical because it placed a great deal of faith in the then fledgling chip organization for both design and manufacturing.

Introducing "the first VAX you can steal"

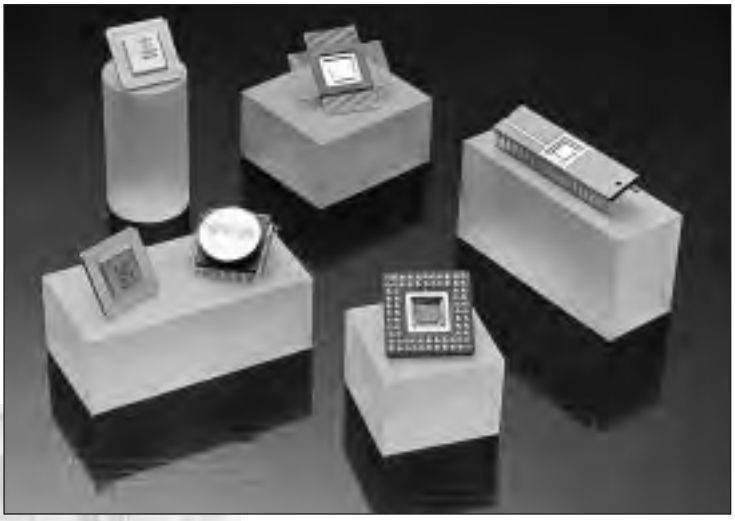
The MicroVAX project was launched in July of 1982 and the silicon was finished on February 4th of 1984—just 19 months later. It was an achievement that was unprecedented in the industry. The entire semiconductor organization rallied around this effort and gave the chip top priority in terms of fabrication and debugging. Thus, they were able to demonstrate the chip running VMS by August of 1984, field test the system in late 1984, and ship it in May of 1985.

Drastically different from any of the earlier VAX systems, the MicroVAX II system was wildly successful. It was the first VAX under $20,000. Commenting on its unprecedented affordability and size, Ken Olsen called it "the first VAX you can steal."

"Bob Supnik had come up with this wonderful scheme to build a VAX on a chip, which became the MicroVAX II chip, ultimately."

—Jesse Lipcon

Senior VP, UNIX and OpenVMS Systems Business Unit



Building on the success of the MicroVAX

The success of the MicroVAX II set the course of development for the VAX chip family for the rest of the 1980s. By reducing the VAX CPU to such a small package and exploiting semiconductor technology, DIGITAL was able to continually improve performance at a dramatic rate.

After the introduction of the MicroVAX II, the company's hardware and software engineers worked together to add back four more instructions out of the commercial instruction set, and the COBOL designers created a version of the compiler that didn't require the complex decimal instructions that had been left out.

The MicroVAX II project would not have been possible if the VMS group had not vigorously supported the whole concept from the outset. When MicroVAX was first put together, it was a more drastic departure from the VAX architecture than the final design—particularly with its simplified form of memory management. But the original VAX memory management was reintroduced back to MicroVAX II to make it a machine with complete functionality.

The MicroVAX II was the system that put the VAX CPU on a chip. With powerful VAX virtual memory, 32-bit computing power, and software compatibility across all VAX processors, the MicroVAX II microsystem provided functionality and flexibility that was unparalleled in the industry.

Skyrocketing sales

DIGITAL was showing the largest volumes of VAX systems sales ever. Up to that point, a highly successful VAX system sold 2,000 units in its lifetime. MicroVAX sold 20,000 units in its first year.

The VAXstation 2000

The VAXstation 2000 was a step down in size from the MicroVAX II. Like the MicroVAX II system, it was built around the MicroVAX II chip. Where the MicroVAX II was housed in a small, desk-side cabinet and supported a variety of PDP-11 peripheral devices, the VAXstation 2000 came in a shoebox-sized cabinet. All the essential functions—CPU, graphics display controller, disk controller, and two serial ports were integrated on a single circuit board. Its peripherals were limited to a keyboard, monitor, and mouse, plus up to two fixed disks, and a floppy disk and tape drive. In return for those limitations, it delivered near VAX-11/780 performance for a $5,000 entry price. Customers called it "a MIP on a stick."

In its first year, the VAXstation 2000 sold 60,000 systems. This demonstrated the principle of elasticity—showing that if you have a capability and you bring its price down, you enhance its marketability. Now, with unprecedented affordability, everybody wanted a VAX.

CVAX

The company's second chip was called CVAX—the C stood for CMOS. A conversion in technology from the earlier NMOS (N channel, metal oxide semiconductor) to CMOS (complementary metal oxide semiconductor) was due to the market's relentless climbing power requirements.

This second-generation VLSI VAX microprocessor offered 2.5 to 3.5 times the power of its predecessor. It was the company's first internally manufactured CMOS microprocessor. High performance came from features such as a macro-instruction pipeline, 1 K Byte onchip datacache, and a 28 entry onchip translation buffer.

The CVAX chip was also much more complicated than the MicroVAX chip. The engineers had to develop the CPU/Floating Point functionality in VLSI and develop separate VLSI chips for Memory Control, the Q-Bus Interface, and a Support Chip which included the Time of Year Clock and Serial Line Interfaces. The number and complexity of these chips added significant challenges to the project.

The CVAX chip was introduced in the MicroVAX 3500 and 3600 systems in September 1978. Another CVAX-based system, the VAX 6000 platform, was announced April 1988.

Incorporating SMP

The CVAX-based VAX 6000 series was the company's first venture into symmetric multiprocessing (SMP).

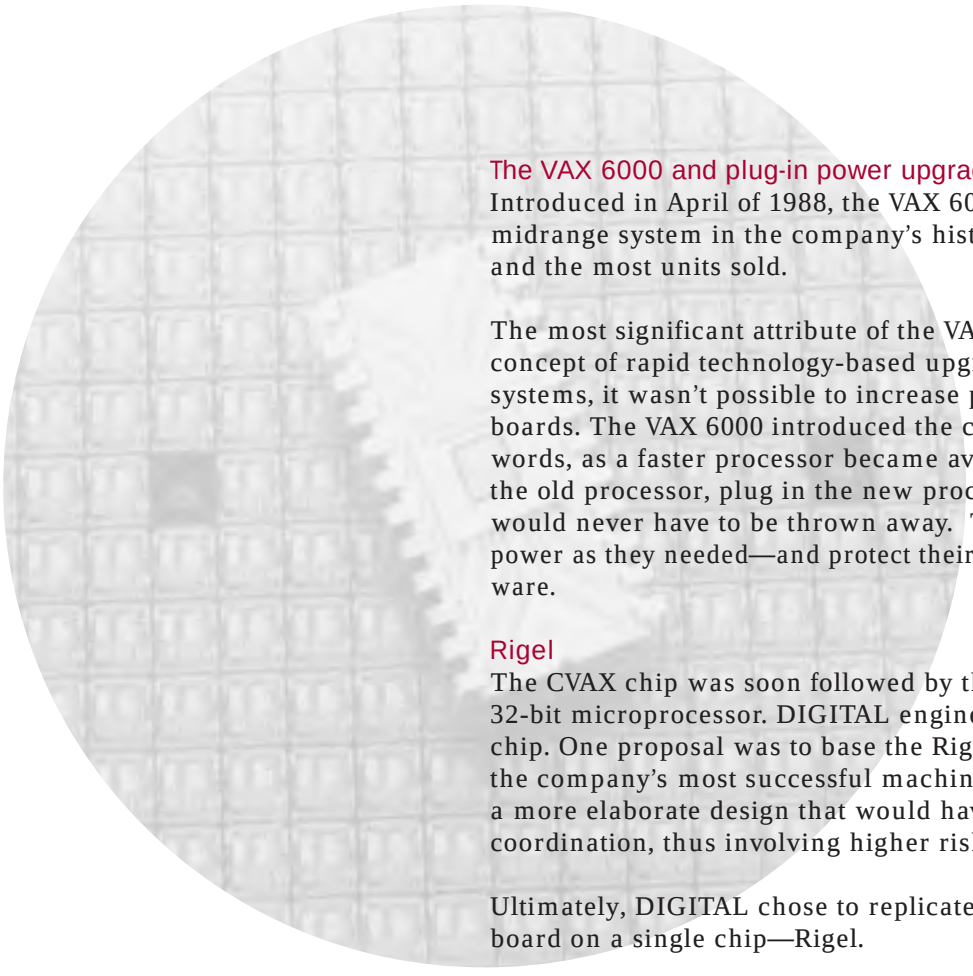
DIGITAL believed that SMP would require tearing VMS up by the roots and starting over again. However, DIGITAL engineers found a simpler approach. The places where VMS did interlocking against interrupts were located and determined to be the points where VMS had to put in a more formal lock structure for multiprocessors. A very small team produced a working prototype of VMS SMP in nine months.

SMP was introduced in VMS version 5.0, announced April 1988.

"The sense I always had was that there were four key technical visionaries at the beginning of MicroVAX: Dave Cutler, with his creation of the MicroVAX I system for early software development; Bob Supnik, who headed up MicroVAX chip development and also wrote the microcode; Jesse Lipcon who headed up MicroVAX II Server Development; and Dick Hustvedt, who drove the MicroVMS Software Strategy."

—Jay Nichols

Computer Special Systems, Manager of Engineering



The VAX 6000 and plug-in power upgrades

Introduced in April of 1988, the VAX 6000 system was the most successful midrange system in the company's history, with the fastest time-to-market and the most units sold.

The most significant attribute of the VAX 6000 was that it introduced the concept of rapid technology-based upgrades. With previous DIGITAL systems, it wasn't possible to increase power simply by replacing processor boards. The VAX 6000 introduced the concept of plug-and-play. In other words, as a faster processor became available the customer could unplug the old processor, plug in the new processor, and the original equipment would never have to be thrown away. This allowed customers to increase power as they needed—and protect their investments in hardware and software.

Rigel

The CVAX chip was soon followed by the Rigel chip, the company's third 32-bit microprocessor. DIGITAL engineers considered two options for this chip. One proposal was to base the Rigel chip on the VAX 8800, which was the company's most successful machine. The other proposal was to produce a more elaborate design that would have required multiple chips and more coordination, thus involving higher risk but higher performance.

Ultimately, DIGITAL chose to replicate the circuit design of the 8800 CPU board on a single chip—Rigel.

The Rigel chip was manufactured in 1.5-micron CMOS technology. Introduced in July 1989, the Rigel chip shipped in the VAX 6400 system and later, in the VAX 4000 system. Rigel also included the first implementation of the vector extension of the VAX architecture.

Mariah

In October 1990, DIGITAL introduced the Mariah chip set, which shipped in the VAX 6500. An improvement on the Rigel chip set, the Mariah chip set was manufactured in 1.0-micron CMOS technology. The VAX 6500 processor delivered approximately 13 times the power of a VAX-11/780 system, per processor. The VAX 6500 systems implemented a new cache technique called write-back cache, which reduced CPU-to-memory traffic on the system bus, allowing multiprocessor systems to operate more efficiently.

NVAX

The NVAX chip was introduced in November of 1991. The company's fourth VAX microprocessor, the NVAX chip was implemented in 0.75-micron CMOS technology and shipped in the VAX 6600. The NVAX incorporated the pipelined performance of the VAX 9000 and was the fastest CISC chip of its time—delivering 30 times the CPU speed of the VAX-11/780.

The NVAX chip is the current technology used in VAX systems shipping today.

Moving at breakneck speed

Chip development at DIGITAL was remarkably speedy. The timeframe from MicroVAX to CVAX was about two and a quarter years. From CVAX to Rigel was less than two years. From Rigel to Mariah was about a year. Mariah to NVAX was 15 months.

Growing the business through silicon

The VAX chip set launched the company's product development in a new direction. In the first full fiscal year, the VAX chip business grew into a billion-dollar business. Ultimately, it grew to a two to three billion-dollar business.

When MicroVAX was introduced, less than 10 percent of the company's systems revenue came from products based on microprocessor chips. By 1990, microprocessor chips were responsible for 90 percent of the systems revenue. By the early 1990s, the DIGITAL semiconductor group was the largest and most profitable business in the company.

Major performance increases

Powered by MicroVAX chips, VAX systems increased in performance from one MIP, to 2.5 MIPS, to 7 MIPS, to 11 MIPS to over 30 MIPS in five generations of design. The VAX system had established a worldwide reputation as the fastest, highest-performance machine on the market.

DIGITAL measured the performance of its chips against the competition from the time MicroVAX was introduced. The company's goal—to produce the industry's fastest microprocessors—was achieved with CVAX, which was the fastest chip of its time.

"Reflecting on the different chip sets, from MicroVAX II through CVAX, Rigel and NVAX — the primary focus of architectural energy was processor performance, with the NVAX architecture pushing creativity to its limits."

—Jay Nichols
Computer Special Systems,
Manager of Engineering

VAX 9000 chip manufacturing clean room.

CHAPTER X Building the Bridge to Alpha

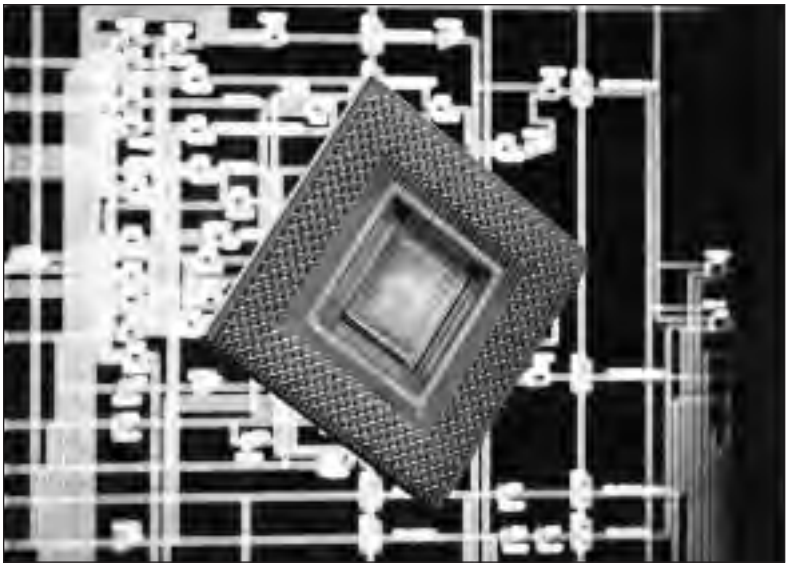
Prism: VMS on RISC technology

DIGITAL began working on RISC technology in 1986 when Jack Smith, VP of Operations, tapped Dave Cutler on the shoulder and said, "You will be RISC Czar for DIGITAL. Organize a program." The program, code named Prism, was to develop the company's RISC machine. Its operating system would embody the next generation of design principles and have a compatibility layer for UNIX and VMS.

The team discussed such issues as: Should it be 32 or 64 bits? Should it be targeted for the commercial or technical market? The proposed implementation of Prism was an ECL machine. While known for being particularly power-hungry, ECL was the fastest semiconductor technology available during the 1970s and '80s. The VAX 8600, 8800, and 9000 series were built using ECL. However, with the NVAX chip in 1991, CMOS technology surpassed ECL's performance at a much lower power cost.

There were already two other ECL projects underway, the VAX 9000 and a successor to the VAX 8800. Would these machines be competitive or overlapping in the marketplace? What would be their comparative performance? What about cost? Obviously, it made no sense for DIGITAL to be developing three projects of the same magnitude.

In April of 1988, a group of workstation engineers made a counter proposal to get DIGITAL into the technical computing market via existing RISC technology. They started building a RISC workstation that would run ULTRIX—the company's port of UNIX—using microprocessors from a startup company called MIPS. Prism was canceled in favor of using MIPS technology.



February 1991, DIGITAL announced Alpha, programming for the 21st century.

Porting VMS to Alpha

While the Alpha architecture was being designed, the principal piece of work needing attention was VMS. Nancy Kronenberg led the VMS challenge, which seemed rather formidable. VMS contained more than 10 million lines of code—much of it written in VAX assembly code. It was coded to all the features of the VAX instruction set, and it was unclear how to separate VMS from VAX.

Through careful analysis, Nancy's team discovered that even though VMS looked monolithic, it was a well-structured operating system with a machine-dependent and a machine-independent layer. The machine-dependent layer could be ported and the machine-independent layer would follow. The team invented solutions such as the macro compiler, which treated VAX macro code as a higher level language and compiled it to Alpha.

In 1991, the final task—porting VMS to Alpha—fell to Jean Proulx and her team who accomplished the porting challenge brilliantly. VMS was Alpha-ready!

The speedy MicroPrism chip

Meanwhile, the semiconductor group in Hudson, Massachusetts, was working on the MicroPrism chip—a single-chip CMOS implementation of the Prism architecture. After the Prism program was canceled, the Hudson group was allowed to complete the MicroPrism chip, since it was very near completion. The small batch of MicroPrism chips produced ran successfully at 45 MHz—a speed that was unheard of at that time, and that far surpassed the performance of any RISC chip available on the market.

The birth of Alpha

The Prism program was significant for DIGITAL because of the legacy it left for Alpha—the company's future 64-bit technology. A small team formed in July of 1988 to determine what RISC technology could do for VMS. First the team asked themselves, "What do we have to do to get VMS up on RISC?" Then they turned the question around. "If the customers have to go through a transition, how do we get the maximum performance and minimize their pain?" That's when Alpha was born.

Alpha was very much the "son of Prism." The primary changes made to produce Alpha were for VMS compatibility. The original Prism design had serious compatibility problems with the VAX and VMS in two areas—numerical data types and privileged architecture.

The Alpha architecture was built on four premises. First, it had to be a very long-lived architecture. Second, it had to deliver the highest performance for both technical and commercial applications. Third, it had to be very scalable in terms of both implementation size and range of systems supported. And fourth, it had to support customers' applications and operating systems, VMS and UNIX. Windows NT had not yet entered the scene.

"We've done a lot of work to make sure that moving from VAX to Alpha is very easy. If a customer doesn't want to move their entire environment to Alpha at one time, they don't have to. We support mixed architecture clusters, which allows VAX and Alpha to run together in a cluster. They can stay on VAX as long as they'd like to. We'll continue to do releases of OpenVMS Alpha and OpenVMS VAX at the same time."

—Rich Marcello
Vice President, OpenVMS Systems Software Group

Asking the right questions

The team made decisions about the product by asking questions. “If the objective is to create a 20-year target, will a 32-bit machine be viable 20 years from now?” The answer, “No.” So it became a 64-bit machine. That part was easy. “What would it take to drive performance over 20 years via clock rate improvements, multiple instruction issues, internal organization, and multi-processing?” The architecture reflects exactly what it takes to do that.

They looked at the issue of scalability from small to large, and therefore had a model of what could be the minimum implementation. Research done on Prism helped the team solve operating system, data flexibility, and code handling issues. Another critical development issue was the notion of VAX-to-Alpha binary translation to ensure a smooth migration for DIGITAL customers who would eventually move to 64-bit computing.

Ken Olsen visits manufacturing facility during the power-up of the first Alpha system.

Determining Alpha’s building blocks

The basic building blocks of Alpha were: an architectural commitment to move to 64 bits with the highest levels of performance that would preserve DIGITAL customers’ investments, a matching commitment on VMS to preserve customers’ operating environments, and silicon that would stand the industry on its ear. The design team studied high-speed implementation techniques discovered through the MicroPrism project. The team concluded that a chip could be built that would run two to three times faster than anything else in the industry—one that would run at 200 MHz when competitors were talking about 50 MHz.

Bringing the company on board

The Alpha program ran as a loose confederation of people who shared the vision of putting DIGITAL back on top with leadership systems. There was an Alpha project in VMS, and an Alpha project in DIGITAL semiconductor group. These team members went out and proselytized to the rest of the company and convinced it group by group to participate, until eventually the Alpha program consumed roughly a third of the company’s engineering resources.

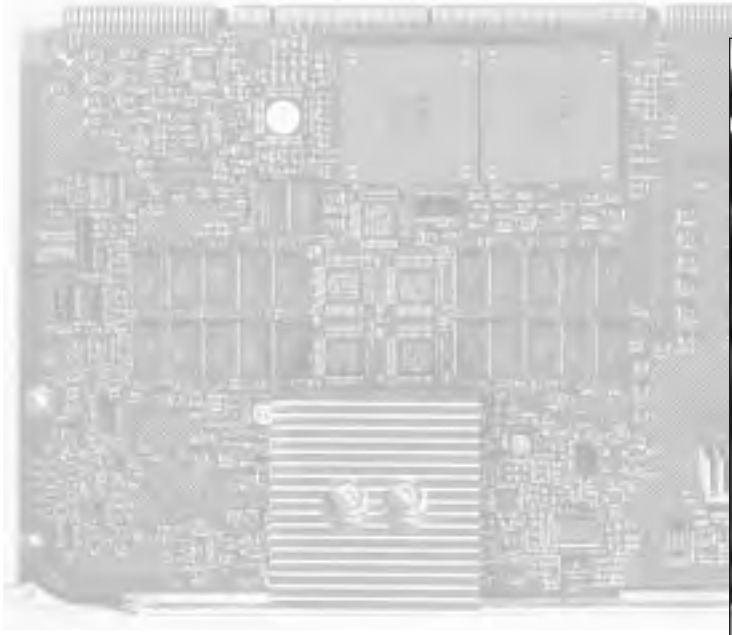
Getting business partners on board

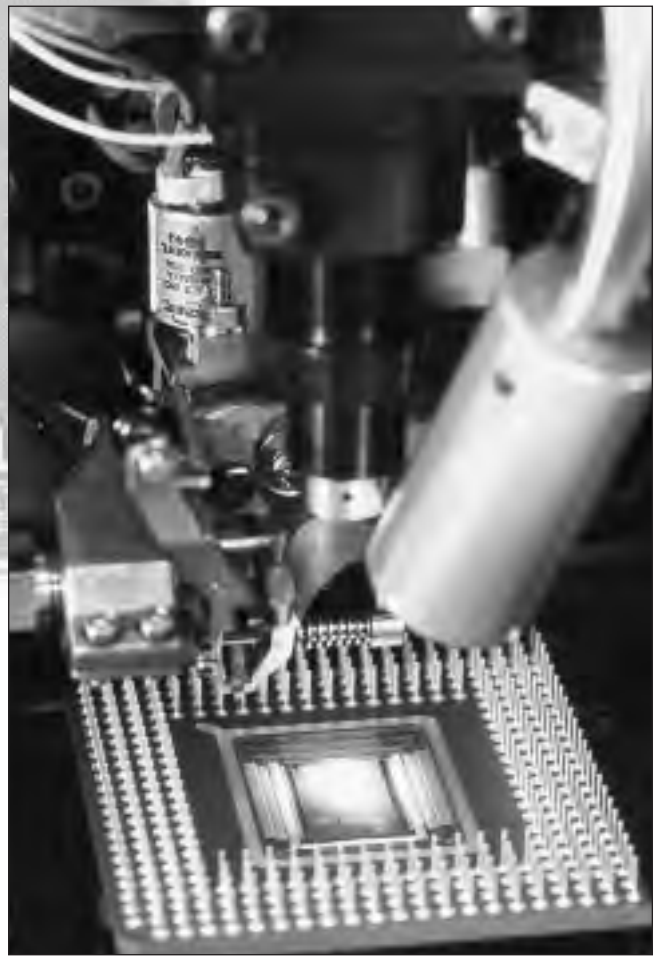
In order for the partners at DIGITAL to take advantage of this record-breaking technology, Vice President Bill Demmer set up the Alpha AXP Partners Office six months before the announcement so that the company's business partners would be signed up and on board at announcement time. Early Alpha partners included Andersen Consulting, Cray Research, Encore, Kubota Pacific Computer, Raytheon, and Olivetti.

By September 1992, DIGITAL had shipped more than 1,000 Alpha systems to software developers.

Bringing on the customers

To meet customer needs, DIGITAL developed programs and services to support this new technology. For two years prior to the 64-bit announcement, a group of customers met regularly to review plans for the Alpha AXP program. The group, the ALPHA AXP Customer End User Advisory, included representatives from communications, manufacturing, technology, government, the university community, and other potential markets for the Alpha technology.





With peak execution rates of up to 2 BIPS, these top-performing Alpha 21164 chips push the performance envelope for visual computing applications such as video conferencing, 3-D modeling, video editing, multimedia authoring, image rendering, and animation.

Chapter XI AlphaChip—The 64-bit Breakthrough

On February 25, 1992, DIGITAL introduced another significant technology advance: the world's first 64-bit architecture. This revolutionary architecture was based on the AlphaChip 64-bit RISC technology and 150 MHz DECchip 21064 microprocessor.

Announcing the Alpha AXP family of systems

In November 1992, DIGITAL announced a complete family of ALPHA AXP systems. It included ALPHA AXP workstations, departmental servers, data center servers, mainframe-class servers, and system software, as well as services, layered products, peripherals, and upgrade programs. Four hundred software partners announced availability dates for nearly 900 Alpha applications.

Alpha AXP achieved record-breaking status. In April, ALPHA AXP performed the world's fastest sort and fastest transaction processing to date. The company announced the industry's highest-performance workstations in the less than $5,000, $10,000, and $15,000 price categories.

Above to the left; Bill Demmer, VP, discusses AlphaChip in 1992 announcement.
Above; Ken Olsen visits Alpha manufacturing facility.

"Today is the beginning of a new revolution in computing. With nearly limitless 64-bit computing power and the applications of three major operating systems, the path ahead leads wherever the imagination can take it. ALPHA AXP computing will enable customers to invest in profitable new ways to serve people."

—Robert B. Palmer
Chairman, President and CEO of Digital Equipment Corporation
Q2 FY93

VAX—enjoying life after Alpha

Many people thought that after DIGITAL announced its family of 64-bit Alpha computers, there would be no more VAX systems introduced. Not so.

In 1995, DIGITAL announced the Catamount project, which was responsible for producing the VAX 4000 Model 108 system and MicroVAX 3100 Model 88 and 98 systems. DIGITAL added new functionality into the product set, including an increased memory capacity by a factor of four. Engineers increased the memory capacity in response to customer requests for more memory to meet prior increases in CPU performance. The Catamount products were designed to be both rack mountable and used on the desktop. The focus was lowering the customers' cost of ownership and allowing customers to take advantage of lower cost memory and storage technology.

Beyond the product enhancements, the real significance of this new line of VAX systems was the fact that DIGITAL was continuing to make investments to support the company's Installed Base of VAX customers.

Ken Olsen and Bob Palmer discuss future technology.

Destination Alpha: Removing the barriers

To help ensure that customers have a risk-free transition from VAX systems to Alpha systems, DIGITAL launched the Destination Alpha program in 1995. Under this program, DIGITAL opened 34 application migration centers around the globe to help customers migrate their applications. In addition, an engineering hotline is available to help customers resolve their most critical migration issues.

DIGITAL also developed a program called Project Navigator that addresses any financial or technical barriers that customers may face. Through these programs and services, DIGITAL has provided customers with a smooth transition to the Alpha platform.

"When we were designing the Destination Alpha Program, we realized that we needed to develop customized solutions so customers could move from VAX to Alpha at their own pace."

—Janet Darden
Destination Alpha Program Manager

VMS becomes OpenVMS

The 64-bit Alpha system became the most powerful system in the industry. Major developments included strategic Alpha software combined with the availability of Microsoft's Windows NT on the Alpha platform. During this time frame, DIGITAL officially changed the name of VMS to OpenVMS to reflect the ease of portability and openness of this operating system. With OpenVMS, VMS now supported the widely accepted POSIX standards of the IEEE. The VAX operating system was also "branded" by X/Open, the non-profit consortium of many of the world's major information system suppliers.

OpenVMS supports key standards such as OSF/Motif, POSIX, XPG4, and the OSF Distributed Computing Environment (DCE). Extensive support for standards in the operating system helps when building an open systems environment using OpenVMS as the base. Supported open systems standards include networking, data, document, systems, software development, and user interface. OpenVMS supports all major open systems standards, including those for networking, data, document, systems, software development, and user interface.

VMS becomes OpenVMS.

With this name change came the introduction of 13 Alpha-ready OpenVMS VAX systems and servers. Alpha-ready was the term coined to indicate that these VAX machines were easily upgraded to incorporate the new 64-bit technology.

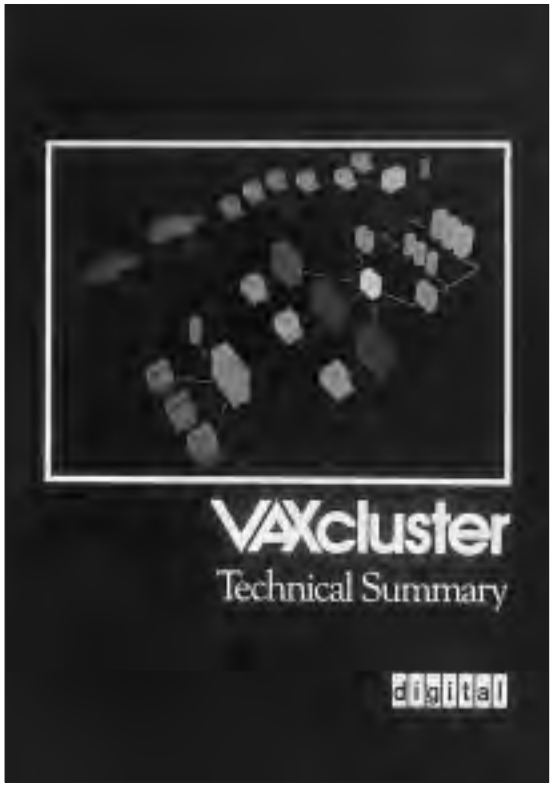
In February 1993, the company shipped 26 OpenVMS ALPHA AXP products ahead of schedule to provide a software suite for developers, system integrators, and end users. In May, more than 2,000 applications were available for OpenVMS ALPHA AXP.

ALPHA AXP family members.

CHAPTER XII Inaugurating the King of Clusters

Throughout the industry, increasing demands were placed on computers as customers' applications grew. One way to provide more computing power was to build bigger, faster systems up to the current technical limits. DIGITAL came up with an alternate solution that provided more power—without sacrificing the benefits of distributed computing customers wanted. That ideal was clustering.

Cluster computing, invented by DIGITAL, has become a widely accepted alternative method of providing higher system availability and scalability using mainstream computing products than can be provided by a single computer system. In fact, in the eyes of our customers, DIGITAL's OpenVMS Clusters became the standard by which all other clusters are measured. Cluster computing provides a dimension of scalability as an alternative to extending or upgrading a single system, and allows older installed systems to be coupled into the cluster to provide an economical way to increase computing power and deliver higher availability of data and applications.



VAXclusters was the first clustering capability in the industry! VAXclusters tied VAX processors together, which allowed VAX computers to operate as a single system, extending the characteristics of VAX to high-availability applications.

Introducing VAXclusters

In May 1983, DIGITAL announced VAXclusters. VAXclusters tied VAX processors together in a loose processor coupling that allowed VAX computers to operate as a single system—extending VAX characteristics to high-capacity and high-availability applications.

OpenVMS Clusters

Over the years, VAXclusters evolved to VMSclusters, and today are OpenVMS Clusters for VAX and Alpha Systems. OpenVMS Clusters are unparalleled in the industry today. Most of the world's stock exchanges and electronic funds transfer activities run on OpenVMS Clusters.

An OpenVMS Cluster is a highly integrated organization of VAX and Alpha systems, application and systems software, and storage devices. Systems sized from the desktop to the datacenter can be connected into an OpenVMS Cluster. OpenVMS Cluster software enables the system to work an easy-to-manage virtual system that shares printing resources, storage devices, and print and batch queues.

OpenVMS Clusters offer the best benefits of both centralized and distributed systems with the added benefit of power that can surpass that of a mainframe—at a fraction of the cost. And they can be added to or divided as customer requirements dictate.

"(Open)VMS remains King of the Clusters. DIGITAL's technology is still the high bar against which other clustering schemes are measured."

—Datamation, August 15, 1995

Unparalleled benefits of OpenVMS Clusters

High availability—Guaranteed access to data and applications due to multiple connected systems.

Easy growth—A cluster can contain anywhere from 2 to 96 systems, depending on the changing needs of the business.

Shared access—All users can easily access applications, storage devices, and printers within a cluster.

Easy to manage—An entire cluster can be managed as a single system, remotely or on-site.

Investment protection—Existing systems can be integrated into the same cluster along with new VAX and Alpha technology.

Multiple interconnects—Clusters can be configured using many different interconnects, including CI, DSSI, SCSI, NI, and FDDI.

Automatic caching—Enhances performance and reduces I/O activity.

DECamds—Optional availability management tool allows monitoring and managing resources availability in real-time.

Lock manager services—Allows reliable access to any resource or file, without the danger of losing or corrupting the file and its data.

Local Area VAXclusters

In 1986, DIGITAL introduced Local Area VAXclusters, which extended distributed computing capability to the workgroup and used a standard Ethernet network as the cluster interconnect.

With Local Area VAXclusters, VMS extended its cluster technology to the NI. The CI interface was a large, expensive controller available only on large, expensive VAX systems. That fact, plus the limit at that time of 16 systems on a CI, limited CI clusters to the large "computer room" VAX systems. Also, all cluster-accessible storage had to be connected directly to the CI. However, the advent of the MicroVAX and VAX workstations (concurrent with clusters in 1984) created the demand to connect larger numbers of smaller VMS systems into the cluster.

To answer this demand, DIGITAL modified VMS to allow the cluster communication protocols to operate over the NI, which was the only interconnect available on small VAX systems. In addition, software was introduced to allow all storage devices on the cluster to be served to all cluster members. This allowed the NI cluster members access to the HSC-based storage even though they had no direct connection.

Local area VAXCluster systems extended VAXCluster technology to Ethernet. Bringing the software advantages of the VAXCluster environment to the MicroVAX II and VAXstation II systems.

Supporting clusters via more interconnects

In the years since, DIGITAL added more interconnects to support cluster connections:

- **FDDI**—an industry-standard, optical fiber interconnect approximately ten times faster than an Ethernet. The FDDI also provided access to bridges to a number of common carrier communications media, allowing cluster connections over great distances.
- **DSSI**—a low-cost CI that allows connection of up to three VMS systems and a limited number of directly attached disks.
- **Memory Channel**—a very fast direct memory access path between VMS systems located close together.

Bring on more mules

An analogy can be drawn from farming. To do more plowing, the farmer can work his mule longer and harder, or trade in his old mule for a bigger and stronger one. Another option is to buy a second mule and team them together.

Clustering, or joining computers together to share a task, was like hooking up a second mule. And a third. And a fourth. Customers could keep their existing investments and grow from there.

"The high-availability characteristics of OpenVMS and clusters are very important to us...the scalability and clustering capability of OpenVMS allow us to provide our clients with technology as they need it."

—Scott Fancher
Vice President and Product Line Executive, Cerner Corporation

Fault Tolerant and Disaster Tolerant Systems

Just as clustering was an outgrowth of networking, fault tolerant and disaster tolerant systems were an outgrowth of clustering.

Clusters offer high availability; they are not fault tolerant. Clustering enabled the development of fault tolerant and disaster tolerant systems by providing availability that guaranteed 24x365 days of service. Fault tolerant systems provide what is considered five nines of availability, meaning that the system would be available 99.999 percent of the time. Fault tolerant systems allow applications to continue computing in the event of equipment failure. The system does not have to wait to restart or boot after encountering a failure. Rather, the failed piece of equipment drops off while the redundant pair continues to run from the time of the fault. In certain situations this kind of availability is needed—such as 9-1-1 emergency services, financial/stock market transactions, air traffic control, and nuclear reactor monitoring. Fault tolerant applications are needed where the consequences are disastrous if the computer is out for a few minutes or more. Fault tolerant failover occurs in a minute or less, with no loss of data.

VAXft 3000 Announcement

Multi-site clustered systems are used in disaster tolerant applications. Disaster tolerant systems are set up to prepare for man-made or environmental disasters including terrorism, fires, earthquakes, floods, etc. All these situations have the potential to take out a computer room. If there is a back-up system that can send out data to another location, the systems will remain functioning to prevent losses of data and business that an interruption would cause. Two fault tolerant systems clustered together in two different locations provide site diversity and automatic failover to a site distanced from the disaster. If one site goes down, the other takes over and continues operating—without missing a beat.

OpenVMS Clusters continue to reign as the King of Clusters

Today, more than 65,000 OpenVMS Cluster systems are found at the heart of continuous computing solutions for such critical applications as stock exchanges, electronic funds transfers, healthcare, telecommunications, and process manufacturing. No other solution can match OpenVMS Cluster systems when it comes to our over 14 years of providing a continuous computing environment. Only OpenVMS Cluster systems can span up to 500 miles to enable continuous operation through even large scale natural or man-made disasters ensuring optimal data and transaction integrity and fast recovery. OpenVMS Cluster system support “rolling upgrades”, enabling system processors, boards, peripherals, operating software, databases, and program modules to be replaced, upgrades, or updated without interrupting the operation.

The Fault Tolerant Group

CHAPTER XIII OpenVMS Today

The OpenVMS Ambassadors Program, formerly known as OpenVMS Partners, is an international program that provides a liaison between customers and the company's OpenVMS Systems Software Group and expert field organizations in sales support, systems integration, Technical Consulting Center (TCC), and benchmarking. The OpenVMS Ambassadors provide valuable customer feedback, and because of their technical expertise can relay information in engineering terms and can make recommendations about what types of changes are needed from the customers' perspective. The Ambassadors must meet three essential criteria: technical competence, commitment, and a high level of communication skills.

OpenVMS is a general purpose, multi-user operating system that runs in both production and development environments. OpenVMS Alpha supports the DIGITAL Alpha series of computers, while OpenVMS VAX supports the VAX series of computers. The software supports industry standards for facilitating application portability and interoperability. It also supports symmetrical multiprocessing (SMP) support for multiprocessing Alpha and VAX systems.

An integral part of three-tier computing

Today, the core of the OpenVMS strategy is to leverage the inherent affinity between Windows NT and OpenVMS by combining the unequaled strengths of OpenVMS with the emerging power and application library of Windows NT in a seamless computing environment.

OpenVMS is the environment of choice in the most demanding of continuous computing situations. The high levels of availability, integrity, security, and scalability of OpenVMS make it a natural unlimited high-end for Windows NT in a three-tier client/server environment. OpenVMS is the number one operating system in healthcare today. It also enjoys a major presence in the financial, funds transfer, and stock exchange industries, as well as manufacturing, education, and government.

OpenVMS Ambassador Team (Business Partners)

"The OpenVMS operating system environment holds a special place in the computer industry. It was the centerpiece of the minicomputer revolution, the first operating system to prove that scaling from desktop to data center was practical, and the first to demonstrate that clustered systems could achieve levels of availability well beyond mainframes or 'fault tolerant' systems. It was, and continues to be, a huge market success."

—Wes Melling

VP of Windows NT and OpenVMS Systems Group

Unparalleled availability

OpenVMS provides immunity to planned and unplanned downtime with proven 24x365 availability, including disaster-tolerant multi-site clusters spanning 500 miles.

OpenVMS systems scale to meet the performance, availability, and data requirements of the largest enterprise applications through 64-bit, Very Large Memory (VLM), and Very Large Data Base (VLDB) support, and clusters of up to 96 nodes.

OpenVMS provides enhanced performance, clustering flexibility, easy Internet connection, and 64-bit VLM for business-critical applications. New features have been incorporated to further performance in OpenVMS Clustering and to improve system management. Memory Channel clusters, extended VLM capability, cluster failover, and the OpenVMS Internet Product Suite are also provided by OpenVMS.

"It is important to emphasize the significance of the Installed Base to DIGITAL. With over 700,000 systems installed worldwide, it is more critical than ever before for us to continue to nurture the base.

We brought a bright future to our OpenVMS Installed Base customers with the Affinity strategy. OpenVMS continues to be one of three strategic platforms from DIGITAL."

—Wally Cole
VP, Installed Base Marketing

Enhanced support for clustering

OpenVMS Cluster technology enables customers to configure disaster-tolerant multi-site clusters located up to 500 miles (800 kilometers) apart.

OpenVMS provides features specifically designed to improve performance and expand OpenVMS Cluster configuration flexibility. OpenVMS supports mixed architecture clusters and allows customers to connect up to 96 Alpha and VAX systems and storage controllers to share common data and resources across systems, as well as architectures. OpenVMS Cluster systems can utilize FDDI, CI, DSSI, Ethernet, and Mixed-interconnect transports.

Two powerful features of OpenVMS Clusters are Memory Channel and the Business Recovery Server. Memory Channel comprises a high-performance interconnect technology for PCI-based Alpha systems that improves OpenVMS Cluster performance and reduces costs. Business Recovery Server Cluster support allows businesses to withstand disasters—floods, fires, earthquakes—at any site, without loss of access to data or applications.

OpenVMS Cluster systems can be managed centrally, as a single system, providing a single domain for data, users, queues, and security.

"Marketing for OpenVMS is really fun activity. We have the most loyal, most enthusiastic customer groups out there. They appreciate the technology. They appreciate the ease of use. They appreciate the value of an operating system that has become tried and true over a number of years and has evolved to the state where many of the world's largest banks, stock exchanges, healthcare organizations, and production manufacturing environments are trusting their business to the true 24x365 capabilities of OpenVMS."

—Mary Ellen Fortier
Director, OpenVMS Marketing

Supporting 64-bit environments

In November 1995, at DECUS, DIGITAL announced OpenVMS Version 7.0—supporting 64-bit virtual addressing. 64-bits of address space is 18 exabytes. That's four billion times the 32-bit address space of four billion bytes. Using 64-bit addressing allows developers to map large amounts of data into memory to provide high levels of performance and to support very large memory systems.

The current Alpha memory management architecture allows actual address space usage of eight terabytes. On the VAX, only half the address space is available for applications (2GB), so the currently available Alpha address space is 4,000 times that on the VAX.

As ever larger memory becomes available, the Alpha memory management architecture can be extended to support more of the theoretical maximum of 18 exabytes. This was the largest incremental release in the OpenVMS operating system since the introduction of VMSclusters.

CHAPTER XIV Serving Customers Worldwide

Supporting the family–global services

DIGITAL realized that a key factor in the success of VAX and VMS was customer services. Almost from its founding, the company has supported customers worldwide from strategically located field service facilities.

Educational services provide software and hardware training—providing DIGITAL customers with the necessary skills to implement and work with the company's system effectively.

DIGITAL developed its Services group to ensure that the first release of VMS could be supported by field support services. This group formulated strategies for support of VMS, and learned the software in depth to be able to support it and train people in the unique features of the new software. A back-up support group called VAXworks was also formed to address customer needs. The VAXworks group received phone calls and telexes from people all over the world.

DIGITAL set out to have the best support and field service operations as well as the best education and training organization. These services have always been a vital part of the company's success and have contributed greatly to the business.

Field service engineer repairing customer CPU board.

Bringing in the voice of the customer

From its inception, DIGITAL has believed that two-way customer communication was necessary to ensure that the company was building products to solve real-world needs. That strategy exists today, as DIGITAL takes a comprehensive approach to working with customers at all levels of their organizations.

DIGITAL listens to customers through a variety of forums, including:

Customer visits—DIGITAL makes more than 500 visits to OpenVMS customers annually.

Technical Direction Forums—Twice a year, DIGITAL presents new strategies and technologies to 12 top customers at the Director of MIS level. This feedback has a direct impact on future directions.

OpenVMS Executive Counsel—Every six months, DIGITAL meets with 35-40 CIOs in various customer organizations to look at overall strategies of business and direction.

DECUS—Founded in 1961, the Digital Equipment Computer Users Society (DECUS) is an opportunity for people who work with OpenVMS on a day-to-day basis to receive training on all technologies and provide valuable feedback.

"A big part of the success at DIGITAL was the support and the service. We gave enormous service to the customer. And without that, even VAX and VMS wouldn't have been so successful."

—Ken Olsen, 1997

Service strategy today

The range of DIGITAL Service spans the spectrum from systems integration to hardware and software maintenance. The company's service effort focuses on three areas. The first area is to support the company's strategic growth areas: high-performance 64-bit computing, NT across the enterprise, and Internet connectivity. The second area is multivendor service. DIGITAL is the only major vendor that has declared a vendor neutral strategy. The third area is value-added services and innovation in the marketplace.

DIGITAL has an investment in global resources and infrastructure that's second to none in the industry. The worldwide DIGITAL Services Organization—between its Multivendor Customer Services and Systems Integration Organization—includes more than 25,000 Service Professionals worldwide and over 450 locations around the world. At the company's Solution Centers, System Integration Specialists and Network Consultants help customers successfully solve their most challenging information technology problems.

Strategic partnerships

To ensure continued growth and to meet the changing business needs of its customers, DIGITAL has established strategic partnerships with industry-leading companies such as Microsoft Corporation, Oracle Corporation, and others.

"We find that customers are using information technology to get greater access to their data. They want to spend more time on the analysis of this information and its distribution using the Internet to create competitive advantage. They don't want to spend a lot of time becoming information technology experts. More and more they're relying on key service partners to take responsibility for the management of this information infrastructure."

—John Rando
VP and General Manager,
Multivendor Customer Services Organization

Getting the bugs out. Or, using cockroaches as semiconductors

A VAX-11/780 installed at the Carling Brewery was crashing several times a day with no pattern at all. Field services reps had replaced everything and they couldn't figure it out. Every time the machine crashed, accidents would happen, usually spilling large quantities of beer.

One day, a software specialist was in the booth with the machine pouring over the last dump. All of a sudden there was the familiar rhythm of another crash. He looked out the windows and saw people scurrying for cover. The capping machine had run amuck and was spitting out bottle cap blanks, which in their raw state are like little two-inch diameter, razor-sharp, aluminum frisbees. The software specialist couldn't take it anymore. He walked over to the VAX-11/780 and kicked the front panel as hard as he could. A bunch of cockroaches came scurrying out.

Naturally, cockroaches are attracted to beer dregs. And it was warm and dry inside the machine, so they moved in. He figured that cockroaches are at least somewhat conductive. As the insects ran up and down the backplane, every once and a while one of them would get two legs across—a pair of contacts. And the machine would crash.

The software specialist went out to the local store and bought Roach Motels which installed in the bottom of the machine. The problems ended. After that, changing the Roach Motels became a part of the monthly product maintenance.

"When we were about up to Version 3 of VMS, I was at DECUS and a customer came up to me and she said "You won't believe this, but we're still running Baselevel 5 and we love it. We think it's the best thing ever. and we're never going to change it because it does just what we need." So I said, "Well, if it does just what you need, I think you're right, don't ever change it."

—Kathy Morse
VMS Engineer

VAX: Built to last

780 drops off a forklift and lives

In 1978, a VAX-11/780 was shipped to the National Computer Conference in Anaheim. At the loading dock it dropped off of a forklift—which was hard on something this big. A replacement machine from a nearby local office was brought to the show, and the carcass of the dropped system was shipped back to New England. The engineers took it apart, straightened the frame, and replaced the backplane. Other than that, it worked perfectly, and was put in service for years, many years. It was called the Phoenix.

Another VAX-11/780 slams into the side of a building and keeps on ticking

Another VAX-11/780 was shipped to a customer in Washington, D.C. It was too big for the elevator, so the customer decided to lift it on a crane and swing it in through a window. Instead of going into the open window, the system slammed into the side of a building. The machine looked very damaged.

At that time, VAX-11/780 systems were on a six-month backlog and the customers didn't want to wait for a new one. So the feild service engineers put new skins on it and replaced a slightly bent backplane, offering a replacement if necessary. That machine always worked perfectly. And the customer was delighted because he got a six-month lead.

"The customers were members of the family, and there was a strong dialogue at all levels between engineers and customers. We spent a lot of time hanging around listening to customers, and DECUS was very active and effective as a lobbying committee for new product requirements. We built what the customers told us they needed."

—Larry Portner
VP of Software Engineering

Top; Mastermind of the VAXbar, Vance Haemmerle. Bottom; Old VAX-11/780 systems never die, they just find new ways to serve humanity.

"The VAX-11/780 has always held a lot of sentimental value for me. Like your first love, you never forget your first computer. An VAX-11/780 was the first computer I programmed on in the early '80s. My very first program, a test of a FORTRAN subroutine, I jokingly called 'FAST.EXE' so that I could 'RUN FAST' under VMS. Noteworthy because it was the first in the very popular VAX line of computers from DIGITAL, the VAX-11/780 is also one of the few computers that can actually be considered a classic. Not only did it play a pivotal role in the mini-computer revolution, but it also evolved into a standard.

So when a VAX-11/780 donated to the HACKS computer club was on the loading dock headed for the dump after its CPU and memory boards were stripped for parts to get two other 780s working, I persuaded fellow HACKS members to let me have it and find another use for it. My ideas were either a bookcase for the 'Grey Wall' of VMS books, or a wetbar. Since I didn't have my own copy of the VMS documentation at that time, the wetbar idea was the obvious choice!

—Vance Haemmerle

DECUS—Digital Equipment Computer Users Society

DECUS, the Digital Equipment Computer Users Society, is an association of Information Technology professionals interested in the products, services, and technologies of Digital Equipment Corporation and related vendors. The Association's purpose is to promote the unimpeded exchange of information, with the goal of helping its members and their organizations to be more successful. The Chapter provides members with the means to enhance their professional development, forums for technical training, mechanisms for obtaining up-to-date information, advocacy programs, and opportunities for informal discussion and interaction with professional colleagues of like interest.

"It was extremely exciting in the early days of marketing VAX and VMS because the company was growing so fast. I remember the first DECUS that I attended. Each time I went to DECUS, the audience doubled. The first time, there were about 300 people in the audience, the next time there were 600, the next time there were 1,200, and the next time there were over 2,000. That was the momentum that we had in the market. In many cases we would learn from our customers about different applications using 32-bit."

—Marion Dancy
VP of Marketing, UNIX and OpenVMS,
Systems Business Unit

CHAPTER XV The Affinity Program

Bill Gates and Robert Palmer.

Leveraging the natural affinity of OpenVMS and Windows NT

On May 8, 1995, at DECUS in Washington, D.C., Digital Equipment Corporation and Microsoft Corporation announced the Affinity for OpenVMS Program to help customers implement the complementary strengths of OpenVMS and Windows NT in a three-tier client/server environment. OpenVMS provides the ultimate high-end, tools, and applications to ensure a seamless integration with Windows NT.

This integrated systems environment brings the bulletproof capabilities of OpenVMS to the world of Windows NT applications. The program includes new software, tools, middleware, and services from DIGITAL and its partners that build on the natural affinity between OpenVMS and Windows NT—making it increasingly easier to develop, deploy, and manage applications across both platforms.

"When our customers had beta copies of Windows NT, they told us that it felt like they were revisiting an old friend. That's not surprising because the chief architect of both operating systems was Dave Cutler. So there is a natural affinity from a technical perspective between the two environments. Wes Melling is often quoted calling it the 'Cutler effect.'"

—Mary Ellen Fortier
Director, OpenVMS Marketing

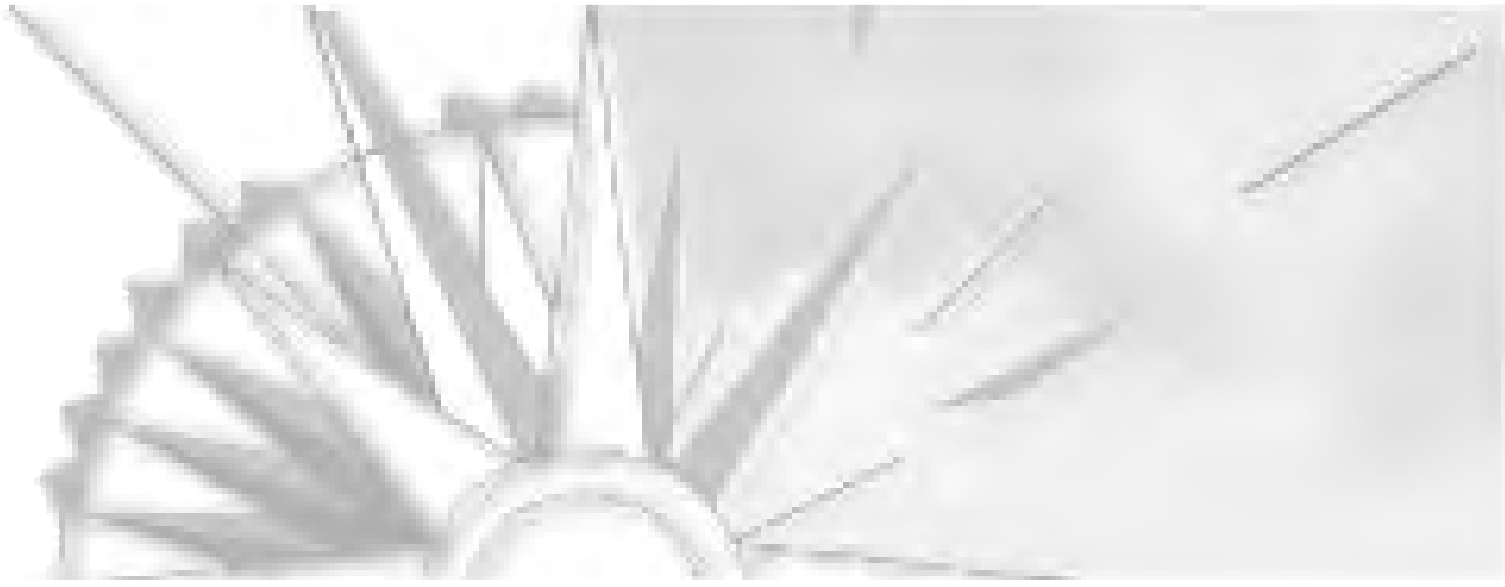
Since May of 1995, DIGITAL has consistently announced new products, capabilities, features, and services that support the OpenVMS Affinity environment.

Key examples include OpenVMS V7.0 for 64-bit computing, new products for system management, World Wide Web hosting, enterprise messaging, and application development. In addition, software vendors have responded to user demand with new applications and tools. Each year, more DIGITAL business partners are bringing application development, data warehousing, and healthcare applications to the Affinity portfolio.

In the two years since its inception, the Affinity for OpenVMS Program has helped more than 20,000 organizations around the world integrate the two platforms in three-tier client/server environments across their enterprises. Customers include worldwide banks and stock exchanges, healthcare providers, manufacturing facilities, educational institutions, government organizations, and more.

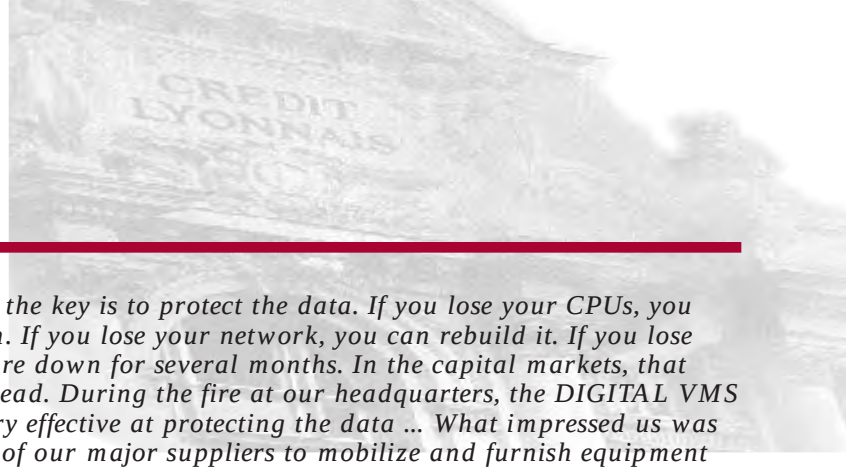
"The magnitude of what we are doing here—the clustering agreement, joint engineering, joint field teams—is much bigger than what we have done in the past with other alliances."

—Bill Gates
President and CEO, Microsoft Corporation



"With the DIGITAL Affinity for OpenVMS Program, OpenVMS and Windows NT integration brings Corning the richest operating environments. It provides superior overall functionality and the most complete set of applications and tools. We've selected Forté—the industry's premier three-tier client/server software—which draws on the strengths of both OpenVMS and Windows NT, making it possible to create and deploy applications with multi-tier, enterprise-wide functionality."

—Mark Joyce
Supervisor of Fiber Systems Engineering
Computer and Information Services, Corning, Inc.



Credit Lyonnais

"In any disaster, the key is to protect the data. If you lose your CPUs, you can replace them. If you lose your network, you can rebuild it. If you lose your data, you are down for several months. In the capital markets, that means you are dead. During the fire at our headquarters, the DIGITAL VMS Clusters were very effective at protecting the data ... What impressed us was the ability of all of our major suppliers to mobilize and furnish equipment and services. DIGITAL managed this very well indeed. They were everywhere with us."

—Patrick Hummel
IT Director Capital Markets Division, Credit Lyonnais

CHAPTER XVI Vision of the Future

"Today, OpenVMS is the most flexible and adaptable operating system on the planet. What started out as the concept of 'Starlet' in 1975 is moving into Galaxy' for the 21st century. And like the universe, there is no end in sight."

—Jesse Lipcon,
Senior VP, UNIX and OpenVMS
Systems Business Unit

Looking skyward: Galaxy

DIGITAL knows that a company's need for computing resources can fluctuate significantly for certain applications at certain times.

For example, let's consider a scenario of a system manager for a large cluster in a telecommunications company. Once every three months, a communication satellite might send enormous quantities of vital data to his receiving station. Transmission time is only two hours, and it's critical that all the data are processed immediately. He gets no second chances. But his systems are already busy crunching day-to-day information. Short of buying, or leasing, new CPUs, memory, and disks, what can he do?

That's where Galaxy will come in. DIGITAL is developing an evolution in OpenVMS functionality that will include a new model of computing that allows multiple instances of OpenVMS to execute cooperatively in a single computer. For companies looking to improve their ability to manage unpredictable, variable, or growing IT workloads, the DIGITAL Galaxy software solution for OpenVMS provides the most flexible way to dynamically reconfigure and manage system resources. Galaxy is a powerful software solution that allows system managers to easily reallocate individual CPUs or memory through a simple drag-and-drop procedure.

Alpha System, the path to the 21st Century.

Enhancing OpenVMS

After 20 years, OpenVMS still has tremendous growth potential. OpenVMS is a key component of the DIGITAL strategy to satisfy its customers' computing needs well into the next century.

"The growing importance of the Internet and corporate intranets perpetuates the value of OpenVMS. This is an area where 24x365 is essential. DIGITAL offers a variety of Web-based servers. OpenVMS is a platform that provides full reliability and availability of Internet services."

—Harry Copperman
Senior VP and General Manager Products Division

The five-pronged OpenVMS strategy

1. DIGITAL will maintain all current OpenVMS capabilities and will ease the migration to a 64-bit environment.
2. The company will continue to invest in OpenVMS development to ensure the long-term future of the operating system.
3. The disaster tolerant, 24X365 strengths of OpenVMS will continue to be enhanced.
4. The company will provide seamless integration with Windows NT.
5. OpenVMS will continue to provide an unlimited high-end to Windows NT. Current OpenVMS engineering projects are upholding the same high standards of engineering excellence that have characterized OpenVMS from its inception.

DIGITAL will continue to focus on 64-bit computing in such areas as the Internet, continuous computing, Windows NT integration, and data warehousing. To meet the demand for the integration of enterprise computing with Windows NT, DIGITAL will develop enterprise applications, visual computing, and mail and messaging that are NT-integrated. Internet business growth will support the marketplace need for the development of customer intranets, Internet commerce, and ISP/Telco support.

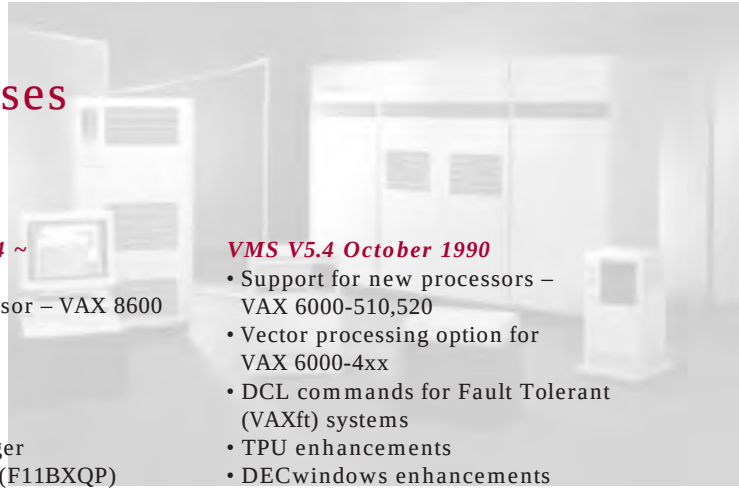
"OpenVMS plays a critical role in our customers' operations. It's a very vibrant, vital operating system, with exceptional performance and high availability."

—Bruce Claflin
Senior VP and General Manager,
Sales and Marketing

"If OpenVMS engineering continues the type of innovation we're doing now, we'll be here for another 20 years, and then we'll be asking ourselves again: what's next?"

—Steve Zalewski
Technical Director of OpenVMS systems software group

VMS TO OPENVMS: Major Releases



VMS V1 August 1978
- Multiuser, multifunction virtual memory operating system
- ODS-1 and ODS-2 file systems
- Integrated DECnet
- ANSI magtape support
- Languages
- VAX-11 FORTRAN IV-PLUS
- VAX-11 MACRO generates native code
- BASIC-PLUS 2 and COBOL
- DCL and MCR command language interpreters
- Supported hardware
- VAX-11/780 with a minimum of 256 KB of memory, up to a maximum of 2 MB
- 2 RK06 disks, or MASSBUS disk and tape
- DMC-11 communications interface
- CR11, LP11, and LA11
- DZ11 with VT52 and LA36 terminals
- Floating point accelerator

VMS V2 April 1980 ~3000 licenses
- Support for new processor – VAX-11/750
- More native languages
- EDT screen editor
- SET HOST
- MAIL, PATCH and SEARCH utilities
- Shared sequential RMS files
- Support for multiport shared memory and DR780
- Connect-to-interrupt driver
- User written system services
- VAX FORTRAN (77)

VMS V3 April 1982 ~10,000 licenses
- Support for new processors – VAX-11/730, VAX 11/725, VAX-11/782
- Asymmetric multiprocessing (ASMP) for VAX-11/782
- Support for new architectures, protocols, busses
- System communication architecture (SCS)
- Mass storage control protocol (MSCP)
- Lock management system services
- MONITOR utility for performance monitoring
- BACKUP
- Command definition utility for DCL
- Terminal autobaud detection, CTRL/T, and hangup on logout
- SPAWN and ATTACH

VAX V4 September 1984 ~ 40,000 licenses
- Support for new processor – VAX 8600 MicroVAX I/II (v4.1) VAXstation I/II (v4.1)
- VAXclusters
- Connection manager
- Distributed lock manager
- Distributed file system (F11BXQP)
- Security enhancements
- Command line editing and command recall
- Local area terminal server
- Access control lists implemented
- Cluster wide operator control
- Variable prompt strings

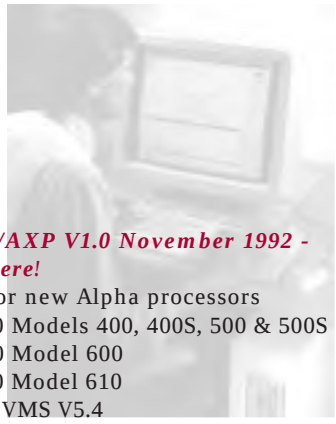
VMS V4.4
- Support for new processors – VAX 8200, VAX 8250, VAX 8300, VAX 8350 VAX 8500, VAX 8550, VAX 8700, VAX 8800
- ASMP support for VAX 83xx and VAX 88xx systems
- Cluster packages VAX 8974 & VAX 8978
- Disk volume shadowing and HSC support

VMS/ V5 May 1988
- Support for new processors – VAX 6210, 6220, 6230, 6240, 8810, 8820, 8830, 8840, 8842, VAXserver 6210, 6220
- Symmetric multiprocessing (SMP) support
- Mixed interconnect VAXclusters
- License management facility
- Terminal fallback utility
- Modularized executive
- Structured DCL: IF-THEN-ELSE, GOSUB and CALL
- System Management enhancements
- Dynamic failover of dual pathed disks
- New batch and print queue features
- AUTOGEN Feedback

DEC Windows (v5.1) VMS V5.2 September 1989, ~300,000 licenses
- Support for new processors – VAX & VAXserver 6400 series, VAXserver 3100
- Clusters of 96 nodes
- Hardware release
- V5.2-1 October 1989
- MicroVAX 3100
- VAXstation 3100 Model 38/48
- VAXstation 6000 Series 4XX

VMS V5.4 October 1990
- Support for new processors – VAX 6000-510,520
- Vector processing option for VAX 6000-4xx
- DCL commands for Fault Tolerant (VAXft) systems
- TPU enhancements
- DECwindows enhancements
- MSCP load balancing and preferred path
- Password history. Dictionary and site specific password filters
- Hardware releases
- V5.4-0A October 1990
- VAX 9000
- V5.4-1, December 1990, replaced
- VAX 9000 SMP
- VAXstation 3100 Model 76
- VAX 4000 Model 200
- VAXft models 110, 310, 410, 610, 612

VMS V5.5 November 1991
- Support for new processors – MicroVAX 3100 Models 30, 40 & 80 VAX & VAXserver 6000-6xx series VAX 4000 Models 60, 500 & 600 VAXstation 4000 Model 60 & VLC
- New queue manager
- New licensing features
- LAT enhancements (SET HOST/LAT, LATmasterfeatures)
- Phase II Shadowing (host based shadowing)
- Cluster wide tape service (TMSCP)
- New RTLs - DECthreads and BLAS fast-vector maths library
- Hardware releases
- V5.5-2HW September 1992
- MicroVAX 3100 Model 90,
- VAX 4000 Models 100 & 400
- VAX 7000 Model 600
- VAX 10000 Model 600
- VAXstation 4000 Model 90
- V5.5-2 September 1992
- VAX 7000 Models 610, 620, 630, 640, 800 through 860
- VAXstation Model 90A
- V5.5-2H4 August 1993
- MicroVAX 3100 Models 85,88,95,96
- VAX 4000 Models 100A, 105A, 106A, 108, 500A, 505A, 600A, 700A, 705A, 800A
- VAXstation 4000 Model 96
- V5.5-2HF August 1993
- VAXft Model 810



OpenVMS/AXP V1.0 November 1992 - Alpha is here!

- Support for new Alpha processors
  DEC 3000 Models 400, 400S, 500 & 500S
  DEC 4000 Model 600
  DEC 7000 Model 610
- Based on VMS V5.4
- DECmigrate for translating VAX images
- MACRO-32 compiler
- No clusters, no RMS journaling, no shadowing, no SMP

OpenVMS/VAX V6.0 June 1993

- Support for new processors – VAX 7000 Model 650/660, VAX 10000 Model 650/660
- Rationalized and Enhanced security (Level C2 compliance)
- Multiple queue managers across cluster
- HELP/MESSAGE utility
- Support for ISO 9660 CD-ROM format
- Adaptive Pool Management
- SYSMAN cluster wide SHUTDOWN and startup logging
- Cluster wide Virtual I/O cache
- Extended physical and virtual addressing
- Protected subsystems
- DECnet/OSI
- DECwindows XUI replaced by DECwindows Motif

OpenVMS/VAX V6.1 April 1994 & OpenVMS/Alpha V6.1 May 1994

- VAX and Alpha
- Support for new processors –
  AlphaServer 2100 4/200 & 4/275
  DEC 3000 Models 700 & 900
  DEC 7000 Model 710 & 7xxx
  VAX 7000 Models 7xxx
- PCSI Product installation utility (PRODUCT command)
- Shadowing and RMS Journaling for Alpha
- DECamds bundled with operating system
- CLUE Crash dump utility
- DPML standard maths library
- C++ support
- DECnet/OSI Extended Node Names

OpenVMS/VAX V6.2 May 1995 & OpenVMS/Alpha V6.2 June 1995

- Support for new processors –
  AlphaServer 2100 5/250, 8200 5/300, 8400 5/300
- Freeware V1.0 CD distributed with operating system
- Automatic foreign commands (like UNIX PATH mechanism)
- RAID subsystem support
- DCL TCP/IP functions e.g.: COPY/FTP and SMTP transport in MAIL
- OpenVMS Management Station
- SCSI clusters
- SCSI-2 Tagged Command Queuing
- BACKUP Manager - Screen oriented interface
- Hardware releases
- V6.2-1H1 (Alpha) November 1994
- AlphaServer 1000A 4/266
- AlphaBook 1
- AlphaServer 2100A 4/275, 5/250 & 5/300
- AlphaStation 255/233 & 255/300
- V6.2-1H2 (Alpha) January 1995
- AlphaServer 300 4/266
- AlphaServer 1000A 5/266, 5/333 & 5/400
- AlphaServer 4000 5/300E
- AlphaServer 4100 5/400, 5/300, 5/400 & 5/466
- AlphaServer 8200 5/440
- AlphaServer 8400 5/440
- AlphaStation 500/300, 500/400 & 500/500
- AlphaStation 600 5/266, 600 5/300 & 5/333

OpenVMS/VAX V7.0 & OpenVMS/Alpha V7.0 December 1995

- Process affinities and capabilities from DCL (Set PROCESS/AFFINITY)
- HYPERSORT High performance SORT utility (Alpha)
- Ingegrated network and internet support
- New MAIL utility (rewritten)
- Timezone and UTC support
- 64-bit addressing – new system services
- Kernel threads
- Spiralog high performance file system
- Dump file compression (Alpha)
- Wind/U – Windows Win32 API
- Fast I/O and Fast Path highly optimized I/O

OpenVMS/VAX V7.1 & OpenVMS/Alpha V7.1 December 1996

- Support for new processors –
  AlphaServer 800 5/333 & 5/400
- Pipes
- Windows NT Affinity
- PPP protocol
- Internet product suite
- Dump Off System Disk for Alpha
- External Authentication (LAN manager single signon)
- 100BaseT Fast ethernet support (Alpha)
- Memory channel high performance cluster interconnect
- Very Large Memory (VLM) support
- BACKUP API
- CDE interface for DECwindows
- 64 bit system services
- Scheduling system services

Compiled by John Gillings Sydney CSC, September 1997.

Sources: Ruth Goldenberg, Max Burnet, Steve Tolna, Thomas Schwarz, Mark Buda, Sharon Rogenmoser, Kim Kinney, Ken Blaylock, Rod Barela, Kelly Oglesby, Marie Teixeira, Michael Junge, Julian Sandoval, Mark Masias, Jason Gallant, Brian Breton, Laura Buckley, Richard Rhodes, Dave Pina, Sue Clavin, Tim Ellison, John Manning, Dave Hutchins, Paul McGrath, Judy Novey, Ian Ring, Ron Decker, Stephen Hoffman, VMS marketing, Sales Updates, Old PID material, VMS information sheet ED-31080-48, VMS SPD's, OpenVMS New Features Manuals

Edited by: Andy Goldstein

VAX and VMS History

1975

- VAX architecture committee formed with goal of "building a computer that is culturally compatible with PDP-11, but with increased address space of 32-bits." The result: VAX, the "Virtual Address eXtension" of the PDP-11's 16-bit architecture.
- VMS, the "Virtual Memory System" operating system was developed simultaneously, allowing complete integration of hardware and software.

1977

- Introduction of VAX 11-/780, the first VAX system.
- VMS V1.0 announced.

1978

- VMS V1.0 shipped. The development goal was to achieve compatibility between PDP-11 and VAX systems so information and programs could be shared.

1979

- DECnet Phase II announced.
- Fortran IV announced.

1980

- VMS V2.0 shipped, offering the industry's largest array of languages on one system.
- DECnet Phase III announced.
- VAX-11/750 introduced, the second VAX family member and the industry's first Large Scale Integration (LSI) 32-bit minicomputer.

1981

- VAX information architecture introduced, which included VAX-11, FMS, DATATRIEVE, CDD, RMS, and DBMS.

1982

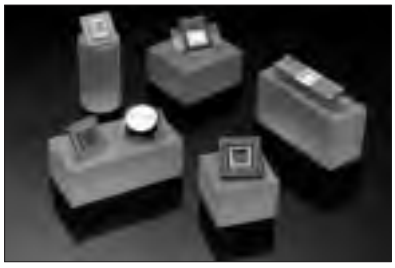
- VAX- 11/730 announced, the third VAX family member, the first low-cost VAX processor to fit on 3 hex boards, the first VAX to fit into a 10.5-inch-high rackmountable box.
- VMS V3.0 shipped.
- RA60 and RA81 disk drives shipped.

1983

- DIGITAL announced VAXclusters: the capability of tying VAX processors together in a loose processor coupling that allowed multiple VAX systems to operate as a single system.
- VAX-11/725 announced.
- CI connectivity introduced.

1984

- VAX-11/785 introduced, the most powerful single VAX computer to date. CPU cycle time was 133ns, 50% faster than the 200ns cycle time of the VAX-11/780.
- VMS V4.0 announced.
- VAX 8600 announced, the first of a new generation of VAX systems. Offered up to 4.2 times the performance of the VAX-11/780; increased I/O capability while maintaining I/O subsystem compatibility with the VAX-11/780 and VAX-11/785 systems.
- VAXstation I announced, DIGITAL's first 32-bit single-user workstation.



1985

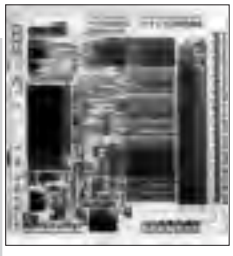
- MicroVAX chip introduced for the MicroVAX II, DIGITAL's first 32-bit microprocessor. First chip manufactured with internally developed semiconductor technology. "VAX-on-a-chip" had the highest level of functionality of any 32-bit processor in the industry.
- VMS V4.2 shipped.

1987

- VAX 8978 and 8974 systems introduced, DIGITAL's most powerful systems to date, offering up to 50 times the power of the VAX-11/780 system.
- VAXstation 2000 announced, the first workstation costing less than $5000, which ultimately became the highest volume workstation in the industry.
- New generation of MicroVAX computers unveiled: the MicroVAX 3500 and 3600.
- CVAX chip introduced, the second-generation VLSI VAX microprocessor, offering 2.5 times the power of its predecessor. The company's first internally manufactured CMOS microprocessor.

1989

- Introduction of the VAX 6500 System, DIGITAL's most powerful and expandable VAX system in a single cabinet.
- VMS V5.1 and V5.2 shipped.
- Rigel chip set introduced. Shipped in VAX 6400 system and later in VAX 4000 system.



1991

- OpenVMS name change announced.
- NVAX, DIGITAL's fourth VAX microprocessor, implemented in 0.75-micrometer CMOS technology; shipped in VAX 6600 systems.
- OpenVMS V5.5 shipped.
- DIGITAL and Microsoft Corporation announced alliance allowing Microsoft Windows to retrieve and exchange data with local area network servers running DIGITAL PATHWORKS software.
- DECnet Phase V announced.

1986

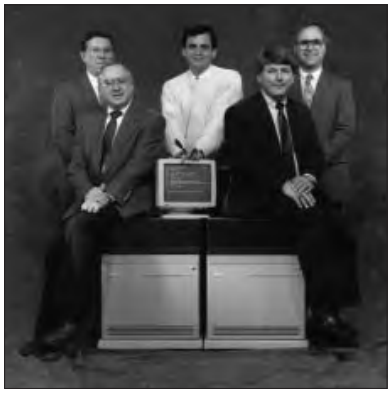
- Top-of-the-line VAX 8800, midrange VAX 8300, and VAX 8200 announced, the first VAX systems to support dual processors. Each machine incorporated VAXBI, a new high-performance bus.
- VMS V4.5 shipped.
- Local Area VAXclusters systems introduced, extending distributed computing to the Workgroup via the Ethernet and bringing the software advantages of the VAXcluster environment in MicroVAX II systems.

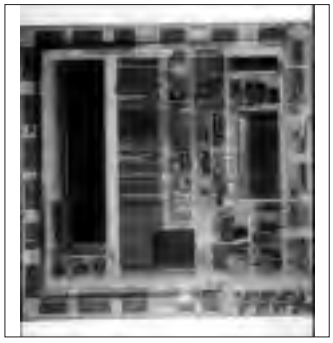
1988

- VAX 6000 System platform announced. Built on 3 key technologies: the DIGITAL CMOS VLSI VAX processor (CVAX chip), a symmetric multiprocessing hardware and software environment, and the VAXB1 I/O interconnect.
- VMS V5.0 shipped in concert with the VAX 6200 system.

1990

- DIGITAL announced VAXft 3000 system. The first fault-tolerant system in the industry to run a mainstream operating system (VMS); first system in which every component, including the backplane, was mirrored.
- VAX 6500 shipped with the Mariah chip set. The processor delivered approximately 13 times the power of a VAX-11/780 system, per processor.
- VMS V5.4 shipped.





1993

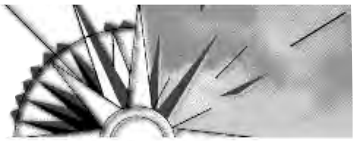
- OpenVMS AXP V1.5 shipped; OpenVMS VAX 6.0 shipped.
- DIGITAL 2100 Alpha AXP Server announced.

1995

- Affinity for OpenVMS and Windows NT program announced.
- Affinity Wave I announced – Application Vendor Partnering.
- OpenVMS Alpha V6.2 shipped; OpenVMS VAX V6.2 shipped.
- VAX4000 Model 106A and VAXstation 4000 Model 96 announced.
- Turbolaser AS8400/AS8200, AS 400 announced.
- MicroVAX 3100 Model 96 announced.

1997

- Wave IV announced – Future strategy for unlimited high-end
- OpenVMS VAX V7.1 shipped; OpenVMS Alpha V7.1 shipped.
- AlphaServer 800 announced.
- AlphaServer 1200 announced.



1992

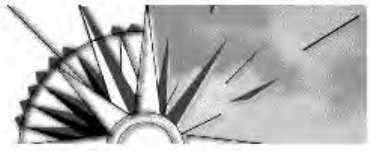
- DIGITAL announced Alpha, 64-bit processor architecture for 21st century computing, Engineered to support multiple operating systems and designed to increase performance by a factor of 1000 over its 25-year life. The first Alpha chip was the 21064, which provided record-breaking 200-Mhz performance.
- First-generation Alpha systems included the DEC 3000 Models 400 and 500 workstations, DEC 4000 system, DEC 7000 System, and DEC 10000 System.
- MicroVAX 3100 Model 40 announced.
- OpenVMS AXP V1.0 shipped.

1994

- OpenVMS VAX V6.1 shipped; OpenVMS Alpha V6.1 shipped.
- VAX4000 Model 505A/705A. announced.
- MicroVAX 3100 Model 85 announced.

1996

- Affinity Waves II and III announced.
- Wave II – Real World Deployment.
- Wave III – Advanced Partner Deployment.
- OpenVMS Alpha V7.0 with 64-bit VLM/VLDM support shipped; OpenVMS VAX 7.0 shipped.
- VAX 7000 Model 800, VAX 4000 Model 108, and MicroVAX 3100 Model 88 & 98 announced.
- AlphaServer 4000/4100, AlphaServer 1000A and AlphaServer 300 announced.



In the very long term, we see OpenVMS as the huge, bullet-proof, 24x365, disaster-tolerant data store for NT applications, in general. That kind of absolute no-excuses availability isn't going to be matched for a long time by anybody. In an Internet world, more and more of our customers need that availability right now.

—Wes Melling
VP of Windows NT and
OpenVMS Systems Group

CELEBRATING

VAX OPEN VMS AT 20

NOTHING STOPS IT.

From : Gordon Bell <gbell@microsoft.com> | Inbox

Sent : Monday, April 5, 2004 11:06 AM

To : "Dag Spicer" <[redacted]>, "John Toole" <[redacted]>, "Kirsten Tashev" <[redacted]>, "Len Shustek" <[redacted]>, "John Mashey" <[redacted]>, "Michael Williams" <[redacted]>, "Mike at Calgary" <[redacted]>

CC : <[redacted]>, <[redacted]>, <[redacted]>

Subject : RE: VAX 8600 aka Venus

Attachment : image001.jpg (5 KB)

This is an interesting machine from several viewpoints.

It was the one of the poorest managed projects in the history of DEC and two + years late.

I was the interim development manager when, after a Friday the 13th mm, yyyy review, it showed up as a clear disaster.

After a few months, I ended up appointing Bob Glorioso who had been in research to run it.

The first week in Marlboro, all the teams (I, E, F, M-box) claimed they were the architects. In the second week, no one would take credit.

Alan Kotok came in to sort through the architectural mess and get the box interfaces and design methodology straightened out as it was partially pipelined.

We started a simulator project to because as a gate array machine, there's no way to build it w/o simulation.

I called it Soul of A New Machine II (in the spirit of abysmal engineering process, management, discipline, etc.)

We were losing something like 3-5 million per day for 2 years by not being able to ship.

Vinod Khosla said it was the reason Sun could start, because the Universities had ear-marked the money for 8600.

There's a DEC Journal describing how great it was, after the project had finally shipped... and in the end it made a lot of money.

Bob had enough credibility to run the 9000 ECL project, which in the end was one of the reasons for DEC's undoing.

Ironically, Alan and I aren't mentioned in the issue.

g

From: Dag Spicer [mailto:spicer@computerhistory.org]
Sent: Monday, April 05, 2004 10:38 AM
To: Gordon Bell; John Toole; Kirsten Tashev; Len Shustek; John Mashey; Michael Williams; Mike at Calgary
Subject: VAX 8600

A few weeks back, we approved a VAX 8600. I neglected to note that the donor (Sellam Ismail) was actually hoping to recover some of his costs and that he was hoping for $500 for the item.

We do have a budget for artifact acquisitions ($5K), for FY04, and from which this could come.

Sound ok?

d.

Dag Spicer
Curator of Exhibits
Editorial Board, IEEE Annals of the History of Computing
Computer History Museum
1401 North Shoreline Boulevard

BOB GLORIOSO

VAX 8974/78 ANNOUNCEMENT

DRAFT 5 1/16/87 13:30

ROSE ANN HAS PROVIDED A GOOD OVERVIEW OF THE CHALLENGES FACING LARGE CORPORATIONS IN TODAY'S COMPUTING ENVIRONMENT. AND SHE HAS DESCRIBED HOW DIGITAL'S UNIQUE STYLE OF COMPUTING IS BEST POSITIONED TO MEET THOSE CHALLENGES.

TODAY'S LARGE CORPORATIONS REQUIRE THEIR COMPUTING SOLUTIONS TO BE JUST AS POWERFUL AS THE RIGID CENTRALIZED MAINFRAME SYSTEMS THAT WE'RE ALL FAMILIAR WITH, YET FLEXIBLE ENOUGH TO SHARE TASKS AND DATA WITH USERS, INTEGRATE DEPARTMENTAL AND DESKTOP SYSTEMS, AND GROW PAINLESSLY AS TECHNOLOGIES EVOLVE. THESE THREE QUALITIES -- FLEXIBILITY, INTEGRATION, AND OPEN-ENDED GROWTH -- ARE WHAT DIGITAL'S STYLE OF COMPUTING IS ALL ABOUT.

WITH TODAY'S ANNOUNCEMENT WE ARE EXTENDING THE DIGITAL STYLE OF COMPUTING TO A NEW LEVEL OF PERFORMANCE AND CAPACITY -- A LEVEL THAT WILL MEET THE NEEDS OF THE LARGEST ORGANIZATION. WE ARE ANNOUNCING, BY FAR, THE MOST POWERFUL VAX/VMS SYSTEMS EVER: THE VAX 8974 AND THE VAX 8978. THESE SYSTEMS ARE DESIGNED, TESTED, SHIPPED, INSTALLED AND MANAGED AS A SINGLE SYSTEM. DIGITAL'S NEW LARGE-SCALE, GENERAL PURPOSE SYSTEMS ADDRESS A WIDE VARIETY OF NEEDS FROM RESEARCH AND DEVELOPMENT TO COMMERCIAL DATA PROCESSING.

Our new systems offer all the advantages -- such as distributed processing, interactive computing, and networking -- that have come to be associated with Digital's style of computing. In short, they offer to large organizations the best of both worlds: the compute power of the mainframe world, and the easy access and integration of the world of departmental and desktop computing -- all with a single software system with the largest number of users in the world, VMS.

These systems are indeed powerful: the VAX 8978 offers up to 50 times the compute power of the VAX 11/780 (or up to 55 MIPS). Both the VAX 8974 and VAX 8978 are offered as truly complete systems. They include a new storage subsystem, data management tools, availability features, and installation and service packages that MATCH the world-class performance of the systems themselves.

The unprecedented compute power of these new VAX systems is the result of a unique multiprocessor configuration based on our proven VAX 8700 technology. The VAX 8978 is made up of eight component processors, joined in a single VAXcluster system. The VAX 8974 links four component processors, to provide 25 times the raw compute power of the VAX 11/780 (or up 27 MIPS). This multiprocessor design permits a major leap in performance while maintaining the integrity of the VAX/VMS family.

The data storage, access, and management requirements of large computing environments are also met by our new VAX system offerings. Each incorporates the new SA482 Storage Array, the highest-capacity

STORAGE PRODUCT EVER OFFERED BY DIGITAL. COUPLED WITH OUR HSC70 I/O PROCESSOR, THE SA482 DELIVERS MAINFRAME-CLASS I/O PERFORMANCE AND ENSURES HIGH LEVELS OF DATA INTEGRITY AND AVAILABILITY.

WE ARE ALSO ANNOUNCING NEW, WORLD-CLASS ADDITIONS TO THE VAX INFORMATION ARCHITECTURE THAT ALLOW OUR NEW LARGE-SCALE VAX SYSTEMS, AS WELL AS OTHER VAXES, TO MANAGE MASSIVE QUANTITIES OF DATA. DIGITAL'S DBMS AND RDB DATABASE MANAGEMENT PRODUCTS HAVE BOTH BEEN ENHANCED TO IMPROVE PERFORMANCE AND FUNCTIONALITY. A NEW PRODUCT, THE VAX DATA DISTRIBUTOR, AUTOMATICALLY DISTRIBUTES RELATIONAL DATA AMONG MULTIPLE PROCESSORS. ANOTHER NEW PRODUCT, VAX SQL, PROVIDES USERS OF OUR VAX SYSTEMS WITH AN INTERACTIVE QUERY CAPABILITY POPULAR IN MANY HIGH-LEVEL COMMERCIAL COMPUTING ENVIRONMENTS. GRANT SAVIERS WILL DISCUSS IN DETAIL THESE NEW DATA MANAGEMENT TOOLS LATER IN THIS PROGRAM.

SINCE SYSTEM MANAGEMENT, AS WELL AS DATA MANAGEMENT, IS CRITICAL IN LARGE COMPUTING ENVIRONMENTS, WE ARE OFFERING SPECIALLY DESIGNED SYSTEM MANAGEMENT TOOLS AS INTEGRAL PARTS OF BOTH THE VAX 8974 AND VAX 8978 SYSTEMS. THE MICROVAX-BASED VAXCLUSTER CONSOLE CENTRALIZES ALL SYSTEM OPERATION AND MANAGEMENT FUNCTIONS. OPERATORS CAN BOOT THE SYSTEM, RUN DIAGNOSTICS, AND VIEW AND ANALYZE CONSOLE MESSAGES, ALL FROM A SINGLE DESKTOP.

ALSO PROVIDED AS A STANDARD FEATURE IS THE VAX PERFORMANCE ADVISOR, A RULE-BASED, MONITORING AND ANALYSIS TOOL FOR THE SYSTEM MANAGER. THIS MANAGEMENT TOOL GATHERS, STORES, AND ANALYZES DATA

AUTOMATICALLY; THEN MAKES RECOMMENDATIONS TO IMPROVE OVERALL SYSTEM PERFORMANCE. THE VAX PERFORMANCE ADVISOR IS ALSO USEFUL AS AN AID TO UPGRADING AND RECONFIGURING THE SYSTEM, ENSURING THAT OPTIMUM PERFORMANCE IS MAINTAINED NO MATTER WHAT NEW APPLICATIONS OR HARDWARE ENHANCEMENTS ARE ADDED.

IN ADDITION TO COMPUTE POWER, DATA STORAGE, AND SYSTEM MANAGEMENT, OUR NEW MAINFRAME-CLASS VAXES OFFER A LEVEL OF SYSTEM AND DATA AVAILABILITY THAT NO TRADITIONAL MAINFRAME CAN MATCH. THE INHERENT REDUNDANCY OF THE VAXCLUSTER MULTIPROCESSOR DESIGN, ALONG WITH BUILT-IN SOFTWARE FEATURES, PROTECT AGAINST THE FAILURE OF ANY COMPONENT OF THE SYSTEM, HELPING TO ENSURE AVAILABILITY UNDER ALL CONDITIONS.

WITH A CONVENTIONAL LARGE SYSTEM, MAINTENANCE, MALFUNCTIONS, SOFTWARE UPGRADES, AND NEW EQUIPMENT INSTALLATIONS ALL CAN DISRUPT SERVICE. THIS ISN'T THE CASE WITH THE VAX 8974 AND 8978. UPGRADING SOFTWARE AND ADDING NEW PROCESSORS AND CONTROLLERS ARE BOTH EASILY ACCOMPLISHED WHILE THE SYSTEM IS IN OPERATION. ALL HARDWARE COMPONENTS ARE REDUNDANT OR DUAL-ACCESSED, SO PREVENTIVE MAINTENANCE OF ONE COMPONENT, OR EVEN THE FIXING OF A MALFUNCTION, DOESN'T INTERRUPT CONTINUOUS OPERATIONS. THERE IS SIMPLY NO SINGLE POINT OF FAILURE.

MULTIPROCESSOR REDUNDANCY PROTECTS THE AVAILABILITY OF VITAL DATABASES AS WELL. DATA AVAILABILITY IS FURTHER PROTECTED BY VAX VOLUME SHADOWING, A SPECIAL SOFTWARE FEATURE STANDARD WITH THESE

NEW VAX SYSTEMS. BY MAINTAINING MULTIPLE COPIES OF CRITICAL DATA ON LOGICALLY IDENTICAL DISKS, VOLUME SHADOWING GIVES USERS ACCESS TO DATA EVEN WHEN A DISK IS UNAVAILABLE. THUS DATA IS ALWAYS AVAILABLE -- EVEN IN THE EVENT OF DISK MAINTENANCE OR FAILURE.

WE ARE SO CONFIDENT ABOUT THE SYSTEM AND DATA AVAILABILITY ACHIEVED BY THESE NEW VAX SYSTEMS THAT WE ARE BACKING BOTH VAX 8974 AND VAX 8978 WITH THE HIGHEST LEVEL, MOST COMPREHENSIVE SERVICE AND SUPPORT PROGRAM EVER OFFERED IN THE INDUSTRY. THIS PROGRAM PROVIDES FOR ONE YEAR OF TOTAL SYSTEM COVERAGE, INCLUDING FULL HARDWARE WARRANTY, SOFTWARE SUPPORT AND TRAINING.

TO MEET THE DEMANDING PRODUCTIVITY AND AVAILABILITY NEEDS OF LARGE SYSTEM USERS, BOTH THE VAX 8974 AND 8978 INCLUDE A RESIDENT SYSTEMS ENGINEER THROUGHOUT THE FIRST YEAR OF OPERATION. THIS SYSTEMS ENGINEER WILL PLAY A VALUABLE CONSULTATIVE ROLE BY ASSISTING THE CUSTOMER IN DEVELOPING EFFECTIVE STRATEGIES FOR USING HIS NEW SYSTEM TO IMPROVE ORGANIZATIONAL PRODUCTIVITY.

WITH A SINGLE PART NUMBER, PURCHASERS OF OUR NEW PRODUCTS WILL BE ORDERING ONE OF THE TWO MOST POWERFUL COMPUTER SYSTEMS DIGITAL HAS EVER BUILT. INCLUDED IN THE PURCHASE PRICE IS A FULL COMPLEMENT OF HARDWARE, SOFTWARE AND SERVICES AS SHOWN ON THE SLIDE. THIS ONE STOP APPROACH TO MEETING THE COMPUTING NEEDS OF THE LARGE ORGANIZATION IS A NATURAL OUTGROWTH OF DIGITAL'S LONG-STANDING COMMITMENT TO PROVIDING OUR CUSTOMERS WITH A COMPLETE AND COMPATIBLE COMPUTING ENVIRONMENT, FROM THE DESK TOP TO THE DATA CENTER.

WITH THE INTRODUCTION OF THE VAX 8974 AND 8978, DIGITAL'S VAX FAMILY NOW OFFERS COMPATIBLE PRODUCTS ACROSS THE ENTIRE RANGE, FROM THE POPULAR MICROVAX SYSTEM, TO THE VAX 8978, WITH FIFTY TIMES ITS POWER. ALL THESE SYSTEMS RUN THE SAME OPERATING SYSTEM, VMS. ALL CAN RUN THE SAME APPLICATIONS. ALL CAN BE LINKED INTO A SINGLE NETWORK, TO OFFER A TRULY INTEGRATED COMPUTING ENVIRONMENT TO ORGANIZATIONS, NO MATTER WHAT THEIR SIZE.

ALL THE BENEFITS OF OUR SINGLE HARDWARE AND SOFTWARE ARCHITECTURE HAVE NOW BEEN EXTENDED INTO THE WORLD OF LARGE SCALE COMPUTING -- BENEFITS NO TRADITIONAL MAINFRAME SYSTEM CAN EQUAL.

VMS, THE WORLD'S PREMIER OPERATING SYSTEM ENVIRONMENT, IS NOW THE LOGICAL CHOICE FOR THE DATA CENTER. NOW, APPLICATIONS DEVELOPED IN THE DATA CENTER CAN BE DEPLOYED IN SCORES OF DEPARTMENTS, OR ON HUNDREDS OF DESKTOPS. JUST AS EASILY, APPLICATIONS PILOTED AT THE DEPARTMENTAL LEVEL CAN BE BROUGHT INTO THE DATA CENTER'S PRODUCTION ENVIRONMENT. OUR SYSTEMS ARE NOT JUST UPWARDLY COMPATIBLE -- THEY ARE TOTALLY COMPATIBLE.

NOW IT'S JUST AS EASY FOR A LARGE ORGANIZATION TO EXPAND ITS COMPUTING RESOURCES AS IT IS FOR A DEPARTMENT. USERS OF THE VAX 8974 AND 8978 CAN ADD PROCESSORS, STORAGE CAPABILITY, EVEN WHOLE NEW SYSTEMS, WITH NO CHANGE IN OPERATING SYSTEM, NO WHOLESALE EQUIPMENT REPLACEMENT, AND NO CONVERSION OF APPLICATION SOFTWARE. THE SAME PROVEN GROWTH PATH, THAT HAS ALLOWED THOUSANDS OF USERS OF OUR

VAXclusters to expand, is available for the VAX 8974 and 8978 systems.

Finally, we have priced the VAX 8974 and 8978 to maintain the price/performance advantage of our entire VAX family. The price of the VAX 8974 is $2.57 million and VAX 8978 is $4.79 million. Included in these prices is a full complement of hardware with peripherals, all of the software listed here, full networking capabilities along with one-year hardware warranty and software support. There is a sustantial discount associated with the single order number on the software for these new systems. Our first quote, to a major financial institution here in New York resulted in a savings to our customer of $238,000 on a VAX 8974 system.

This pricing maintains the $63K/MIPS price/performance consistent with the single processor system kernals of the VAX 8000 family. On a cost comparison basis, the new VAX systems provide more than a 40 percent cost-of-ownership advantage over competitive mainframes. And they do so while providing all the performance, functionality, and graceful growth advantages that we've outlined today.

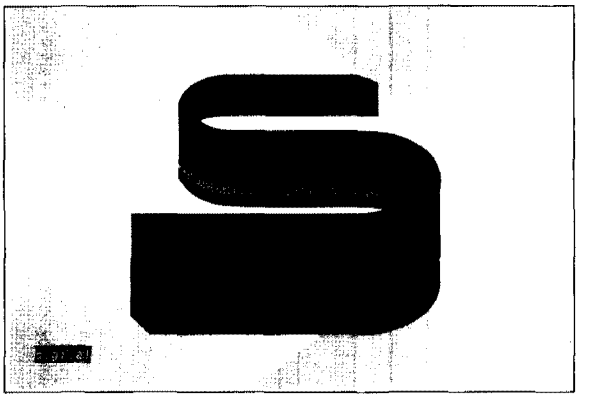
For the first time, these two new VAX systems combine the raw power of large scale computing with the flexibility and productivity of distributed computing. From now on, users won't have to make a hard, no-win choice. They won't have to compromise, either. They can truly have the best of both worlds.

EDITED
NO 24-26
NO 31-36

VAX and VMS -
The Technology Foundation

Bob Glorioso
Vice President
High Performance Systems

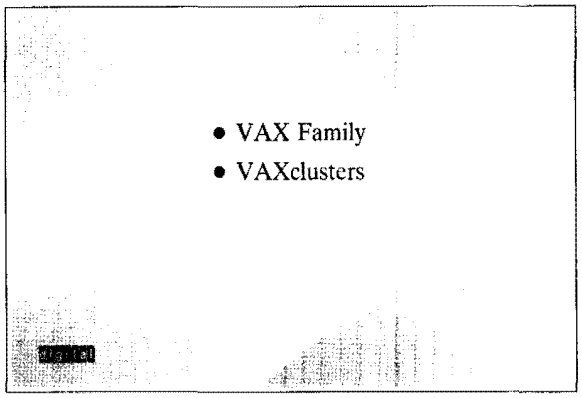
1



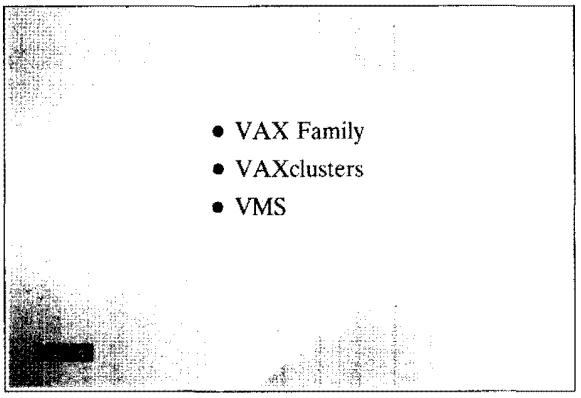
2

- VAX Family

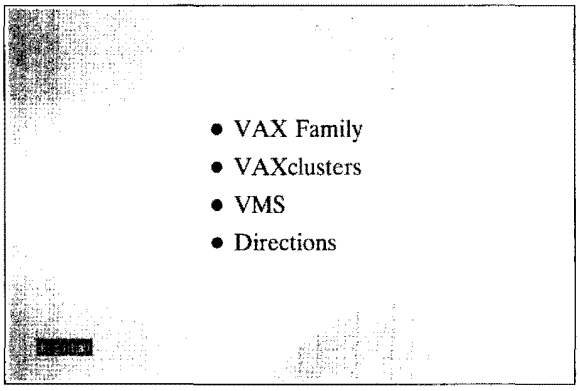
3


• VAX Family
• VAXclusters

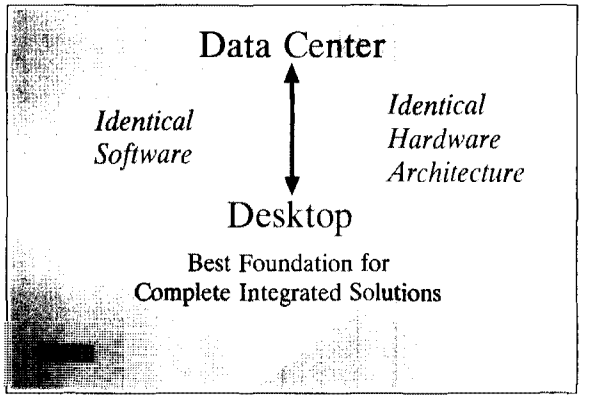
4


• VAX Family
• VAXclusters
• VMS

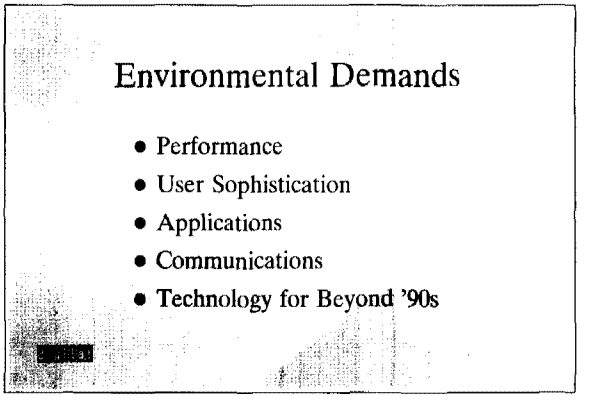
5


• VAX Family
• VAXclusters
• VMS
• Directions

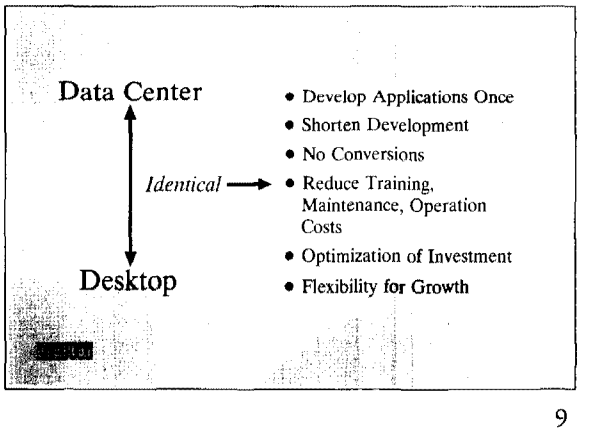
6


Data Center
Identical
Software
Identical
Hardware
Architecture
Desktop
Best Foundation for
Complete Integrated Solutions

7


Environmental Demands
• Performance
• User Sophistication
• Applications
• Communications
• Technology for Beyond '90s

8

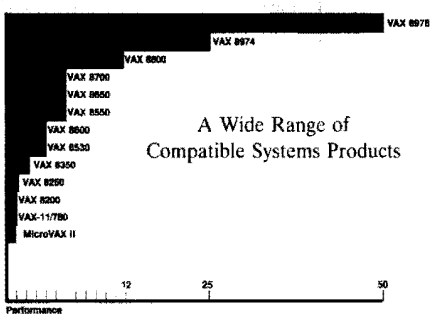

Data Center
Identical
Desktop
• Develop Applications Once
• Shorten Development
• No Conversions
• Reduce Training, Maintenance, Operation Costs
• Optimization of Investment
• Flexibility for Growth

9

Frees Enterprise to Concentrate
on Business Issues
NOT
Computing Environment Issues

10

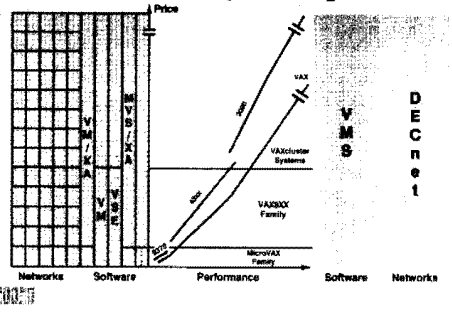
The VAX Family



A Wide Range of
Compatible Systems Products

11

Product Family Comparisons



12

Workgroup Products

- VAXstation 2000
  - Low Cost <$4700
  - Monochrome or Color
  - 15" or 19" Monitor
  - With or Without Disk
- VAXstation II & VAXstation II/GPX
  - Expandability
- VAXserver 100 - Workgroup Server
  - File Service
  - Compute Service
  - Print Services
  - Remote System Booting
  - Communication Servers
  - ULTRIX or VMS

13

Superior Environment for Workgroup Computing

- Best Operating System on the Desk
- Best Integration of All Application Environments

14

Superior Environment for Workgroup Computing

- Transparent Load Balancing
- Access to Remote and Wide Area Resources
- Security
- Systems Management

15

VAXclusters

- Local Area VAXclusters

16

VAXclusters

- "Computer Interconnect" VAXclusters
- Local Area VAXclusters

17

VAXclusters

Objectives

- Increase Performance
- Optimize Storage
- Create Common File System
- Increase System Availability

18

Local Area VAXclusters

Unique Workgroup Capabilities

- Transparent Load Balancing
- Transparent Access to Resources Outside the LAN
- High Level Security
- One-Point System Management

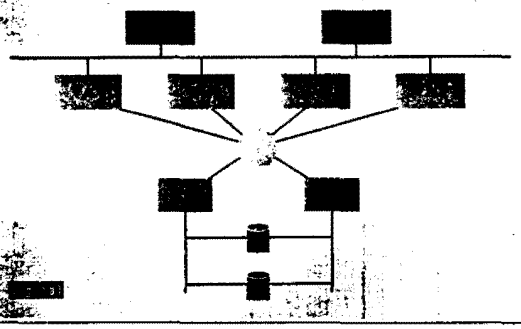
19

Computer Interconnect VAXclusters

- Link Medium and Large Scale Systems
- Provide Very Large Scale Computing
- High Availability
- Relatively Low Cost

20

Computer Interconnect VAXcluster



21

VMS History

- Designed in Conjunction With First VAX
- Architectural Framework

VMS History

- Designed in Conjunction With First VAX
- Architectural Framework

VMS Today

1977	→ 1987
FORTRAN	100s of Base Software Products
DECnet	1000s of Applications
VMS V1	VMS V4

VMS Identical Environment

- Identical: Programming Environment
  User Interface
  Built-in Network
- Shared Libraries/Services
- Complete Transportability from Any VAX to Any VAX

28

Future Trends

- Higher Performance
- Multiprocessing
- VAX
- VMS

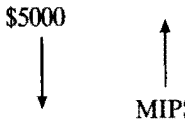
29

Higher Performance Strategies

- Heavy Investment in Base Technologies
- User Multiple Processors
- Provide Balanced Systems

30

VAX Directions



Enhance VAXclusters and
Local Area VAXclusters

37

VAXcluster

Future Directions

- Higher Degree of Integration
- High Availability Applications

38

VMS Directions

- Optimize for Workgroup Computing
- Develop Enterprise-Wide Systems Environment
  - Production/Transaction Processing
  - Next Generation User Interface
  - Enhanced Distributed Computing

39

VMS Directions

Workgroup Computing

- Cooperating Heterogeneous Systems
- Enhanced Performance and Sharing
- Transparent User Operations

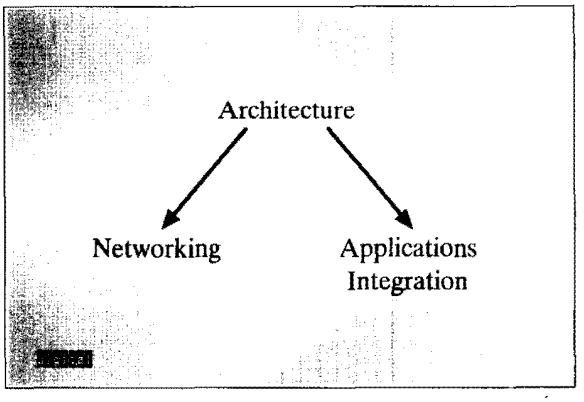
40

VMS Directions

Transaction Processing

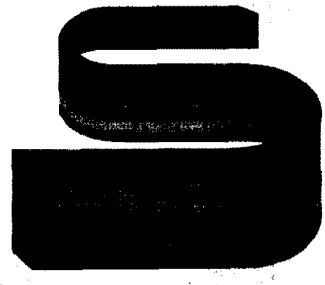
Goals	*Issues*
• Efficiency Increase	• Systems Management
• New Technology	• Media Management
• Better Resource Utilization	• Security and Network Management
	• International

41



42

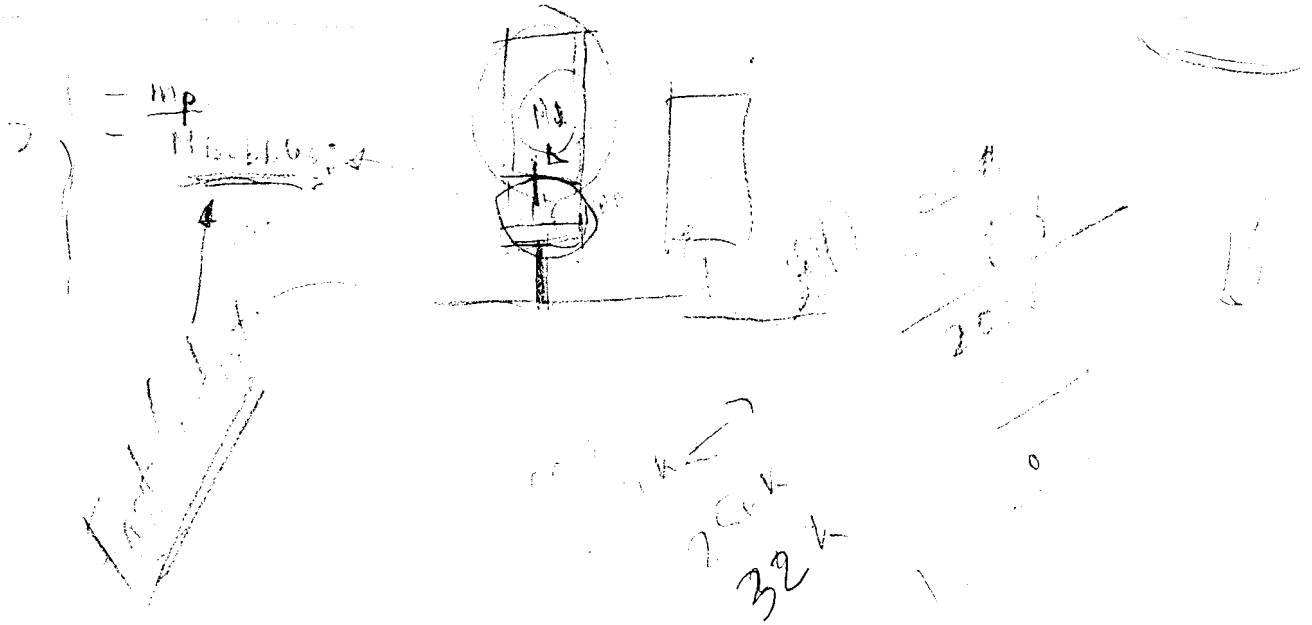
Digital Has It Now



43

NOTES ON OBJECTIVES FOR THE ADDRESS EXTENSION ARCHITECTURE

- Life Expectancy Range should be through the early 1980s.
- Allow all PDP-11 16 bit User-made programs to run.
- Users will be able to continue to write programs that will operate on both 16 and 32 bit versions of the PDP-11.
- Only minimal modifications will be required to DEC operating systems, to allow them to run 16 bit User-made programs on 32 bit hardware systems.
- Existing Operating Systems shall be extendable to take advantage of the major functional enhancements of the 32 bit hardware systems.
- No significiant training will be required for customer programmers to take advantage of the extended functions of the 32 bit PDP-11.
- 32 bit programs may call 16 bit subroutines.
- An expanded virtual address should be linear through a minimum of 24 bits and a maximum of 32 bits.
- Subsets of the architecture will be implementable over a system price range of three orders of magnitude.
- Existing peripherals will be usable on 32 bit systems.



INTEROFFICE MEMORANDUM

TO: Jim Bell

cc: Bruce Delagi

DATE: March 28, 1974

FROM: David Stone

DEPT: Software Engineering

EXT: 3741 LOC: 12-2

SUBJ: Minutes of our New Architecture Goals Discussion (3/25/74)

GOALS OF NEW ARCHITECTURE (TOTAL SYSTEM, HARDWARE AND SOFTWARE)

1. Increase the DEC dollars of profit per person employed.
2. Offer the maximal number of customer perceived products for any fixed investment.
3. Make each product trivially supportable as shipped.
4. Make each product easy to sell (e.g., offer small increments of performance for small increments of cost).
5. Offer a wide range of potential man-machine interfaces (i.e., can emulate old DEC products and competitors products as well as offer new interfaces).
6. Allow easy installation of new hardware components, including multiple units and networks.
7. Offer complete security/privacy of user data as an option.
8. Provide long system life to ensure opportunity for feedback and improvement.

IMPLICATIONS OF THESE GOALS

1. Put heavy emphasis on FA & T and field installation costs.
2. Provide self-reconfiguring and diagnosing features.
3. Create new types of documentation and interactive help dialogue.
4. Create feasible testing mechanism for all permissible system configurations.
5. Use multi-layer design hierarchy with meta-interface at each level.
6. Use a small number of components.
7. Provide a very large homogeneously addressable memory at the user level (at least). (But avoid letting the user screw himself.)
8. Provide continuously variable I/O bandwidth within the same architecture for variable amounts of money.

9. Control data paths such that

   a. all devices can address each other

   b. groups of arbitrary subsets of each system unit can be programmably isolated from the rest of the system.

10. Engage in some human factors research to understand desirable system characteristics (e.g., it may be desirable to force uniform time-sharing response time to trivial requests across lightly and heavily loaded systems even though this means enforced idleness on a lightly loaded system).

DS/jmab

digital INTEROFFICE MEMORANDUM

TO: ATTENDEES

DATE: March 21, 1975

FROM: Bill Demmer

DEPT: Central 11 Engineering

EXT: 4453 LOC: ML5/E67

SUBJ: PDP-11 ARCHITECTURE EXTENSION GROUP

Ref: Our 3/21/75 Meeting

The purpose of the meeting was to explain the plan now underway to define extensions to today's PDP-11 architecture. While there is no committed product strategy that has been accepted by the Corporation, we shall assume the attached strategy definition until further notice, as it leads to a process that best preserves our options; i.e., we shall divide the architectural questions into two categories: those which must be defined soon enough to support a calendar 1976 product shipment, and those which will require more time to define and include in an implementation effort. Our charter allows us until June 1, 1975, to reach a definition of those items needed to achieve a 1976 ship.

To accomplish this, a nucleus architectural function has been established for the next several months, where the members will spend 75-90% of their time working toward this goal. The members of this team are:

Gordon Bell	--	Project Leader
Bill Strecker	--	Architecture
Rich Lary	--	Software
Steve Rothman	--	Hardware

They, along with myself, will soon be housed on 12-3, where an ongoing concentrated effort can be maintained. The MUST output of this team, by June 1, 1975, will be a complete definition of the planned virtual address space extension, including any required supportive definitions (e.g., new instructions).

To initiate some of the other architectural extensions that will be required, and to provide a continuing review of the virtual address work, the following people are expected to provide varying degrees of their time above the 10% level:

Bob Armstrong
Jim Bell
Peter Christy
Dave Cutler
Tony Lauck
Craig Mudge
Dave Nelson

This group will plan on attending a weekly review meeting every Tuesday morning at 8:30, in the 12-3 conference room.

Items to be addressed initially by the Review Group include:

1. Instruction Set Extensions

2. Multiprocessing

3. Process Structures

Craig Mudge and Peter Christy have agreed to provide leadership and the initial proposal on the first two items respectively. I would hope to get someone assigned to the third on Tuesday. While a substantial list of other architectural enhancements is expected to materialize during the next several months (e.g., restructuring the I/O Architecture), I am planning to let them follow the above in a serial fashion, as far as spending time on proposals and definitions. A possible exception to this is a feeling that some form of generalized implementation guidelines should be generated by the Group. Craig Mudge has agreed to present this at our next Tuesday meeting, to see if it falls within the scope of an architectural (vs. implementation) framework.

Dave Nelson has agreed to act as Librarian for all the Group's activity, and will present his document and control plan to us on Tuesday (3/25/75). Other agenda items for this meeting include a discussion of our mission by Gordon Bell and, hopefully, a report of anticipated competitive activities during the next several years.

The ground rule I anticipate to be the most controversial over the next several months is the following: The Base Definition of any architectural extension will be that which is MOST COMPATIBLE to today's PDP-11. Any other alternative will not be accepted until a formal review has concluded that the gains of the Group's favored definition are sufficient to warrant further compatibility deviation.

The Corporation has had a non-compatible product strategy and felt uncomfortable. We have been established solely because of this; let us not forget it.

BD:elb

Attachment

cc: Gordon Bell
Dick Clayton
Bruce Delagi
Larry Portner
Pete van Roekens

ATTENDEES

Bob Armstrong	Tony Lauck
Jim Bell	Craig Mudge
Peter Christy	Dave Nelson
Dave Cutler	Steve Rothman
Rich Lary	Bill Strecker

STATEMENT OF STRATEGY

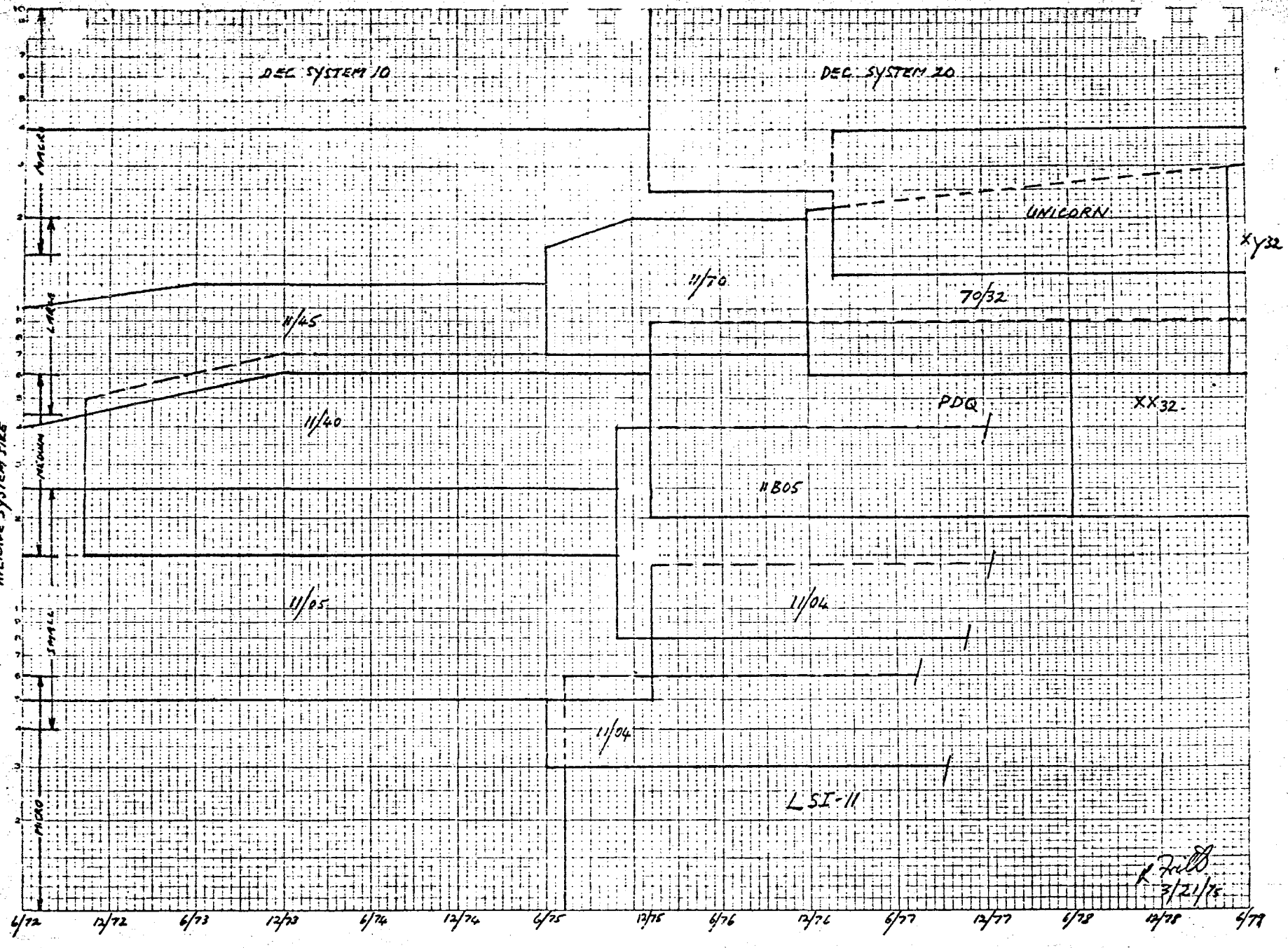
-- BRING 32 BIT PDP-11 COMPATIBLE SYSTEM INTO THE MIDI COMPUTER MARKET ASAP (11/70 - 32)

-- EXTEND THE PDP-10 PRODUCT LINE DOWNWARD, FOCUSING ON NETWORK AND TRANSACTION PROCESSING APPLICATIONS (11/85)

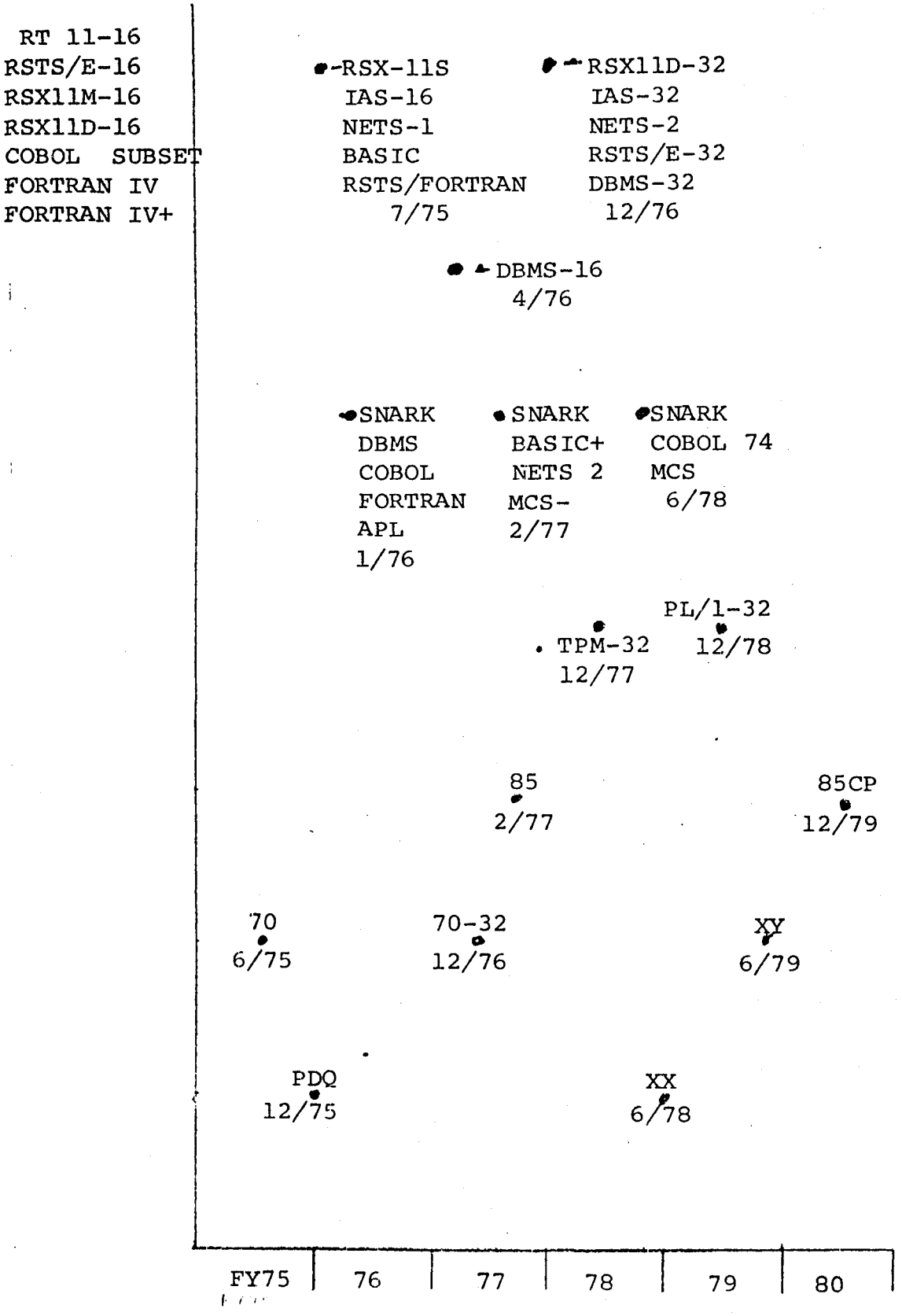
-- EXTEND THE 32 BIT CAPABILITY OF THE PDP-11 FAMILY DOWNWARD OVER TIME

-- BEGIN DEVELOPMENT OF NEW PDP-11 32 BIT TRANSACTION PROCESSING MONITOR AND PL/I LANGUAGE AND DATA MANAGEMENT SYSTEMS

-- CONVERT EXISTING REAL TIME, TIME SHARING, AND FORTRAN TO 32 BIT PDP-11 SYSTEMS ASAP


DEC SYSTEM 10
DEC SYSTEM 20
UNICORN
XY32
11/70
70/32
11/45
PDQ
XX32
11/40
11B05
11/05
11/04
11/04
LSI-11
K Fuld
3/21/75
AVERAGE SYSTEM SIZE
MAINFRAME
CARTS
MINIMUM
SMALL
MICRO
6/72
12/72
6/73
12/73
6/74
12/74
6/75
12/75
6/76
12/76
6/77
12/77
6/78
12/78
6/79

SYSTEM STRATEGY



COMPANY CONFIDENTIAL

DIGITAL EQUIPMENT CORP.

digital

INTEROFFICE MEMORANDUM

LOC/MAIL STOP

TO. Bruce Delagi

DATE. 5/6/75
FROM. Bill Demmer
DEPT. 11 Engineering
EXT. 4453
LOC/MAIL STOP. M15/E67

MAY 06 1975

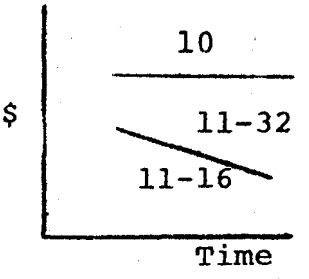
SUBJ. Strategy Alternatives

The following is what I generally concluded from our Monday morning meeting with Dick Clayton.

Strategy I

A. Expand 32 bit system downward ASAP.

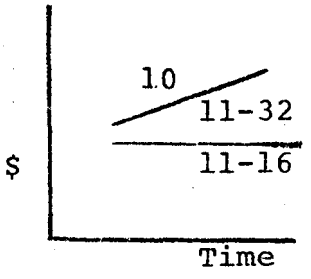
B. Make no effort to expand 32 bit system upward. —



Strategy II

A. Expand 32 bit system upward ASAP (eliminate 10).

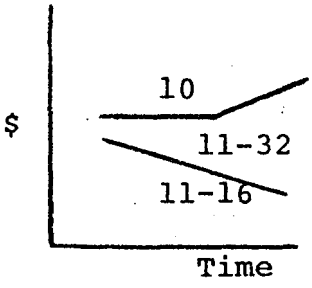
B. Make no effort to expand 32 bit system downward.



Strategy III (My choice)

A. Expand 32 bit system downward ASAP. —

B. Allow 32 bit system to move upward. Start implementation on 11/85 computers.



My Interpretation relative to the Unicorn is:

I. Unicorn and follow-on 36 bit system.

II. No Unicorn or follow-on 36 bit system.

III. Unicorn, but probably no follow-on 36 bit system.

Some Thoughts on the Initial 32 bit system are:

I believe the overriding issue here is getting to the marketplace ASAP with a product that demonstrates the direction DEC is going. This is more important than waiting until we can produce the complete 32 bit system. The most pressing need for this system is at the 11/70 level. If you accept these assumptions (and I would guess that you do not),then the initial 32 bit system should be the one that optimizes the time vs. function tradeoff WITHIN a six month window starting at the beginning of Q2 FY77. Thus, I believe the foregoing Strategy Alternatives should only influence what the second and subsequent implementations of the 32 bit system should be. Furthermore, I see no difference in FY76 activity between Strategies I and III.

B. Demmer

CC: G. Bell
R. Clayton

digital

COMPANY CONFIDENTIAL

INTEROFFICE MEMORANDUM

LOC/MAIL STOP

TO: Dick Clayton

cc: Gordon Bell
Bruce Delagi
Larry Portner

DATE: May 2, 1975
FROM: Bill Demmer
DEPT: Central 11 Engineering
EXT: 4453
LOC/MAIL STOP: ML5/E67

SUBJ: NOW IS THE HOUR -- TO INITIATE A 32 BIT IMPLEMENTATION

Now that there is evidence that we can build a 32 bit hardware machine and modify existing RSX operating systems to support it, I would propose that we proceed with the establishment of a 32 bit implementation group charged with the design of a 32 bit system that could ship in 12/76. This should be initiated immediately, independent of reaching a decision about the UNICORN. I believe this approach will preserve our options to a greater degree than other alternatives.

If there is agreement that DEC will do a 32 bit system some day, there is some probability that the first implementation should be one that minimizes the risks, gets to the market as soon as is physically possible, provides the major attributes of which a 32 bit machine is capable, and can be achieved with minimum programming support. (This, obviously, is the course of action I now prefer.) Until agreement is reached "throughout the land" on just what our Strategy should be, we could be getting a much better answer on whether we can support both a 32 bit system and a 36 bit system next year. If necessary, we could discontinue one or the other at any time during the next two to three months.

Should the direction of the 32 bit effort change toward a more global, long-term implementation, we shall have lost nothing by pressing the architecture effort to proceed more on a step-by-step basis, by pushing for a hardware breadboard this summer and by forming a nucleus design team.

Editorial

I personally believe that the attempt at simultaneous development of an all new hardware system and an all new operating system is a very high-risk approach. In fact, I would bet that we would end up shipping hardware with something other than the full new operating system for which the initial plan might call. If we could ship a hardware system in eighteen to twenty months -- one that would use modified existing operating systems and would be capable of supporting demand paging software as soon as it's available -- I believe we would have the optimum set of trade-offs achievable at this point in time. Furthermore, if we could also pull off the UNICORN on top of this, so much the better.

So, why not at least start in this direction until we know differently?

BD:elb

DIGITAL EQUIPMENT CORPORATION

COMPANY CONFIDENTIAL

STATEMENT OF STRATEGY

BRING 32 BIT PDP-11 COMPATIBLE SYSTEM INTO THE MIDI-COMPUTER MARKET ASAP (11/70-32)

EXTEND THE PDP-10 PRODUCT LINE DOWNWARD, FOCUSING ON NETWORK AND TRANSACTION PROCESSING APPLICATIONS (11/85)

EXTEND THE 32 BIT CAPABILITY OF THE PDP-11 FAMILY DOWNWARD OVER TIME

BEGIN DEVELOPMENT OF NEW PDP-11 32 BIT TRANSACTION PROCESSING MONITOR AND PL/I LANGUAGE AND DATA MANAGEMENT SYSTEMS

CONVERT EXISTING REAL TIME, TIME SHARING, AND FORTRAN TO 32 BIT PDP-11 SYSTEMS ASAP

OFFICE OF DEVELOPMENT STRATEGY TASK FORCE

3/13/75

WDemmer

RECOMMENDATION

FORMALIZE STRATEGY IMPLEMENTATION

Technical feasibility of supporting 32 bit FCS in calendar 1976 established if:

Hardware -- Project Budgets adjusted to match the strategy. ($500K transferred from UNICORN to 11/70-32 --- Delays UNICORN ship until 2/77)

-- Bob Armstrong be temporarily transferred to design and build small breadboard processor for Architecture verification and Software Testing by Q1, FY76.

Software -- Adequate priority given to RSX and FORTRAN conversion efforts.

Architecture -- Focus on completing definition of Address Extension, Mapping, ISP, and the Context portion of the Process Structure.

Aid Avoid -- Attempting to support the development of an all new hardware and software system which would have high risk.

W.R. Demmer
4/30/75

April 15, 1975
Steve Rothman

COMPANY CONFIDENTIAL

HI-END EXTENDED-11

The most accurate comparison method available appears to be to use the Unicorn as a base. The Unicorn has 15 boards in the basic machine (including the Cache). It also has two additional boards of micro-processor for a console and diagnostic use. It appears that a large 11 would have at least one board less, and possibly two. The performance of the machine would be between 350 and 400 ns for ADD R,R (32 bits). Three additional boards would comprise an optional high speed FPP (less than 5μs for 64 bit operations). The Unicorn cost is estimated at $4500 (not including any memory, but including the two console boards and the Cache). This would put the 11 cost at $5000 to $5500, because of higher speed logic, more of the boards being multi-layer, and a larger power supply.

It is believed that given sufficient resources, it is very possible to do the machine in 18 months from the time both the architecture spec and product definition are solid.

Thus, it is possible to ship a machine in calendar 76, if the specs can be nailed down by July 1. This machine could use the 11/70 memory bus and RH7Ø Massbus controls. Using the (16K) 11/70 memory increases the cost by about $2K per 256K bytes over the Unicorn MOS; however, if the 2½D memory happens on the 11/70 memory bus, cost goes down from the Unicorn by about $3.5K.

If the 11/70 Memory Bus and RH7Ø were not used, and the Unicorn SBI bus were used instead, there would probably not be a significant effect on cost or schedule; however, removing that constraint would probably increase project cost, and has the potential to increase the schedule because it might take more time to decide what to do. Note that an RH32 would have to be designed.

DIGITAL EQUIPMENT CORPORATION
MAYNARD, MASSACHUSETTS

digital

INTEROFFICE MEMORANDUM

LOC/MAIL STOP

TO: Marketing Committee
O²D

DATE: April 29, 1975
FROM: Ted Johnson
DEPT: Sales
EXT: 5942
LOC/MAIL STOP: PK-3, 2/A55

SUBJ:

I still don't understand why we, at our size, can't do both the 32 and Unicorn systems.

Both are sure winners. Both build directly on our main business and use our main resources. Surely, on some time frame, we can do both. Surely we can control the future of the product.

I beleive we should look hard at a Small 10 product management group under John Leng, continuing the Unicorn as a 10 machine. If we can answer the initial question here posiitively, then we could figure out how to fund/support the activity under Leng.

mr

digital INTEROFFICE MEMORANDUM

TO: Gerry Moore

CC: Distribution

DATE: April 2, 1975

FROM: Bruce Delagi

DEPT: Advanced 11 Engineering

EXT: 3563 LOC: ML5/E35

SUBJ: UNICORN vs. 32-bit Machine

APR 04 1975

(1) The "quick-and-dirty" virtual address extension to the 11/70 (nee 70-32) represents a trade-off of product life to get early introduction (just as we did in the /70 itself).

(2) The iron-machine orientation of the current 32-bit competitive products aims at a smaller system size than the 11/70 design.

(3) I believe DG's thrust into this area will develop around a virtual memory (demand paging) operating system and that within the calendar year competitive pressure will increase on:

- virtual memory operating systems (e.g. SNARK)
- larger program (data array) size
- languages capability
- DBMS facilities
- network capabilities
- cache organizations
- "process structure" in systems and applications software

and decrease on:

- physical memory size (driven by demand paging)
- memory bus bandwidth (driven by cache organizations and system decentralization as encouraged by network developments)

(4) DG and HP are ahead of the 11 and behind the 1Ø in each of the areas in which I think market pressures will be increasing.

(5) The 70-32 will not permit us to hold a stronger position than DG and HP 18 - 30 months from now, but the UNICORN will.

(6) We can get ahead of DG, MODCOMP and HP in the 11-based products by reserving our shot for an evolutionary system organization that:

- permits users to trade system size against performance at constant functionality (i.e. small systems have the same features as big systems but run slower)

- supports the concepts of program structuring by improving interprocess switching times and communications facilities.

- provides better string and decimal arithmetic facilities for higher-level language (especially COBOL) support.

- allows manipulation of large arrays and more convenient access to random access file structures.

(7) Some guesses: System organization providing the above capabilities and implemented in the PDQ price range would ship nine months or so later than a 70-32. Doing the 70-32 implies delaying this "right solution" by 15 months and delaying the /85 six months.

(8) Alternative 1
/85: September '76
Super PDQ: September '77

Alternative 2
70-32: December '76
/85: March '77
Super PDQ: December '78

(9) We are about to ship the /70 and are having serious reservations about its effective life. If the first "product enhancement" of the original /45 design(begun in 1970) is felt to have a short life (18-21 months) will the second such enhancement be effective over a 24-month life - till early 1979?

Distribution

Ted Johnson
Gordon Bell
Dick Clayton
Bill Demmer
Larry Portner
Win Hindle
Andy Knowles
Stan Olsen

/gml

digital INTEROFFICE MEMORANDUM

TO: Dick Clayton

CC: Ted Johnson
Gordon Bell
Bruce Delagi
Bill Demmer

DATE: March 28, 1975

FROM: Gerry Moore

DEPT: N. A. Sales

EXT: 3148 LOC: PK 3-2

SUBJ: Unicorn vs. 32-bit Machine

The more I think about the problem the more I come to the conclusion that a 32-bit machine must have priority over Unicorn. My perception is that the 11/70 is very definitely an interim machine and that we were in better shape in terms of competitive posture and trend in the marketplace after we introduced the 11/45 than we are now after having introduced the 11/70. The competition is coming on strong from some with full 32-bit high performance minis. Unicorn is not a replacement for the 11/70 and will not be perceived as such in the marketplace. If we don't do a 32-bit replacement for the 11/70 with some degree of urgency we will be in a difficult position in 18 months or so from a competitive standpoint. The 11/70 will then be totally non-competitive (and, as I say, Unicorn will not be able to fill the gap).

It would be nice not to have to establish priorities and do both machines. I believe Unicorn should be done, not as a gap filler, but as a low end DECsystem-10. I believe it's got to be done simply because it can be done, and if we don't do it someone else will. This whole argument hinges on the assumption that the Unicorn manufacturing costs are substantially below DECsystem 20-10. However, from the view of overall priorities, we can afford to let the Unicorn trail the 32-bit machine by a year.

/mp

HIGHLIGHTS OF PROCESS

32 BIT STRATEGY

I. Virtual Address Alternatives (Most Probable)

1. Segmented 16 bit -- very compatible
2. Linear 32 bit approach -- compatible where convenient
3. Linear 24 bit/8 bit segmented approach -- more compatible than 32 bit linear

Recommendation by end of April on VAX (Partial 32 bit system definition)

II. Implementation Strategy Alternatives (Most Probable)

1. Partial 32 bit system in 1976, + 11/85 in early 1977
2. 11/85 in 1976, + full 32 bit system in late 1977
3. Full 32 bit system in mid-1977, + no 11/85

Recommendation on strategy by end of May

III. Full 32 Bit Architecture

1. Recommendation, by end of May, on what can be included in partial 32 bit system for 1976 FCS
2. Recommendation, by end of August, on what the full 32 bit system is

W.R. Demmer
3/25/75

INTEROFFICE MEMORANDUM

TO: Distribution

DATE: 3/19/75

FROM: Pete van Roekens PvR

DEPT: 8/11 Software Development

EXT: 4028 LOC: ML5/E76

MI Copy and return

SUBJ: 32 BIT MACHINE

MAR 27

The goal is to ~~pose~~ pose an architecture (and some alternatives), the evaluation criteria, and ~~some~~ some implementations. The tradeoffs will be short term vs long term. Clearly the dominate part of this is the software.

The 32 bit machine offers DEC several important opportunities. We will be able to:— for the first time on an 11:

1. Extend a single hardware family to encompass a wide range of market needs.
2. Create software systems with a significantly higher degree of compatibility than is possible between the PDP-10 and PDP-11.
3. Sell add-ons to a large customer base with little impact on existing customer software.
4. Provide further impetus to our systems level planning and design activities.

Peter you still have 2 windows a responsibility of compatibility among 11's vis a vis languages & system.

I believe that we should concentrate our resources on the 32 bit machine to realize its full market potential. If we pursue a dual machine strategy, eventually we will develop and support complete software systems for both, but at almost double the cost.

Peter

we currently have a dual strategy 10 + 11 (if you count RT, RSTS, IAS, and RSX's as only 1 machine — in the Software budget they are indeed 4+ Machines!). ~~Retiring~~ Doing the 32-bit Machine gets us a three machine strategy: 10, 11x(?), 11-new

Gordon.

Distribution

Gordon Bell
Jim Bell
Peter Christy
Dick Clayton
Peter Conklin
Bruce Delagi
Bill Demmer
Robin Frith
Larry Portner
Larry Wade

(copy)

DIGITAL EQUIPMENT CORP.

INTEROFFICE MEMORANDUM

TO: Bill Demmer

CC: Dick Clayton ML5/E71
Jega Arulpragasam

DATE: 19 March 1975

FROM: Craig Mudge

DEPT: 11 Engineering

EXT: 5064 LOC: ML5/E54

SUBJ: A SMALL, AGGRESSIVE TEAM TO PROPOSE A 32-BIT 11

My concept of this team and its product is as follows:

1. A small, aggressive team of four people:

	Responsibilities
Marketing	Market needs; user survey; RAS; migration.
Systems Architect	Balanced design; VAX; conceptual integrity.
Software	VAX; other new software.
Hardware	New technology memory, logic; implementation cost estimates.

The team should be...

Marketing	(?)Perhaps Don Street or Keith Myles
Systems Arch.	Mudge
Software	Brender
Hardware	Rothman

Bill Strecker must be available as a consultant 50% of his time.

The manager of the team should be Bill Demmer.

2. Method of operation:

The team should be closeted from interruption and being jerked around by strategy committees. The people mentioned in your memo of 3/14/75 (Armstrong, Christy, Cutler, Nelson, Hughes, Bell, Lauck) and other key DEC people should be available as discussants.

The team should provide a weekly update to Dick Clayton.

The project would have three phases:

Phase I: 4-man team closeted.
Result: Documented proposal.
Phase II: Evaluation of proposal (by everyone, including hipshooters).
Phase III: Final proposal incorporating Phase II modifications; product acceptance or rejection.

3. The document from Phase I:

Business plan - Market, competition, pricing, etc.
System Structure - architectural extensions; implementation; generality.
Build plan
Project evaluation criteria right through L/R

Relation to longer term product development

Software

Alternatives considered

4. Phase I project plan

We can write this over the coming weekend. The key issues to be addressed in Phase I are:

Marketing
VAX
VAX migration
networks
software
alternatives
components (memory, logic)
RAS
systems implementation language
11/70 enhancement

5. Schedule

The key problem is the current, continuing commitments of team members on the 11/70 and PDQ.

With this in mind I suggest

Phase I	3 months
Phase II	1 month
Phase III	2 weeks

Since time to market is of critical importance, recruiting for the build team should begin late in Phase I, if there has been adequate progress in Dick Clayton's judgment.

/ecm

75JA373-123

COMPANY CONFIDENTIAL

DIGITAL EQUIPMENT CORP.

digital INTEROFFICE MEMORANDUM

TO: Gordon Bell
Dick Clayton
Bill Demmer
Larry Portner

CC: Distribution

DATE: 21 March 1975

FROM: Jega Arulpragasam

DEPT: 11 Engineering

EXT: 5545 LOC: ML5/E54

SUBJ: A 32-BIT PDP-11

What is the minimum life we must demand from any extended 32-bit PDP-11 architecture?

I would anticipate a time difference of at least 6 years between the first and last implementations of such an architecture. This implies that technology still about 8 years away will be used. It also implies that implementations of this architecture are still competitive in the marketplace, 10 years from today.

I see a longer time scale needed to maximize the return on our software investment dollar. It would also minimize the absolute cost of software over a reasonable period.

If you agree with me, then you must share my concern about trying to crash this architecture. Crashing implementations is not bad, because hardware costs are small and are continuing to decrease relatively steeply. Crashing architectures, however, is paid for (TANSTAAFL) by amplifying software costs, and by constraining the quality of the product we can deliver -- ever. Three to five years from now, I can hear Gordon ask, "Why are we always playing catch up?"

The major task is to define, rigorously and explicitly, the objectives this architecture is to serve - beforehand. Devising the how of doing so, afterwards, is relatively easy - especially for the kind and calibre of people we have at DEC.

Some of my proposals have been incorporated in Craig's memo of March 19th. However, there are at least two very significant differences between what he proposed and what I favor. Presenting the latter in its entirety will, therefore, be less confusing.

I propose that:

1. A five man team, each of whom will work at least 50% of the time (some full-time) be assigned the explicit responsibility to
   a) Establish objectives
   b) Define architecture
   c) Validate by outline specs. and Phase 0 cost estimates for high performance and low cost implementations; and
   d) Initial project plan for first implementation, including hardware/software phasing, development costs and schedules.

2. The composition of the team will be
   a) A manager,
   b) An architect/computer scientist,
   c) A hardware systems specialist, with constant access to technologists,
   d) A software specialist, and
   e) A marketeer.

   Each of these members will have a good background in at least one of the other areas. They will however, consult with individuals outside the team continually through the project.

3. A wider review body representing both higher management and other technical specialists will have joint sessions with the nuclear group, every 3 or 4 weeks.

   The purpose will be to communicate, to monitor and to guide. But it will not be to direct in any authoritarian sense.

   This last is important in my notion of responsibility. Control by management is exercised by the final approval process, and before that by explicit replacement of members of the team.

4. A reasonable schedule would be

   Phase I (Tasks in item 1 above): 4 to 6 months.
   In this period elapsed time (for maturing) is as important as man-hours spent.

   Phase II (Review of total proposal): 1 month
   In this period, the total plan will be reviewed by a much wider segment of the corporation than the formal review body. While many people will be charged with making formal inputs, the nuclear team will actively solicit verbal inputs. Hipshooters are welcomed for the insights they will undoubtedly provide, but should not expect to provide direction!

   Phase III (Approval): 1 to 2 months.
   In this period, the nuclear team will integrate into the proposal, whatever inputs they feel to be of value from the activity in Phase II, and (interactively) get approval from management.

This plan is 6 months slower than the one we discussed. But the value to the corporation is tremendous, affecting as it does software costs over the next several years, the image of DEC as an innovative company, our ability to provide superior products, and even employee morale -- which affects all the items above.

Finally, I realize this 6 months may require us to provide an interim solution. So be it. But then it is appropriate that band-aids are viewed as interim solutions from the outset.

The bulk memory limited virtual address extension will probably suffice. It is low cost in software (Dave Cutler), and will require the redesign of a single board (Cache/KT) if applied to the PDQ. And manufacturing would not face introducing a whole new product, except the bulk memory itself, of course.

Don't you agree it could hold us for at least a year, and that its life will extend beyond that period?

/ecm
Distribution:
Peter Christy
Bruce Delagi
Craig Mudge
Steve Teicher
Mike Tomasic
Pete Van Roekens
Larry Wade

digital INTEROFFICE MEMORANDUM

TO: Pete van Roekens

cc: Bob Bean
Frank Hassett
Richie Lary

copy to Bill Demmer + file
Larry Portner

DATE: March 10, 1975

FROM: Ken Ellson KE

DEPT: Multi-Access Softw. Eng.

EXT: 3742 LOC: ML5/E76

SUBJ: THOUGHTS ON THE NEW COMPUTER ARCHITECTURE

CONFIDENTIAL

Recently, there has been some serious discussion about building a new machine with 32 bit word size (PDP-X). I have some strong feelings about the parameters for this system and they are set down here for your comments.

I. OPTIMIZATIONS AND TRADE OFFS

At the grossest level, we can optimize in four main areas - minimize short term development costs, minimize long term development costs, maximize system performance or maximize compatibility within the new system software or with the PDP-11. We can, of course, select some combination of the above. It is my belief that the order of priorities should be long term development costs, intra system compatibility, performance, PDP-11 compatibility and short term development costs.

II. IMPLEMENTATION LANGUAGE

I believe that it is absolutely imperative that the first piece of software produced for the new machine be an implementation language. It should not be PL/I since it is far more important for this language to be excellent rather than standard. The implementation language should also have the ability to handle assembly level instructions mixed in with the higher level instructions in order to circumvent Lary's Law*. In fact, I believe that we should not implement an assembler at all - a good implementation language with the ability to mix assembly instructions will do the job better.

III. FIRST SHIP, HARDWARE AVAILABILITY, ETC.

Based on past experience, the development scenario for the PDP-X will go like this: decision to go ahead, hardware designed with minimal input from software people, build

---

* Lary's Law - The quality of the code output by an implementation language is inversely proportional to the readability of the language statements.

RECEIVED
MAR 19 1975
BILL DEMMER

RECEIVED
1975

P. van Roekens

several prototypes, software development gets minimal time in off-hours on flaky configuration, first hardware ship with any conceivable configuration and 6-12 months later, software ships. Net result is multiple pre-releases of software (expensive), large amount of field support (expensive), inferior software due to pressure to do it quickly (expensive to maintain and eventually replace), and reduced long term sales due to customer dissatisfaction (expensive).

What we should do (to maximize net profit instead of gross short-term sales) is first define the market(s), next design the system (hardware and software together), then begin detailed design and implementation of both hardware and software. The first system ship date should be the date when both hardware and software are available. We could (and probably should) announce a hardware only availability date prior to the system availability date and accept orders for iron.

Consider the financial ramifications of the above scenarios. When we accept an order for a system and ship hardware only, we incur the expense of building, shipping and installing the hardware. A sum of money is transferred from inventory to accounts receivable but often the money itself is not paid until the software arrives. Thus, in addition to the cost of building the hardware, we incur shipping, installation and support costs prior to being paid. On the other hand, if we refuse to accept system orders for delivery prior to the software availability, we incur only manufacturing costs early. We have some number of hardware systems in inventory and when the whole system is ready, we can ship and install the system and send the bill. Furthermore, support costs are not only deferred until after payment is received, but are also decreased since a finished, tested and documented product is shipped.

IV. DO IT NOW OR DO IT RIGHT?

Given that we will be living with the PDP-X for years to come, I believe that we must do it right. We will have to resist the pressure that is sure to arise to do something quick and dirty. To some extent, this pressure can be minimized by getting started on the software early, by having an implementation language available, by designing the system so that performance can be improved by replacing modules instead of systems and by talking about system availability instead of hardware availability. Nonetheless, the pressure to get it out soon will be heavy and will increase as the ship date nears. We will have to either resist and suffer the short term consequences or yield and suffer the consequences for years to come.

The whole point of this memo is that it is time to start now. We all know what needs to be done and the sooner we start, the easier it will be for everyone.

fp

digital

INTEROFFICE MEMORANDUM

TO: List

LOC/MAIL STOP ML-5/M-46

DATE: April 9, 1975
FROM: Dave Best
DEPT: Ind. New Products Dev. Group
EXT: 6477
LOC/MAIL STOP: Maynard 5-2

CONFIDENTIAL

APR 11 1975

SUBJ: Industrials View of Software Needs for 32bit Computer

In a meeting on March 28, the Industrial Products view of software needs for the 32 bit machine were worked out. Participating in that meeting were Steve Mikulski, Joe Brownstein, Jim Davis, Nick Wells, Dave Best, and Dave Cutler. The results of that meeting were presented to the Industrial Strategy Committee for review on April 7. The following points summarize the software requirements for the 32 bit machine as view by the Industrial Products Group.

First Customer Ship

Strong emphasis should be placed on real time. Would like to see a compatible implementation of RSX-11M or RSX-11D. The system must support execution of all programs written in MACRO-11 and higher level language (FORTRAN in particular) with the possible exception of programs which run in Kernal mode (i.e. priviledged tasks and device drivers in RSX-11D). The interface to the operating system, files system, etc. should be compatible with current RSX-11M/D.

Users should be able to develop and run programs written to take advantage of the 32 bit hardware features in a mixed environment with unconverted 16 bit programs. The operating system must support any virtual address extension hardware, and allow users to develop programs to take advantage of the VAX features.

The system must be part of DECNET, and support development of RSX-11M/S and tasks for 16 bit PDP-11's for down line load.

Speed of the system should be such that it can run 16 bit PDP-11 programs at least as fast as the PDQ (if not faster).

Additional data on actual hardware implementation is required to further "tune" the 16 to 32 bit transistion, mix, and compatability issues.

Next Immediate Priority

The next step for the operating system should include transaction processing capabilities, with more emphasis on the manufacturing/commerical capabilities. This includes production batch processing, and better data base management similar to "IDMS" and compatible with CODASYL DBTG standard. The DBMS should provide a convenient, easy to use interface such as Cincom's "TOTAL". It should be a basis for implementing functions such as, a bill of material processor and must include access methods such as ISAM, linked records, backward (up tree) references, etc.

More emphasis should be placed on high level language support for transaction processing (i.e. Cobol programs should be able to converse with any device in a compatible manner, without regard to the particular device characteristics).

Better communications capabilities for interface to IBM 370 OS/VS are needed. 2780 emulation is not sufficient. This may include HASP work station and multileaving support and/or emulation of an SDLC device such as 3270, 3790, etc. depending on how IBM proceeds.

Time Sharing Strategy

If the 11/85 with Snark is done, time sharing on the 32 bit machine should not be a requirement. In general, the time sharing world is not hung-up on the 32 versus 36 or 48 bit issue. What time sharing users want/need is a "wide word" machine for vitural address extension, arithemitic precision, and speed.

DB:bb

List: Gordon Bell
Dick Clayton
Bill Demmer
Larry Portner
Bob Puffer

CC: Industrial Strategy
Joe Brownstein
Dave Cutler
Al Mastendino
Steve Mikulski
Nick Wells

B.J. Chelson 4/9/75

CONFIDENTIAL

DISTRIBUTIONS

VAXA
xB
&C

A.

Gordon Bell
Bill Demmer
Rich Lary
Bill Strecker
Steve Rothman

B.

(Group A, plus)
Bob Armstrong
Peter Christy
Dave Cutler
Jim Bell
Craig Mudge
Dave Nelson
Tony Lauck — John Buckley
? — Hotzpuy — Gary Hendrischen

C.

(Group A,B, plus)
Dick Clayton
Bruce Delagi — Teiler
Larry Portner
Larry Wade
Pete van Roekens
Mel Woolsey
Jega Ega
Tomasic

HEADING CATEGORIES

{P, M, S, D, K, T, L} . [sub cat.] /TITLE

EXAMPLES

PC.ISP	--	INST SET
PC.PS	--	PROCESS STRUCTURE, PS
K.UN	--	UNIBUS CONTROLLER
K.MB	--	MASSBUS CONTROLLER
S.UN	--	UNIBUS
S.MB	--	MASSBUS
S.M	--	MEMORY BUS
C.MP	--	MULTIPROCESSORS
MP.Cache	--	CACHE
MP.MAP	--	MEMORY MAP
D.FP	--	FLT. PT. UNIT
MS,	--	MASS STORAGE
T	--	TERMINALS
L	--	COMMUNICATION LINKS

D I G I T A L INTEROFFICE MEMORANDUM

PAGE 1

SUBJ: VAX DOCUMENT INDEX DATE: 04-15-75
FROM: MARY JANE KEENE
EX: 2237
MS: ML12/A51

TO: GORDON BELL ML12/A51

To: VAXC

Please find attached the latest listing for the VAX DOCUMENT INDEX. Dave Nelson, I would appreciate it if you would scan the listing and change the notations where necessary--there seem to be a lot of "C.'s".

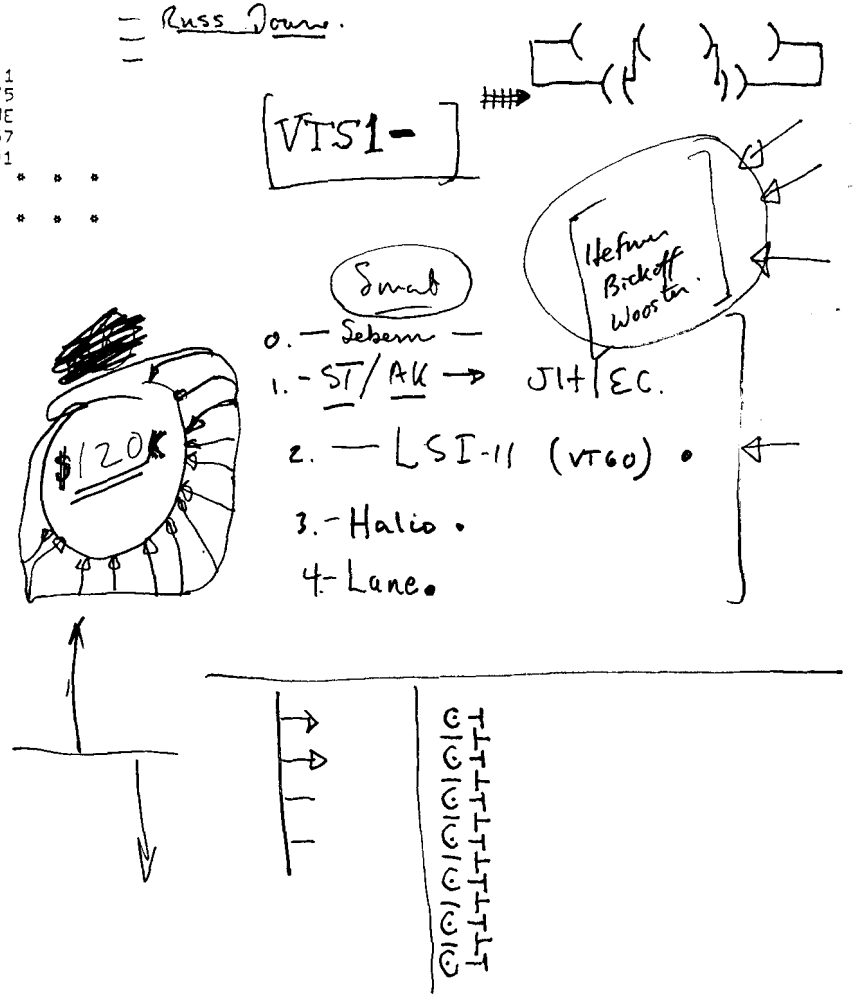
If any of you have documents you feel should be represented, please send the original, if possible (which I will return), or a good copy.

Be sure to identify everything you write with your name, date, subject, and sent to. It would be helpful if you would precede the subject with the proper notation:

Ps.isp	INST SET
Ps.ps	PROCESS STRUCTURE, PS
K.un	UNIBUS CONTROLLER
K.mb	MASSBUS CONTROLLER
S.un	UNIBUS
S.mb	MASSBUS
S.m	MEMORY BUS
C.mp	MULTIPROCESSORS
Mp.c	CACHE
Mp.map	MEMORY MAP
D.fp	FLT.PT. UNIT
MS.	MASS STORAGE
T.	TERMINALS
L.	COMMUNICATION LINKS

We can expand this list as we go,

mj



digital

INTEROFFICE MEMORANDUM

TO: John Buckley

DATE: 27 September 1976
FROM: Bill Demmer
DEPT: Advanced 11 Systems
EXT: 4453
LOC/MAIL STOP: ML3/E35

SEP 30 1976

SUBJ: INFORMATION ON THE EVOLUTION OF VAX

The attached is a summary of the thinking process that led us to the current VAX strategy regarding the tradeoffs made in the architecture and implementation plan relative to the base 16 bit address PDP-11.

I believe the two key decisions made in this process were:

1) The basic architectural definition of a linear address space.

2) The first implementation system could develop a satisfactory migration strategy without adding the penalties of incorporating the 11/70 memory management function.

The first is fundamental to achieving a long term architecture that also preserves the style and structure of PDP-11 programming in the extended environment. The second is based upon the general need not to proliferate all PDP-11 system types and assumes the presence of a PDP-11/VAX "Umbrella" migration strategy, thus permitting a broad range of migration capability among sets of PDP-11/VAX systems.

I would hope you will be able to extract from this document only what is necessary to lead your interested people through the process and having them reach the same conclusions as Digital has.

Attachment
BD:kj

Bill

The PDP-11 / VAX Evolution

Background

It is anticipated that by the 1980s both Digital and its customers will have made massive investments in software for the PDP-11 family. It is vital to both that future systems capitalize on these investments as much as possible, whether they be in the form of programs, data bases, operational environments/procedures, people training, etc.

It is also anticipated that by the 1980s that the inherent virtual address limitation of the PDP-11 architecture will become a significant detriment to the addition of many new potential applications and hence the continued growth of the family.

In early 1975 a small project team was set up to develop a plan that would attempt to make the best tradeoffs possible for optimum resolution of these divergent goals: utilization of the investments and expansion of the virtual addressing capabilities of the PDP-11 architecture.

VAX - The Virtual Address Extension

The fundamental question initially addressed by this group was, could the basic PDP-11 architecture have its addressing structure expanded in such a way that the following goals would be achieved:

1) That the expansion be completely transparent to the base architecture.

2) That the expanded architecture permit long term competitive cost/performance implementation in similar styles and structures as the base architecture.

The Migration Philosophy

The reason that there appears to be little need to incorporate the PDP-11 memory management is that we can develop a migration strategy that would render it a little used function which could not justify the corresponding cost increase. The key ingredients in this migration philosophy include:

1) The majority of existing data bases would be directly usable. Those that are not (RSTS generated) could be converted once with a conversion utility program we would provide.

2) PDP-11 programming knowledge would be directly applicable with only minor additional information required to take advantage of new functionality.

3) Operational environments/procedures would be unchanged, although RSTS users would see some Command Language differences.

4) User programs to a given operating system interface would run unmodified under the VAX implementation emulator which is composed of the non-priviledged instructions executing directly in the hardware and microcode and the system function being handled by a software emulator running under the VAX operating system. It should be noted that this permits the use of all the utilities and high level languages that meet the PDP-11 operating system interface. (The initial implementation of VAX will support those programs that meet the PDP-11 RSX11M operating system interface). These user programs can be run simultaneously with new "native mode"

are also user programs that interface other PDP-11 operating systems (eg. RSTS) which would also need modification to run under the VAX PDP-11 emulator. Our initial assessment is that there is a minimal exposure here due to the use of high level language or the often rewrite of these types of programs, even though preservation of the data base may be important. However, if our assumption proves wrong this is probably correctable through the development of an emulator that meets the other PDP-11 (RSTS) operating system interface.

In summary, we have evolved an architecture that cleanly solves the basic limitation of the PDP-11 and we have developed an implementation plan to overlay both the extended architecture and the basic PDP-11 architecture that permits the system to appear to be an extended model of the PDP-11 family. In doing this we have a situation where both Digital and our customer base can greatly capitalize their investment in the PDP-11.

wrd 9/24/76

24 July '75

Class	Design	Writing	
1		½	Reading
		1	Review writeups
2		3	Arbitrating
3	3	2	Coll/Returns
	1	1	Structure & Const.
3.5	½	½	String Proposal (First draft)
	½	½	Fields Proposal
4	3	3	Personal Finance
	1	1	Message Passing
5	4	2	Computability & Soviet
6	2	2	Review Problems
	15	16	

Processor Reference Man'l 2 — work writing

current addressing proposal — done ——

more policy flexibility in PS — Wall Cutler 8/19 Upham Hastings — Hastings 8/20

traps & interrupts — — Hastings? 8/20

call/return — — done — —

strings/decimal arith — Rodney Infante 8/21 — Rodgers 8/

fields-bits — — done — —

I/O instructions — Rodgers Stewart 8/22 — Rodgers 8/

compatibility — Shrader Upham Cutler Kenney Stewart 8/22 — Shrader 8/

74 comments ~~suggested~~ amplified. — — done — —

Review by 7-8 September (?)

Processor Ref. Man'l S.

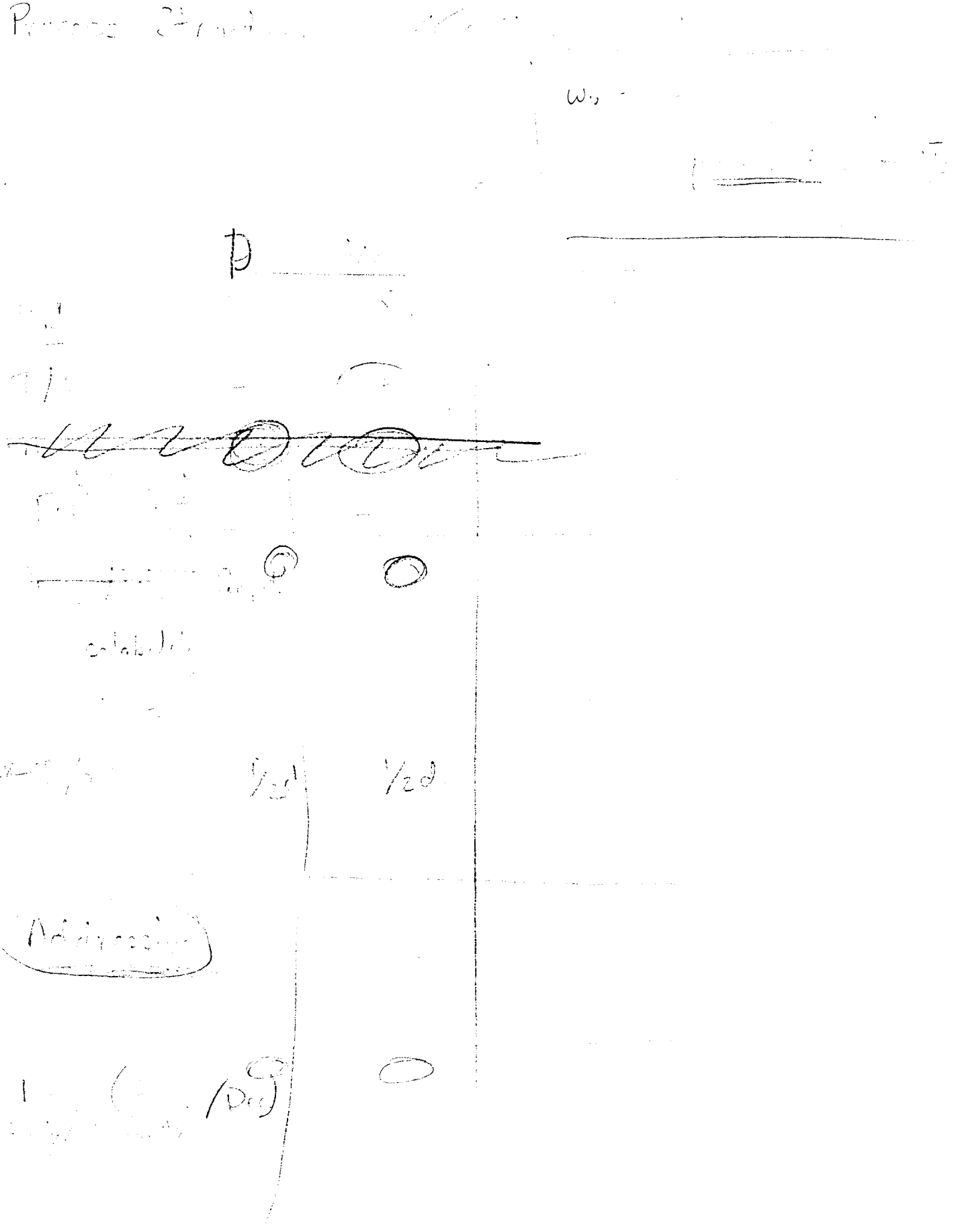
message passing

capabilities — Wall Shrader 8/22 — Shrader 8/

multiprocessor.

subsetability

arithmetic Rounding and Bell



Hookup / Structure / Redraw / Long / Self Design / Write / Review

Type / Proposal / Distribute

S Encoding, Addressing, Space update, Call/Ret acts for Review

It Addressing

S Str. Control.

S Call / Return.

DL Strings / Fields / Bits

Gen. Cleanup.

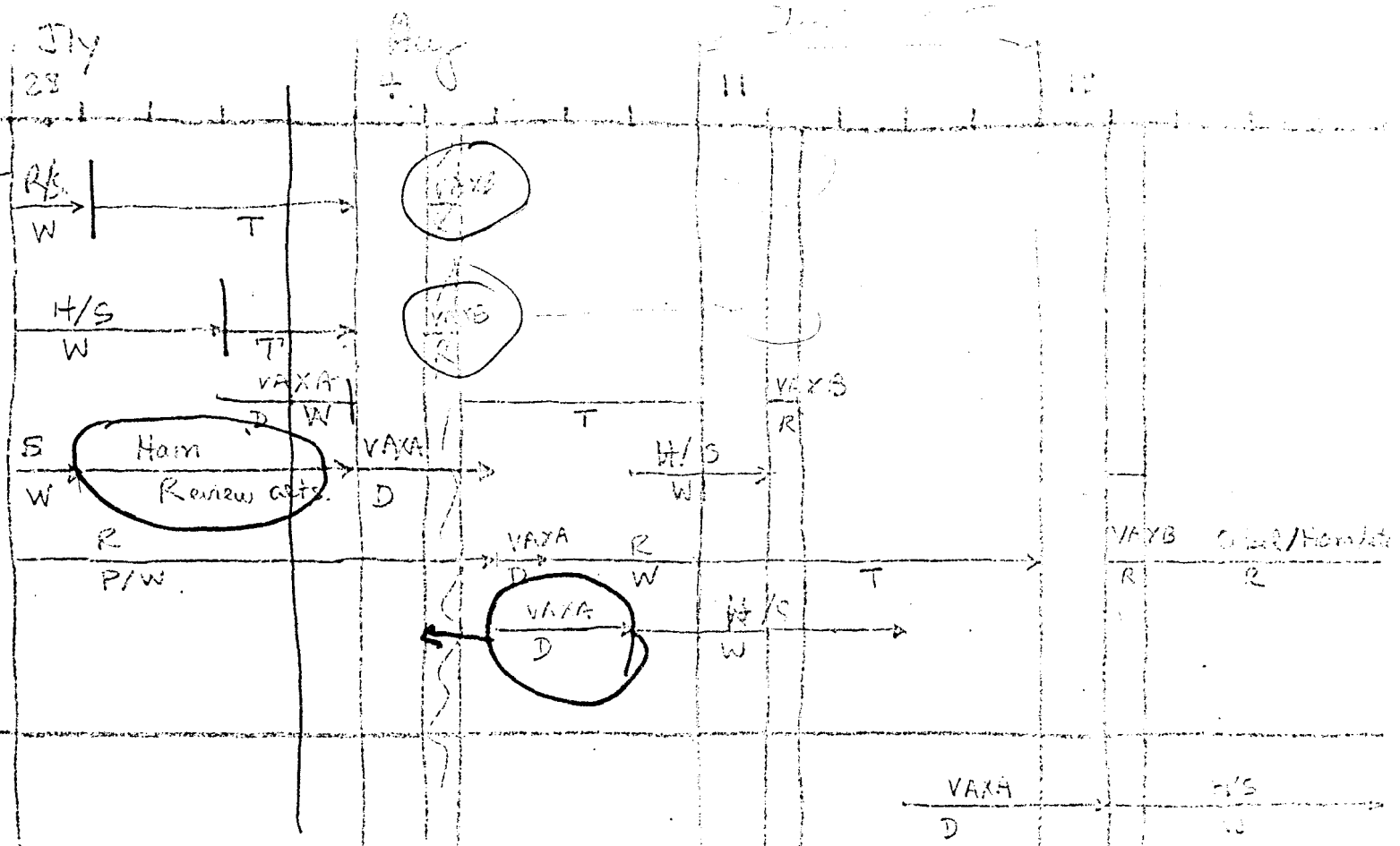
S Process Str.

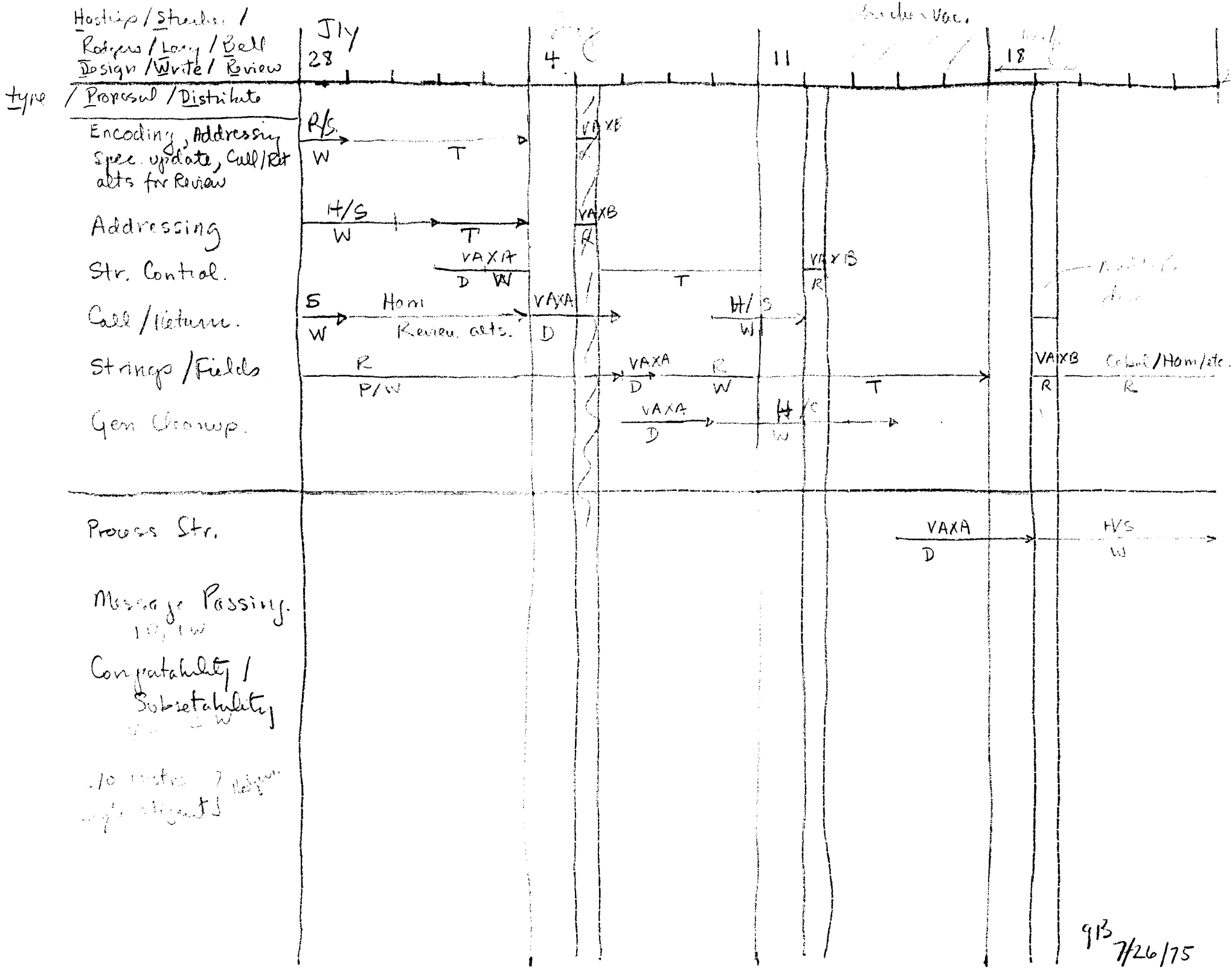
Linkage Passing.
ID, IW

R Compatability / Substatability

R I/O instruct.
I -1

M Multi-C
I -




Hastip / Strauber / Rodgers / Lacy / Bell
Design / Write / Review
type / Proposal / Distribute
JLY
28
4
11
18
Encoding, Addressing Spec. update, Call/Ret alts for Review
R/S
W
T
VAXB
Addressing
H/S
W
T
VAXB
R
Str. Control.
VAXA
D W
T
VAXB
R
Call / Returns.
S
W
Hami
Review alts.
VAXA
D
H/S
W
Strings / Fields
R
P/W
VAXA
D
R
W
T
VAXB
R
Gen. Cleanup.
VAXA
D
H/S
W
Process Str.
VAXA
D
H/S
W
Message Passing.
Compatability / Subsetability
GB 7/26/75

D I G I T A L INTEROFFICE MEMORANDUM

SUBJ: VAXB MEETING AGENDA--8/12/75

DATE: 08-07-75
FROM: GORDON BELL
EX: 2236
MS: ML12-1/A51

TO: FILE

Place: ML3-4 CENTRAL MILL CONFERENCE ROOM
Time: 8:30

8:30-10	VA Mechanism	Hastings
10-10:30	Field/Bits	Rodgers
10:30-11:30	Call/Return	Hastings/Conklin
11:30-12	Status of 75 Design Review Problems	Rodgers

VAXA--DOCUMENTATION FOR VAXB (BY 5:00, AUG. 8)

- Status of 75 Design Review Problems — Rodgers
- Call/Return, Field/Bits (handwritten) — Strecker
- VA Mechanism (whatever is ready by Friday evening) — Hastings

---

VAXA SCHEDULE WEEK OF 11 AUGUST

- Strecker - out
- Compatibility/supersetability Design (Rodgers) with Lipman, Stewart, Delagi, Gourd, [illegible]
- Process Structure/Interrupts/Traps Design with Lipman/Stewart

GB:mjk

Groups

- Coding conventions, assembler stuff
- Language generation/contracts —

- Process struct. + interrupts, + multi processors

- I/O.

- Compatibility

- Call / Return. + other contexts
- String.

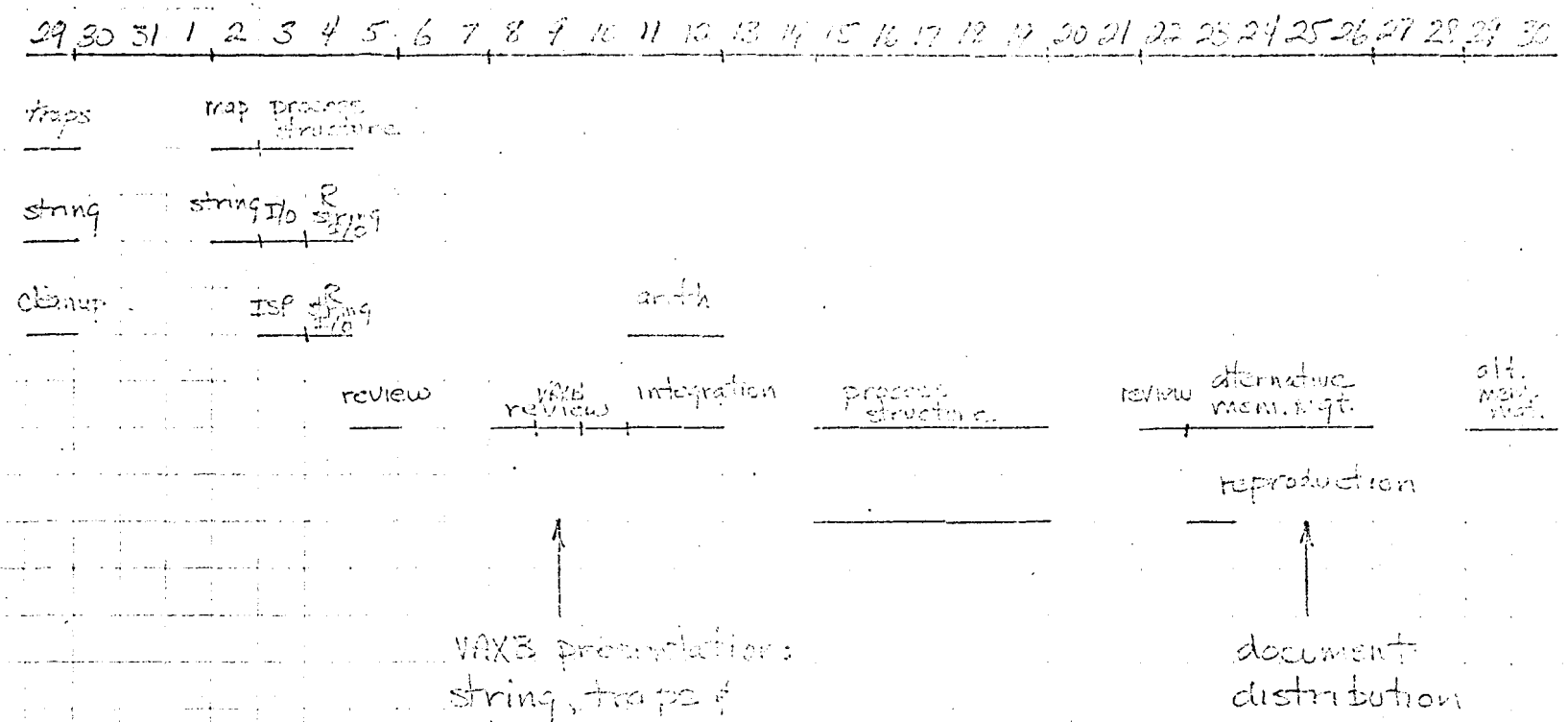
Jack Infants

- Op. Sys.

~~need i/o, alignment probs.~~

- Authentic. Group — Slack.

VAXA SCHEDULE 29 AUGUST THROUGH 30 SEPTEMBER



SOFTWARE SERVICES REPORT
Field Comment Sheet

MANAGER: Mike Craven DIVISION No. Cal. MONTH 2 Q 3 FY 75

Chuck Samuelson

TOPIC	COMMENTS
	Several customers I have dealt with in the past several months (LLL, LBL, NASA, SYNTEX, SRI) have expressed growing concern over some deficiencies in DEC's PDP 11 hardware. For the most part, I agree with them and would like to make a few comments, not in any particular order: 1. 11/45's FPU - When designed, the FPU was a very fast and powerful device. Today, it is no longer as effective when compared with our competition (DG Eclipse, MODCOMP IV, Varian 73 1/2), sometimes by a healthy margin. This will be a big disadvantage to the 11/70. Speed is the one criteria that always comes out in competitive situations. Any efforts to redesign the FPU should be of high priority. 2. Writable Control Store - There are good arguments pro and con for WCS. An increasing amount of customers are including WCS in their RFQ's. When all the arguments are weighed, WCS will sell computers. 3. Context Switching - There are 50 some registers to save and restore in the 11/45 -11/70 between FPU, memory management and general registers. These are not all pushed and popped off a stack, they are stored and retrieved in task headers, etc. DEC's argument is you really don't context switch that often. There are two points to remember. First, context switching is pure overhead and will always be a required part of an operating system. Second, and very valid to our customers: If you spend $200K for a system and it spends 10% of its time context switching (a reasonable number), you have just spent $20,000.00 to do context switching. Let us design some hardware to help us out here. It is certainly cost effective. 4. Memory Management - This is a large bag of tricks. All our sophisticated customers (and a lot of the rest) are well aware of our short comings here. The limitation of 8 memory pages, the limitation to 32K words of total address space to a program, the difficulty in implementing I & D space are all severe problems.

Extending the 11 User & Environment RL 9/3

THIS SHORT PAPER EXPLORES THE WAYS IN WHICH THE CURRENT USER ENVIRONMENT OF RSX-11D/M COULD BE EXTENDED ON FUTURE PDP-11 PROCESSORS WITH A LARGE VIRTUAL ADDRESS SPACE AND BETTER MEMORY MANAGEMENT FACILITIES. THE RSX-11M/D ENVIRONMENT IS THE OBVIOUS STARTING POINT AS RT-11 AND DOS DO NOT SUPPORT MEMORY MANAGEMENT AND THE RSTS ENVIRONMENT HAS NEVER BEEN DOCUMENTED.

SOME OF THE MODIFICATIONS HINTED AT IN THIS PAPER ARE NECESSARY TO PROVIDE MINIMAL SUPPORT OF THE EXTENDED ADDRESS SPACE, SOME ARE DESIGNED TO PASS ON TO THE USER SOME OF THE EXTENDED CAPABILITIES OF THE MACHINE, AND SOME REPRESENT MY PERSONAL PREJUDICES ABOUT THE PHILOSOPHIES BEHIND RSX-11.

EXECUTIVE CALL FORMATS

IN THE CURRENT RSX SYSTEM, THE EXECUTIVE IS CALLED VIA THE EMT 377 INSTRUCTION. THIS IS NICE IN THAT IT ALLOWS ALL OTHER TRAPS TO BE RETURNED BY THE EXEC TO THE USER, ALLOWING EMULATION OF OTHER ENVIRONMENTS. UNFORTUNATELY, THE ARGUMENTS TO THE EMT APPEAR ON THE TOP OF THE STACK IN TWO FORMS - AN ODD NUMBER INDICATES THE DIRECTIVE PARAMETER BLOCK (DPB) IS ITSELF ON THE STACK, WHILE AN EVEN NUMBER IS TAKEN AS A POINTER TO THE REAL DPB ELSEWHERE IN MEMORY. THIS SCHEME DOESN'T WORK WITH DOUBLEWORD ADDRESSES, AS THE HIGH ORDER PART OF A DOUBLEWORD ADDRESS MAY BE ODD. THERE ARE FOUR SOLUTIONS TO THIS.

1) HAVE THE HARDWARE STORE INDIRECT POINTERS LOW-ORDER FIRST - THIS IS UNACCEPTABLE AS IT ELIMINATES SHORT-FORM (16-BIT) INDIRECT WORDS.

2) REQUIRE RSX PROGRAMS TO PUSH DPB ADDRESSES SWAPPED - THIS IS KLUDGEY.

3) USE TWO EMT CODES - ONE FOR DPBS ON THE STACK AND ANOTHER FOR DPB POINTER ON THE STACK - THIS MIGHT SCREW SOME USERS AND IS ESTHETICALLY UNNICE.

4) PROVIDE ANOTHER MECHANISM FOR EXEC COMMUNICATION IN THE EXTENDED ENVIRONMENT. THE BEST WAY WOULD BE TO MAKE EXECUTIVE CALLS NORMAL SUBROUTINE CALLS TO A PART OF THE EXECUTIVE WHICH WOULD RESIDE IN EVERY TASK'S ADDRESS SPACE. THIS SCHEME HAS THE ADDITIONAL ADVANTAGE OF ALLOWING PART OF THE EXECUTION OF EXECUTIVE REQUESTS TO RUN AT THE CALLING TASK'S PRIORITY, HOWEVER IT WOULD REQUIRE PROTECTION MECHANISMS IN THE HARDWARE TO MAINTAIN A PROTECTED USER ENVIRONMENT. (E,G. PRIVELEGED SEGMENTS WITH RESTRICTED ENTRY VIA PORTALS.)

OBVIOUSLY, ALL POINTERS IN DPBS IN THE EXTENDED ENVIRONMENT MUST BE EITHER EXPLICIT OR IMPLICIT 32-BIT ADDRESSES. THIS EXTENDS THE DPBS THEMSELVES IN AN EQUALLY OBVIOUS WAY. TO MAINTAIN SOME COMPATIBILITY (AND EXECUTIVE SANITY) THESE ADDRESSES WILL REQUIRE DOUBLEWORDS EVEN IF THE SHORT FORM IS USED. ALSO, THE BASE ADDRESS AND LENGTH OF THE USER'S PARTITION ARE CURRENTLY RETURNED TO THE USER IN TERMS OF 32-WORD BLOCKS - SINCE THIS WILL BECOME A DOUBLEWORD ALSO, WE MIGHT AS WELL CHANGE THE UNITS TO BYTES TO ELIMINATE MEMORY MANAGEMENT DEPENDENCIES.

ENHANCEMENTS TO THE USER ENVIRONMENT

MOST OF THE FEATURES DESCRIBED BELOW COULD BE IMPLEMENTED WITH EXISTING HARDWARE, BUT THEY WOULD NOT BE AS GENERAL OR AS POWERFUL.

1) DYNAMIC CONTROL OF THE VIRTUAL ADDRESS SPACE - THIS HAS TWO ASPECTS - THE ABILITY TO CREATE AND DESTROY NAMED AREAS OF THE VIRTUAL ADDRESS SPACE AND SHARE THEM WITH OTHER TASKS, AND THE ABILITY TO DYNAMICALLY EXPAND AND CONTRACT THESE AREAS (IF THEY ARE PRIVATE TO YOUR TASK). RSX CURRENTLY HAS A CONCEPT OF NAMED SEGMENTS (CALLED "LIBRARIES" OR "GLOBAL COMMON") BUT IT IS STATIC - THE SEGMENTS AND THEIR LENGTHS MUST ALL BE NAILED DOWN AT TASK BUILD TIME.

2) FAST LOCK MECHANISM

RSX-11 CURRENTLY IMPLEMENTS NO FACILITY FOR TASKS TO SYNCHRONIZE ON SHARED RESOURCES. A MECHANISM HAS BEEN PROPOSED FOR THIS WHICH IS QUITE NICE IN TERMS OF GENERALITY BUT REQUIRES AN EXECUTIVE CALL AND A FAIR AMOUNT OF OVERHEAD (DEPENDING ON HOW MANY RESOURCES ARE CURRENTLY BEING LOCKED). IF WE ARE SERIOUS ABOUT BUILDING A GENERAL MULTIPROCESSOR SYSTEM WE SHOULD PROVIDE A LOW-OVERHEAD LOCK MECHANISM WHICH OPERATES ON WORDS IN MEMORY SHARED BY THE TASKS. THIS MECHANISM WOULD NOT REQUIRE AN EXECUTIVE CALL IF THE HARDWARE INSTRUCTION WHICH GRABBED THE LOCK SUCCEEDED, NOR WOULD IT REQUIRE A CALL ON UNLOCKING UNLESS SOMEONE WAS WAITING FOR THE RESOURCE.

3) INTERTASK COMMUNICATION (MESSAGES)

THE CURRENT RSX ENVIRONMENT ALLOWS TASKS TO SEND MESSAGES TO EACH OTHER BY BUFFERING THE MESSAGE IN THE EXECUTIVE FREE STORAGE POOL UNTIL THE RECEIVER IS READY TO RECEIVE IT. THE SIZE OF MESSAGES IS LIMITED - 13 WORDS IN RSX-11M AND PRE-V6 RSX-11D, "VARIABLE" BUT LIMITED BY NODE POOL USAGE CONSTRAINTS IN RSX-11D V6. THIS METHOD HAS THREE DISADVANTAGES - IT LIMITS THE SIZE OF MESSAGES, IT IS SLOW (MESSAGES ARE MOVED TWICE) AND IT CAN CHEW UP THE EXECUTIVES FREE CORE POOL (ESPECIALLY IF A BOTTLENECK EXISTS AT ONE RECEIVER.).

THERE ARE SEVERAL WAYS OF SENDING LARGE MESSAGES BETWEEN TASKS MORE EFFICIENTLY. SOME OF THEM ARE:

A) BUILD MESSAGES IN A COMMON AREA, PASS ADDRESSES - THIS WORKS AND COULD BE DONE USING THE EXISTING STRUCTURE - IT'S DISADVANTAGES ARE IT HAS PROTECTION PROBLEMS, IT'S IMPOSSIBLE TO BACK UP MESSAGE OVERFLOW TO DISK EASILY, AND IT'S NOT CLEAR HOW MANY OF THESE COMMON PARTITIONS YOU NEED.

B) BUILD MESSAGES IN CREATED SEGMENTS, PASS THEIR NAMES - THIS SOLVES THE PROTECTION AND DISK BACKUP PROBLEMS BUT REALLY POUNDS ON THE ABILITY TO CREATE SEGMENTS - IT PROBABLY ONLY WINS FOR EXTREMELY LARGE MESSAGES.

C) PASS PAGES - THIS REQUIRES A NEW EXECUTIVE CALL AS THE EXEC WILL HAVE TO DELETE THE PASSED PAGES FROM THE SENDERS ADDRSS SPACE AND

INSERT THEM SOMEWHERE IN THE RECEIVERS ADDRESS SPACE AND PASS HIM THE VIRTUAL START ADDRESS AS THE MESSAGE. IT IS EFFICIENT AND PROTECTED AND CAN BE BACKED UP TO DISK. IT MAY REQUIRE THAT THE MONITOR HANDLE SPARSE PAGES WITHIN A SEGMENT.

4) FASTER CONTEXT SWITCHING

HARDWARE CONSIDERATIONS ASIDE, CONTEXT SWITCHING IN RSX-11D (AND RSX-11M TO A LESSER EXTENT) IS SLOW BECAUSE IT'S EVENT FLAG STRUCTURE REQUIRES THE SYSTEM TO SCAN THE ACTIVE TASK LIST ON EVERY CONTEXT SWITCH. THE ADDITION OF A SIMPLE DIRECTIVE WOULD RELIEVE THE WORST PART OF THIS PROBLEM; HOWEVER TO ACHIEVE THE BEST SOLUTION THE CONCEPTS OF "SIGNIFICANT EVENT" AND "GLOBAL EVENT FLAG" WOULD HAVE TO BE MODIFIED.

5) FASTER INTERRUPT HANDLING BY USER TASKS

THE PROCESS-STYLE INTERRUPT SOLVES MOST OF THIS PROBLEM. THERE IS STILL AT LEAST ONE DIFFICULTY THAT I CAN SEE, THOUGH - IN ORDER NOT TO SCREW UP THE MONITOR, USER-MODE INTERRUPT PROCESSSES MUST RUN AT A HIGHER HARDWARE PRIORITY THAN ANY PART OF THE SYSTEM WHICH "KNOWS" WHO THE CURRENT USER IS. THIS INCLUDES THE SYSTEM SCHEDULER, THEREFORE THE SYSTEM WILL NOT BE ABLE TO STOP AN INTERRUPT PROCESS WHICH LOOPS.

PERSONAL GRIPES

THIS WILL BE BRIEF - IT'S LATE AND MY FINGERS HURT.

1) THE NOTIONS OF PROGRAM AND PROCESS ARE ALL LUMPED TOGETHER IN A VERY SMARMY WAY.

2) THE CONCEPT OF "I/O IN PROGRESS COUNT" SHOULD EXIST FOR PARTITIONS (SEGMENTS) AS WELL AS FOR TASKS TO ALLOW CHECKPOINTING OF UNAFFECTED PARTS OF A TASK

MT please give me a copy

SUBJ: COS.PERFORMANCE DATE: 03-31-75 PAGE 1
FROM: GORDON BELL

~~Hal~~ Rollins ! Please return this. Also

PLEASESEND TO: ROLLINS TURNER ML3/E44

APR 04 1975

I need copies, in general, of all memos on this subject I write.

SUBJ: Cos.performance CHARACTERIZING THE PERFORMANCE OF RSX11/M, D, RSTS, TOPS 10, and VIROS COMPONENTS

To: Rollins Turner, Peter Christy, Larry Wade

CC: VAXA

In trying to understand the kinds of modifications we might make to the 11 in terms of instruction-set, structure, implementation(s), operating system, etc., we need to know something about the nature of monitors (operating systems), and specifically their performance. It is difficult to completely characterize and model them, but we do need a model in order to go after, and hopefully get, a significant improvement. Can you characterize these in terms of capability, size, residency, and performance values for various components?

This should be both from a user's viewpoint, and an internal structure.

For example, what file structures are supported, what is mapping on files, and what is the performance (in accesses), and in time for various disks? What terminal modes are supported, what is maximum rate and what is cost (time)?

GB:mlk

Gordon,

Larry Wade and I are working on this. We will compile the available information on capability, size, and residency. Performance is a much more difficult issue, and there is no data available that permits comparisons of all five systems on the same components. I will list what information is available, and sketch out a proposal of what I think would be needed to make reasonable comparisons of the systems in terms of performance.

Rollins Turner

digital INTEROFFICE MEMORANDUM

TO: Gordon Bell
VAXC

CC: Rollins Turner
Larry Wade

DATE: April 25, 1975

FROM: Roger Gourd

DEPT: Software Engineering

EXT: 5127 LOC: ML5/E35

SUBJ: RE: TURNER/WADE 9 APRIL 1975 MEMO ON COMPARISON OF SYSTEMS

I appreicate the effort that Rollins and Larry went through to get the data they compiled for their comparison memo. I have accumulated data like that before and it is not easy. However, the casual reader could well be mislead by the content as I beleive I was, and therefore I want to respond, line by line, to question and try to correct certain entries.

As to table format, I believe that the five operating system headings must have descriptive nomenclature. What version of each system, or release, is implied?

RSX-11M version?
RSX-11D version? /IAS release?
RSTS-E version?
TOPS-10 5.07, 6.01, 6.02?
SNARK release?

Without such supportive information, one would be mislead as to existing and/or intended features at any point in the product's life.

On the above basis, I will describe my interpretation of the line items, relative to the systems with which I am familiar.

Real Time - SNARK will, in release 2 or 3, dominate TOPS-10 in real time capabilities

CCL - Corporate command language is not scheduled for SNARK release 1 but is committed for release 2.

Overlays - SNARK uses a single segment COBOL compiler and does not overlay.

Multiprocessor Support - Since this line requires a definition of the term, here goes - multiprocessor: a system of closely coupled processors running an operating system on which it is transparent to a user job or task which processor is executing that job. This definition sells computer science a little short but should suffice for DEC products. With the above in mind, none of the five operating systems have multiprocessing support. IPL's won't ship until December 1975 and have no

implication to multi-processing anyway. TOPS-10 master/slave systems are not multiprocessors, although the definition needs less bending for that product than any other.

On-Line Diagnostics - TOPS-10 on the 1080 will have some (I don't know how much) on-line diagnostic capabilities.

Logical Record - The record access facility (RAF) under Gil Steil and Sumner Blount will provide logical record capability for SNARK.

Buffered - "Buffered" file access is an access method I have never heard of or read about in our literature.

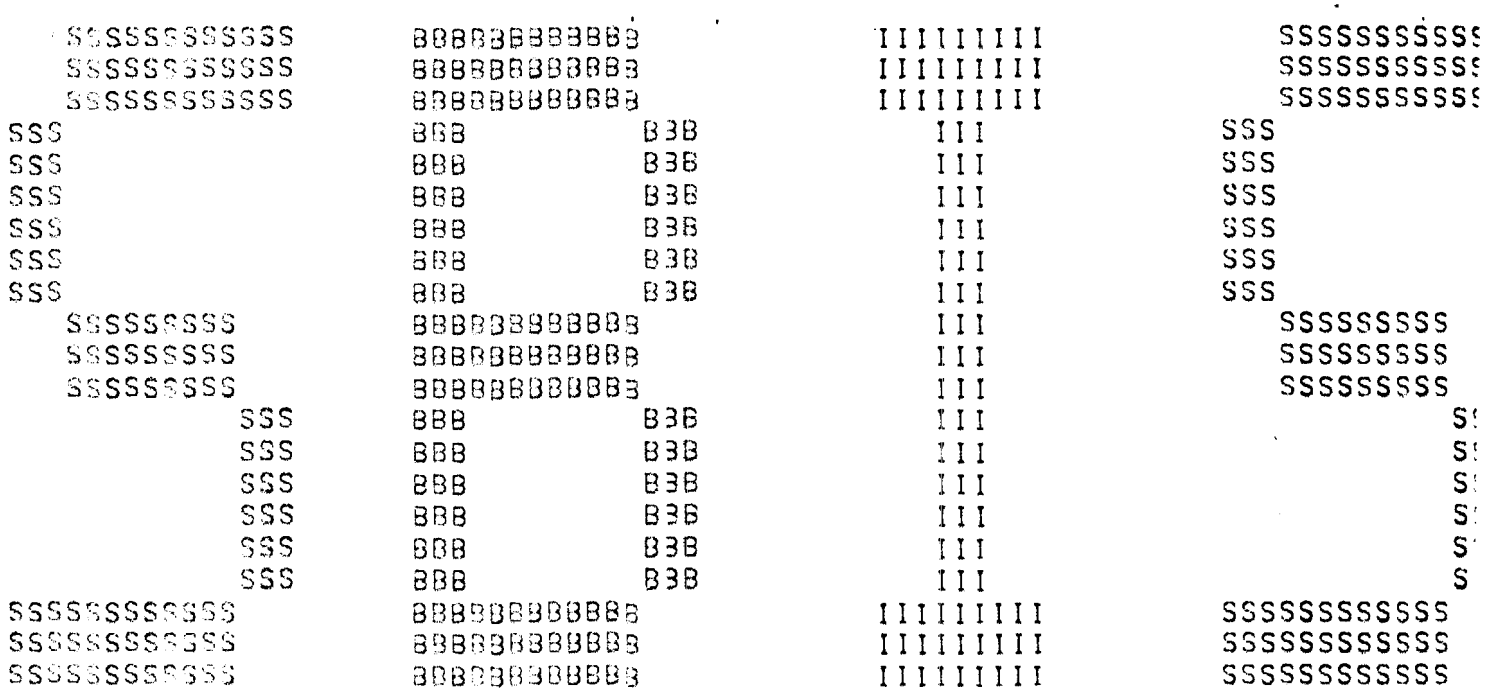
Removeable Structures - SNARK release 2 will support this.

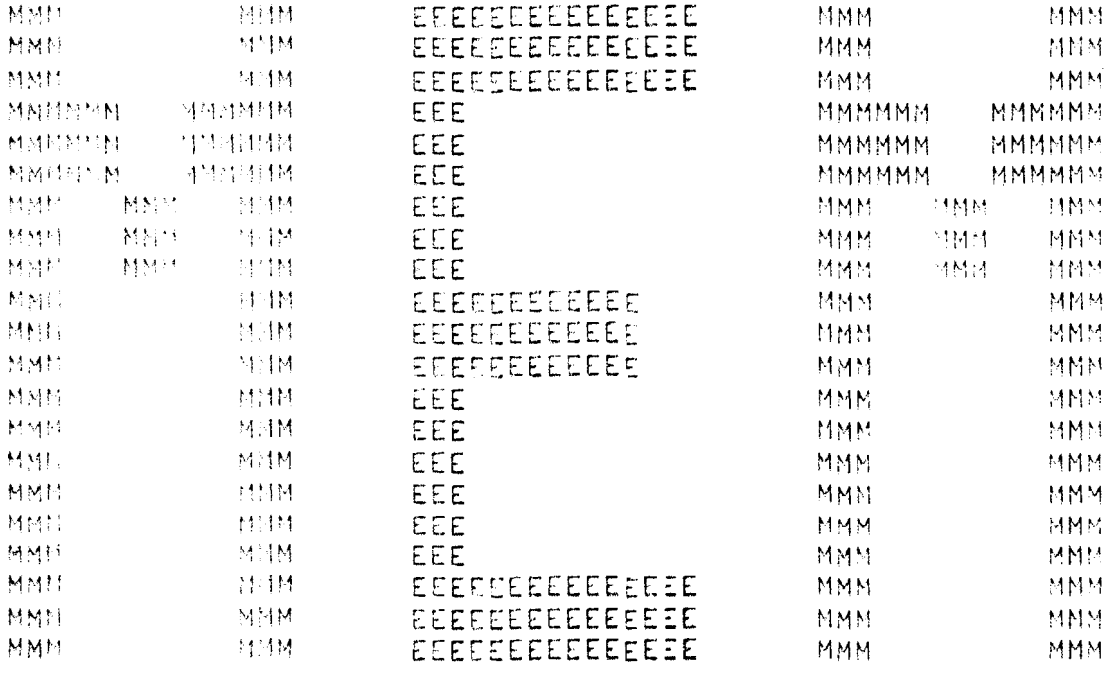
Multi-Volume Structures - Ignore magnetic tape for a moment and just consider disks. SNARK today has only one "structure" and that is multivolume, defining a volume as a disk pack. Release two will support removeable structures and therefore removeable volumes since a structure may be more than one volume. RSTS/E supports a multivolume structure but not multivolume files, another interesting aspect of confusion for a definition.

IPL - How can RSX-11M or RSX-11D/IAS support an an interprocessor link when we won't have one to ship until December?

Front End - SNARK supports a front end (11) as the character device concentrator for the DECsystem20 and the front end is an inseperable piece of the system. TOPS-10 supports a front end (11) on the 1080.

/gml





```
LPTSPL Version 6(344)  Running on LPT1
 *START* User RODGERS,DAVI [142,3041] Job  SBISPC Seq. 14823 Date 29-Jan-75
Request created: 29-Jan-75  13:41:36
File: DSK:SBISPC.MEM[142,3041] Created: 27-Jan-75 18:47:00 <077>  Printed:
QUEUE Switches:  /PRINT:ARROW /FILE:ASCII /COPIES:5 /SPACING:1 /LIMIT:176 /F
```

Revision 1 - January, 1974 Revision 2 coming soon

1.0 INTRODUCTION

1.1 Definition

The Synchronous Backplane Interconnect is a data path and communication protocol for information transfer among the elements of a data processing system. Information may be exchanged between central processor and memory, I/O controller and memory or CPU and I/O controller. The communication protocol allows the data path to be time-multiplexed such that an arbitrary number of information exchanges may be in progress simultaneously.

1.2 Goals

Specifically designed to meet the requirements of the UNICORN system, the Synchronous Backplane Interconnect (SBI) provides checked, parallel data transfer synchronous with a common system clock. In each clock period or cycle, interconnect arbitration, data transfer and transfer confirmation may occur in parallel. Using a 100 nanosecond clock period, the SBI achieves a maximum data transfer rate of 2.77 million words per second.

1.3 Arbitration

Arbitration logic, distributed among all system components, determines which unit of those requesting access to the interconnect in a particular cycle will perform a data transfer in the following cycle.

1.4 Data Path Usage

The same data transfer path is used to convey command and address words, information words, interrupt summary read words and interrupt summary response words. Each exchange consists of two data transfers; one command/address word and one information word or one interrupt summary read word and one interrupt summary response word. For write type commands, the commander uses two successive cycles, transmitting command/address then information. Read type commands also begin with a command/address transfer from the commander, however, information emanates from the responder and may be delayed by whatever access time is characteristic; responder's information transfer, like all others, is synchronous with the system clock. Interrupt summary exchanges begin with an interrupt summary read transfer from the

central processor and are completed two cycles later with an interrupt summary response transfer.

1.5 Confirmation

Each data transfer, command/address or information, is confirmed by the receiver exactly two cycles after transmission. Both transfers of a write type command are confirmed by the responder. Responder confirms the command/address transfer and commander confirms the information transfer in read type exchanges. Confirmation lets the transmitter know whether the data transfer was correctly received and, in the case of a command/address transfer, whether the receiver can process the command.

1.6 Control Signals

In addition to signals for arbitration, data transfer and confirmation, the SBI includes priority interrupt request signals and control signals to synchronize and provide notification of changes in system state.

1.7 Data Path Width

The SBI is word oriented (rather than block or byte) with the word size chosen to match the system components. The UNICORN requires 36-bit words.

2.0 SIGNAL DESCRIPTION

2.1 Signal Types

The Synchronous Backplane Interconnect consists of 73 signals. Divided by function into five classes, there are 16 arbitration lines, 46 data transfer lines, 3 confirmation lines, 4 control lines and 4 interrupt request lines. See figure 1.

2.2 Arbitration Line Assignment

Arbitration lines, TR<ØØ:15>, are assigned one per physical connection to establish fixed priority access to the data path. Priority increases from TR15 to TRØØ which is reserved for use as a HOLD signal by units requiring the two adjacent cycles of a write type exchange.

2.3 Arbitration Line Use

To acquire control of the data path, a unit asserts its assigned Transfer Request line at the beginning of a cycle. At the beginning of the next cycle, the unit must examine the state of all TR lines of higher priority and if none are asserted it will remove its TR and assert data path signals. If higher priority TR's are present, then the unit's TR remains asserted and the test is made again at the start of the next cycle.

2.4 Number Of Units Arbitrating

A total of 16 physical connections may be used. The 15 highest priority units operate TR<01:15>; the lowest priority unit needs no actual TR signal. The lowest priority level is reserved for the central processor.

2.5 Data Transfer Lines

Four subfields comprise the data transfer path: parity check (P<Ø:1>), information tag (TAG<Ø:2>), source/destination identity (ID<Ø:4>) and information bits (B<ØØ:35>). Parity check bits provide redundancy for detecting single bit (all odd numbers) errors in the data path. Transmitting units generate PØ as even parity for TAG<Ø:2> and ID<Ø:4> and P1 as even parity for B<ØØ:35>. When no units are transmitting, the data transfer path assumes an all zeros state; thus, P<Ø:1> should always carry even parity in the absence of errors.

2.6 TAG Determines The Format

TAG<0:2> are asserted by the transmitting unit to indicate the type of information being transmitted. The interpretation of ID<0:4> and B<00:35> by a receiving unit is determined by the TAG bits.

2.7 All Units Receive Every Transfer

In each cycle, all units examine the data transfer lines to determine whether they are the intended receiver of the information. The type of information present determines how this decision is made. Six information types are defined. See figure 2.

2.8 Command/Address TAG

TAG<0:2> = 100 specifies that B<00:35> are a command/address word and that ID<0:4> is a unique code identifying the logical source (commander) of the command.

2.9 Read Data TAG

TAG<0:2> = 011 indicates that B<00:35> contain data solicited by a previous read type command and that ID<0:4> identifies the logical destination for which the data is intended. This will be the unique code which was received with the read type command.

2.10 Write Data TAG

TAG<0:2> = 101 flags B<00:35> as write data and ID<0:4> as the logical source. A transmission of this type always occurs in the cycle following a write command transmission and the ID fields of both transmissions are identical.

2.11 Corrected Read Data TAG

TAG<0:2> = 001 identifies a transmission in response to a previous read type command. B<00:35> contain the data solicited by the command and ID<0:4> indicate the logical destination as for TAG = 011; however, the data accompanying TAG = 001 is flagged as corrected by ECC (error correcting code) logic.

2.12 Read Data Substitute TAG

TAG<2:0> = 000 denotes a transmission which substitutes for data solicited by a read type command. When ECC logic is unable to correct the requested data, TAG = 000 is used to indicate this condition. D<00:35> contain, if possible, the uncorrected data or other meaningful information and ID<0:4> specify the identity of the read commander as for TAG = 011 and TAG = 001.

2.13 Interrupt Summary Read TAG

TAG<2:0> = 111 defines B<00:35> as the level mask for the special INTERRUPT SUMMARY READ command used to identify the origin of an interrupt request. ID<0:4> identify the commander, usually a CPU.

2.14 Unused TAG Codes

TAG<2:0> = 010 and TAG<2:0> = 110 are unused but reserved for future use.

2.15 Information Source/Destination Identifier

As described above, ID<0:4> identify either the logical source or the logical destination of the information contained in D<00:35>. In general, the assignment of ID codes to units is not fixed since the association between any particular code and the commanding unit is established with the transmission of the command/address word. Note that this implies that ID codes are assigned to units which issue command/address words. Any physical unit may use more than one logical ID to originate data exchanges, however, the controlling logic must be sufficiently sophisticated to tolerate responses to read type commands which deviate from the order in which the commands were issued.

2.16 Reserved ID Code

Of the 32 possible ID codes, the all zero code is reserved so that the idle state of the interconnect (READ DATA SUBSTITUTE, destination ID = 0) causes no unit to be selected. Note that even though no units are selected, all are checking for correct SBI parity.

2.17 Command/Address Format

Information bits B<Ø0:35> carry the payload of the SBI. Information appears on these lines in command/address format, data format, interrupt summary read format or interrupt summary response format. In command/address format, information is grouped in two fields: F<Ø:3>, the function code and A<Ø9:35>, a 27-bit physical address. Unused bits are ignored but transmitted as zeros. See figure 3.

2.18 Command Functions

Four functions are defined: READ, WRITE, INTERLOCK READ and AND INTERLOCK WRITE.

2.19 Read Function

Encoded into F<Ø:3> as ØØØ1, READ instructs the unit selected by A<Ø9:35> to retrieve the addressed data and transmit it to the ID accompanying the command as READ DATA (TAG = Ø11) or as CORRECTED READ DATA (TAG = 101). If this is impossible due to uncorrectable data error or other catastrophic failure, the addressed unit shall transmit READ DATA SUBSTITUTE (TAG = ØØØ) instead. No fixed time is established for either response but the commander after an interval of 1ØØ milliseconds without response may assume catastrophic error.

2.20 Write Function

WRITE, F<Ø:3> = ØØ1Ø, instructs the unit selected by a<Ø9:35> to modify the storage element addressed using data transmitted in the next succeeding cycle with TAG = 101.

2.21 Interlock Read Function

F<Ø:3> = Ø1Ø1 denotes the INTERLOCK READ function. This instruction, used for process synchronization, causes the unit selected by A<Ø9:35> to retrieve and transmit the addressed data as for READ. In addition, this command causes the selected unit to set an INTERLOCK flip-flop. While INTERLOCK is set the unit will respond with BSY confirmation to INTERLOCK READ commands. INTERLOCK is reset on receipt of an INTERLOCK WRITE function.

2.22 Interlock Write Function

F<0:3> = 0110, the INTERLOCK WRITE function, instructs the unit selected by A<29:35> to modify the storage element addressed using data transmitted in the succeeding cycle with TAG = 101 exactly as for WRITE. Additionally, the INTERLOCK flip-flop is reset.

2.23 Unused Function Codes

Function codes 0000, 0011, 0100 and codes 0111 through 1111 are unused but reserved for future use.

2.24 Physical Address Space

The 27 physical address bits define a 134,217,728 word address space which is divided into two equal sections. Addresses 0 through 377777777(8) (A09 = 0) are reserved for primary memory. Addresses 400000000(8) through 777777777(8) (A09 = 1) are reserved for device control registers, system configuration information and CPU status registers. Generally, primary memory begins at address 0, the address space is dense and consists only of storage elements. In contrast, the control address space is sparse with address assignments based on device type and access may have side effects.

2.25 Confirmation Lines

The confirmations lines, CNF<0:1> and FAULT , provide a signal path from the receiver to transmitter two cycles after each data transmission. Confirmation is delayed to allow data path signals to propagate, to be received, checked and decoded by all receivers and to be generated by the responder. During each cycle, every unit in the system receives, latches and makes judgements on the data transfer signals. See figure 4. Barring multiple bit transmission errors or unit malfunction, only one (or none) of the units receiving the data path signals will recognize an address or ID. This unit will assert CNF<0:1> as appropriate. Any or all units may assert FAULT for any of several reasons; assertion of FAULT indicates protocol or data path failure.

2.26 Confirmation Codes

CNF<0:1> may legitimately assume three values: 00 (N/R), the unasserted state, indicates no response to selection; 01 (ACK) is a positive acknowledgement to any transfer; 10 (BSY) in response to a command/address transfer indicate successful selection of a unit presently unable to execute the command.

2.27 Unused Confirmation Code

CNF value 11 is never transmitted but is treated as no response (N/R). BSY (10) response to other than command/address transfers is also treated as no response.

2.28 Use Of Confirmation

Transmitting units sample the CNF and optionally FAULT lines at the beginning of the third cycle after transmission. ACK is the expected response indicating either that the command will be executed or the data has been correctly received. Command/address transfers may from time to time receive BSY confirmation. The commander should be prepared to repeat the command transmission until it is accepted. No response confirmation should be treated as abnormal and invoke contingency mechanisms.

2.29 WRITE Command Confirmation

Because WRITE operations consist of two successive transfers, acknowledgement is somewhat more complex. If the command/address transfer is not positively confirmed (ACK) then the command should be aborted and no notice should be taken of the data transfer confirmation. If ACK is not received as confirmation for WRITE DATA then the command should be aborted. In either case action consistent with other data transfer failures should be initiated.

2.30 Fault Detection

As shown in figure 4, any of several conditions may cause a unit to assert FAULT. FAULT[A], data path parity error may be generated by one or more units depending on the origin of the problem causing the calculated P<0:1> to disagree with the received P<0:1>. FAULT[B] and FAULT[C] result when a unit which received a WRITE or INTERLOCK WRITE command in the immediately preceeding cycle does not receive the anticipated WRITE DATA from the same transmitter in the following cycle FAULT[D] is indicated

when a unit which has not issued a READ or INTERLOCK READ command receives a response to a read type command. FAULT[E] arises when a unit receives an INTERLOCK WRITE command and INTERLOCK has not been set by an INTERLOCK READ command. FAULT may also be asserted by a transmitting unit which detects multiple transmitters in the same cycle.

2.31 Control Lines

Of the four control signals, one is used to synchronize system activity and three provide specialized global system communication.

2.32 ALERT Signal

ALERT is asserted for one cycle by any physical connection which has detected failing power or the resumption of power. A unit losing power must control all drivers and receivers during the transition to unpowered so that no disruption of communication occurs. After signalling power loss, the unit must indicate in a status register the power failing condition as long as feasible. A unit regaining power also asserts ALERT for one cycle and then resumes normal activity. CPU's in the system respond to the ALERT by determining the new system configuration and taking appropriate action.

2.33 DEAD Signal

DEAD is a fail-safe signal asserted when interconnect activity is non-viable due to termination power loss or system clock failure. Assertion of DEAD precludes assertion of any signal by any unit including ALERT. The presence or absence of system clock while DEAD is asserted is not guaranteed. At least ←← clock cycles will precede the negation of DEAD.

2.34 UNJAM Signal

UNJAM is provided as a direct clear signal to cause logic to be reset to a known state when ordinary initialization fails. UNJAM is accessible only to the field service engineer and to diagnostic microcode. It shall be impossible to cause the assertion of UNJAM through the macrocode. UNJAM is asserted for one cycle.

2.35 CLK Signal

The CLK signal provides a universal timebase for all units connected to the SBI. As shown in figure 5, CLK defines two instants in each cycle: T0 and T3. At T0, the beginning of the cycle, a transmitting unit enables its drivers; at T3, all units clock the SBI signals into latches. The time from T0 to T3 is determined by maximum cable delay and driver-receiver propagation delay. T3 to T0 time, while bounded by receiver propagation delay, is fixed by the propagation delay of the logic required for SBI arbitration.

2.36 Interrupt Request Lines

Four interrupt request lines, REQ<1:4>, are used by units which must invoke a CPU to service some condition such as transfer completion for an I/O controller. Synchronous with the SBI clock, requesting units assert one of REQ<1:4> to signal a processor that attention is required. Any of the REQ lines may be asserted simultaneously by more than one unit and any combination of lines may be asserted by the collection of requesting units.

2.37 Interrupt Summary Read

At a time judged appropriate by the CPU responding to the REQ lines, the CPU will issue an INTERRUPT SUMMARY READ command (TAG = 111) with B<01:04> containing a single one bit corresponding to the REQ line being serviced. B00 and B<05:35> are zero. Service of REQ1 is indicated by B01 and so on. Units receiving the INTERRUPT SUMMARY READ command without error and asserting the REQ line specified by the one bit in B<01:04> assert, with the timing of CNF, two one bits in B<01:33>. Responding units do not assert TR, ID, TAG or CNF lines; Units detecting incorrect parity assert FAULT. The pair of bits asserted, in corresponding positions in the left and right half words, uniquely identifies the unit among those using the particular REQ line. For convenience, only 15 bit pairs B01-B19 through B15-B33 are used. Since pairs of bits are asserted, parity remains correct.

2.38 CPU Response To Interrupt

HOLDing control of the SBI using TR00, the CPU waits two cycles after the command is transmitted before latching B<00:35> in an internal register. By encoding the REQ level and bits received from responding units, the CPU generates a vector unique to the level and unit which is used to invoke the unit's service routine. The program invoked must take some explicit action such as reading or writing a device register to cause the unit to

de-assert the REQ line. Usually, the CPU will service requests of REQ1 before REQ2, REQ3 and REQ4. Similarly, units identified by B01-B19 are usually serviced before those using B02-B20, B03-B21, etc. If multiple units are requesting using the same REQ line, multiple INTERRUPT SUMMARY READ commands will be issued until all units have been serviced and the REQ is no longer asserted.

3.0 ELECTRICAL CHARACTERISTICS

3.1 Interface IC's

The SBI may be implemented using either SSI or LSI components. The LSI implementation includes registers and parity networks common to all SBI taps.

3.2 Discrete Bus Interface

The discrete interface will be via the AMD 26S10 or equivalent. The receiver threshold is 2.0 +/- 0.2 volts. The driver offset voltage is 0.8 volts maximum at 100 mA sink current.

3.3 LSI Bus Interface

The LSI chip which interfaces the bus is the DEC 8646. The specifications of the 8646 are still being negotiated.

3.4 Bus Cables

Bus cables will have the following properties:

Characteristic impedance:	75 +/- 7 Ohms
Propagation delay:	1.6 +/- 0.1 nS/ft.
Near end crosstalk coefficient:	0.018
Far end crosstalk coefficient:	0.016

3.5 Backpanel Characteristics

The backpanel will have the following properties:

Backpanel impedance:	75 Ohms
Backpanel propagation delay:	2.0 nS/ft.
Backpanel crosstalk:	to be determined.

3.6 Connections

Module to backpanel: these connections should be as short as possible, in any case less than 10 centimeters. Not more than one IC may connect to any one bus line per interface. Backpanel to cable: this connection will be made as directly as possible. More complete specification awaits further work on packaging.

3.7 Terminations

The terminations will be physically at each end of the bus, and each will be active, i.e., a voltage divider to +5 volts.

3.8 Signal Voltages

Discrete (SSI) signal voltages will be nominally 3.9 volts and 0.7 volts for logical zero and logical one respectively. For LSI implementations, the signal voltages will be 2.9 volts and 0.7 volts.

3.9 Timing

The nominal clock period is 180 nanoseconds. T0 to T3 interval is 150 nanoseconds and T3 to T0 interval is 30 nanoseconds.

```
   SSSSSSSSSSS       BBBBBBBBBBB          IIIIIIIII          SSSSSSSSS
   SSSSSSSSSSS       BBBBBBBBBBB          IIIIIIIII          SSSSSSSSS
   SSSSSSSSSSS       BBBBBBBBBBB          IIIIIIIII          SSSSSSSSS
SSS                  BBB         B3B        III        SSS
SSS                  BBB         B3B        III        SSS
SSS                  BBB         B3B        III        SSS
SSS                  BBB         B3B        III        SSS
SSS                  BBB         B3B        III        SSS
SSS                  BBB         B3B        III        SSS
   SSSSSSSSS         BBBBBBBBBBB             III           SSSSSSSSS
   SSSSSSSSS         BBBBBBBBBBB             III           SSSSSSSSS
   SSSSSSSSS         BBBBBBBBBBB             III           SSSSSSSSS
            SSS      BBB         B3B        III                    S
            SSS      BBB         B3B        III                    S
            SSS      BBB         B3B        III                    S
            SSS      BBB         B3B        III                    S
            SSS      BBB         B3B        III                    S
            SSS      BBB         B3B        III                    S
SSSSSSSSSSS          BBBBBBBBBBB          IIIIIIIII          SSSSSSSSSSS
SSSSSSSSSSS          BBBBBBBBBBB          IIIIIIIII          SSSSSSSSSSS
SSSSSSSSSSS          BBBBBBBBBBB          IIIIIIIII          SSSSSSSSSSS


MMM          MMM     EEEEEEEEEEEEEEE     MMM          MMM
MMM          MMM     EEEEEEEEEEEEEEE     MMM          MMM
MMM          MMM     EEEEEEEEEEEEEEE     MMM          MMM
MMMMMM    MMMMMM     EEE                 MMMMMM    MMMMMM
MMMMMM    MMMMMM     EEE                 MMMMMM    MMMMMM
MMMMMM    MMMMMM     EEE                 MMMMMM    MMMMMM
MMM   MMM   MMM      EEE                 MMM   MMM   MMM
MMM   MMM   MMM      EEE                 MMM   MMM   MMM
MMM   MMM   MMM      EEE                 MMM   MMM   MMM
MMM          MMM     EEEEEEEEEEEEE       MMM          MMM
MMM          MMM     EEEEEEEEEEEEE       MMM          MMM
MMM          MMM     EEEEEEEEEEEEE       MMM          MMM
MMM          MMM     EEE                 MMM          MMM
MMM          MMM     EEE                 MMM          MMM
MMM          MMM     EEE                 MMM          MMM
MMM          MMM     EEE                 MMM          MMM
MMM          MMM     EEE                 MMM          MMM
MMM          MMM     EEE                 MMM          MMM
MMM          MMM     EEEEEEEEEEEEEEE     MMM          MMM
MMM          MMM     EEEEEEEEEEEEEEE     MMM          MMM
MMM          MMM     EEEEEEEEEEEEEEE     MMM          MMM


LPTSPL Version 6(344)  Running on LPT1
 *START* User RODGERS,DAVI [142,3041] Job  SBISPC Seq. 14823 Date 29-Jan-75
Request created: 29-Jan-75  13:41:36
File: DSKB2:SBISPC.MEM[142,3041] Created: 27-Jan-75 18:47:00 <077>  Printed:
QUEUE Switches:  /PRINT:ARROW /FILE:ASCII /COPIES:5 /SPACING:1 /LIMIT:164 /F
```

SYNCHRONOUS BACKPLANE INTERCONNECT SIGNALS

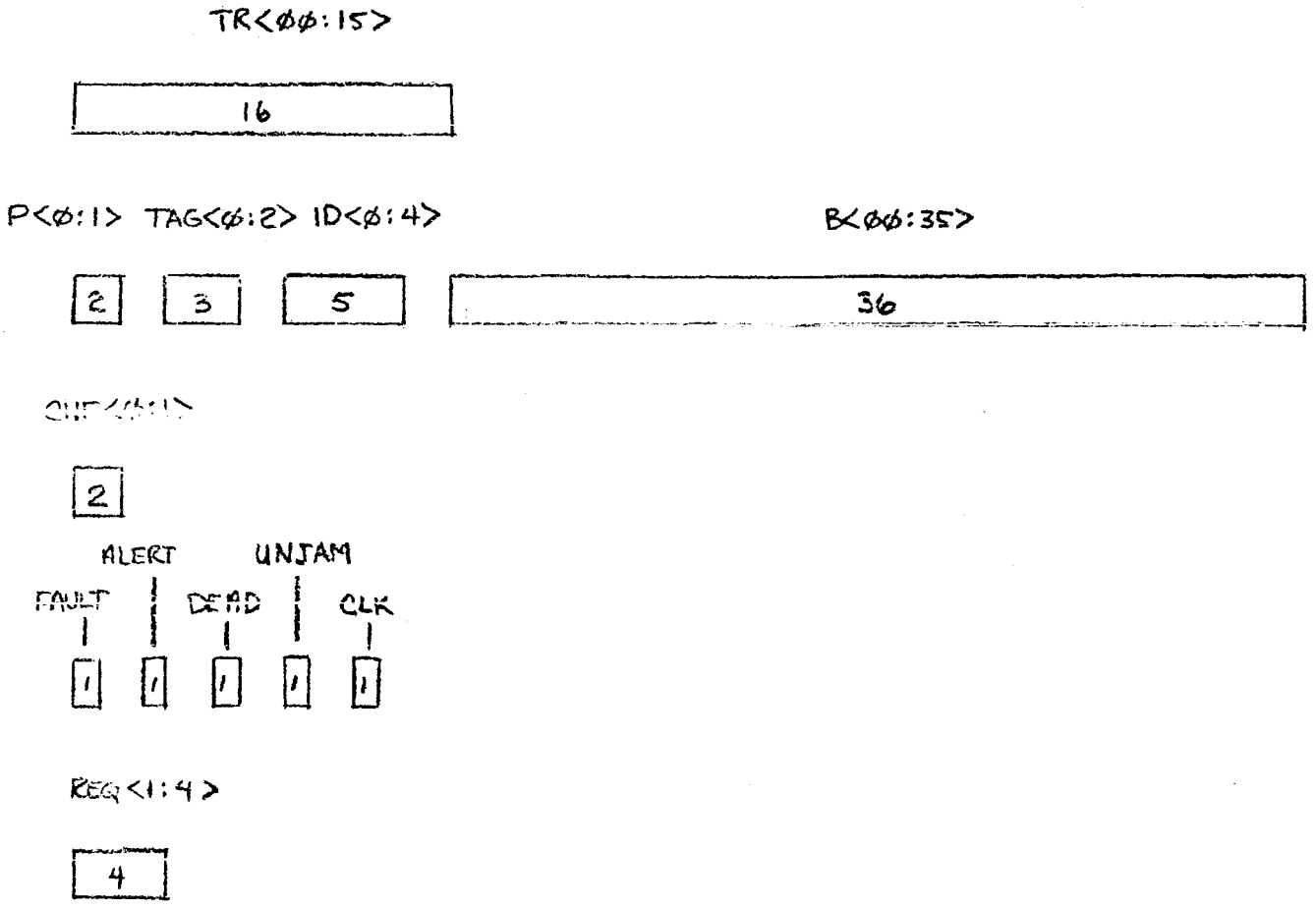
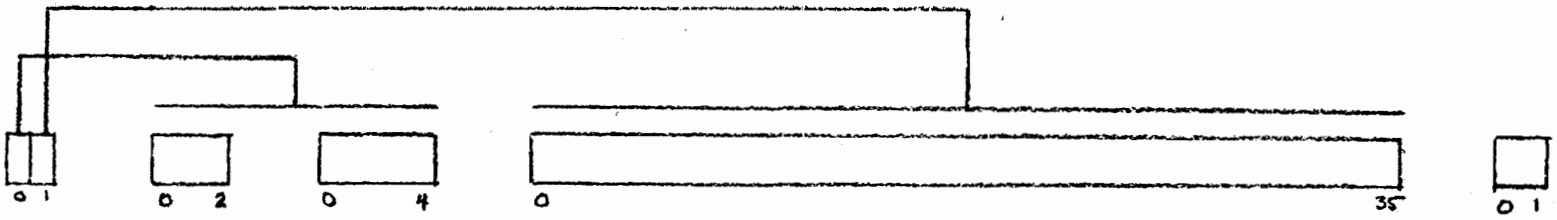


FIGURE 1.

SYNCHRONOUS BACKPLANE INTERCONNECT
CODES AND FORMATS



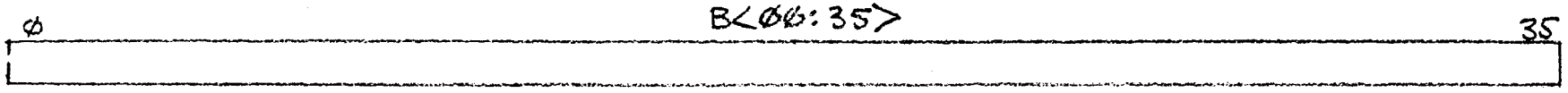
P<Ø:1> TAG<Ø:2> ID<Ø:4> B<ØØ:35> CNF<Ø:1>

EVEN	ØØØ	RCVR	READ DATA SUBSTITUTE
	ØØ1	RCVR	CORRECTED READ DATA
	Ø1Ø	—	RESERVED
	Ø11	RCVR	READ DATA
	1ØØ	TRANS	COMMAND/ADDRESS
	1Ø1	TRANS	WRITE DATA
	11Ø	—	RESERVED
	111	TRANS	INTERRUPT SUMMARY READ

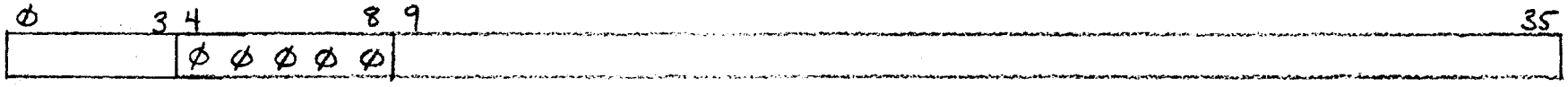
ØØ	NO RESPONSE
Ø1	ACKNOWLEDGE
1Ø	BUSY
11	RESERVED

FIGURE 2.

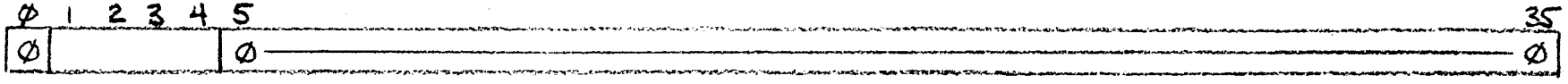
SYNCHRONOUS BACKPLANE INTERCONNECT
INFORMATION FORMATS



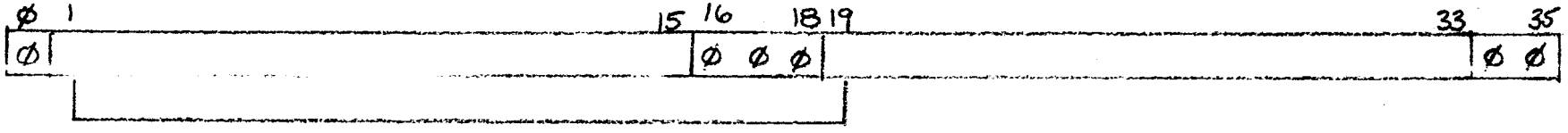
READ DATA, CORRECTED READ DATA, READ DATA SUBSTITUTE, WRITE DATA



COMMAND/ADDRESS



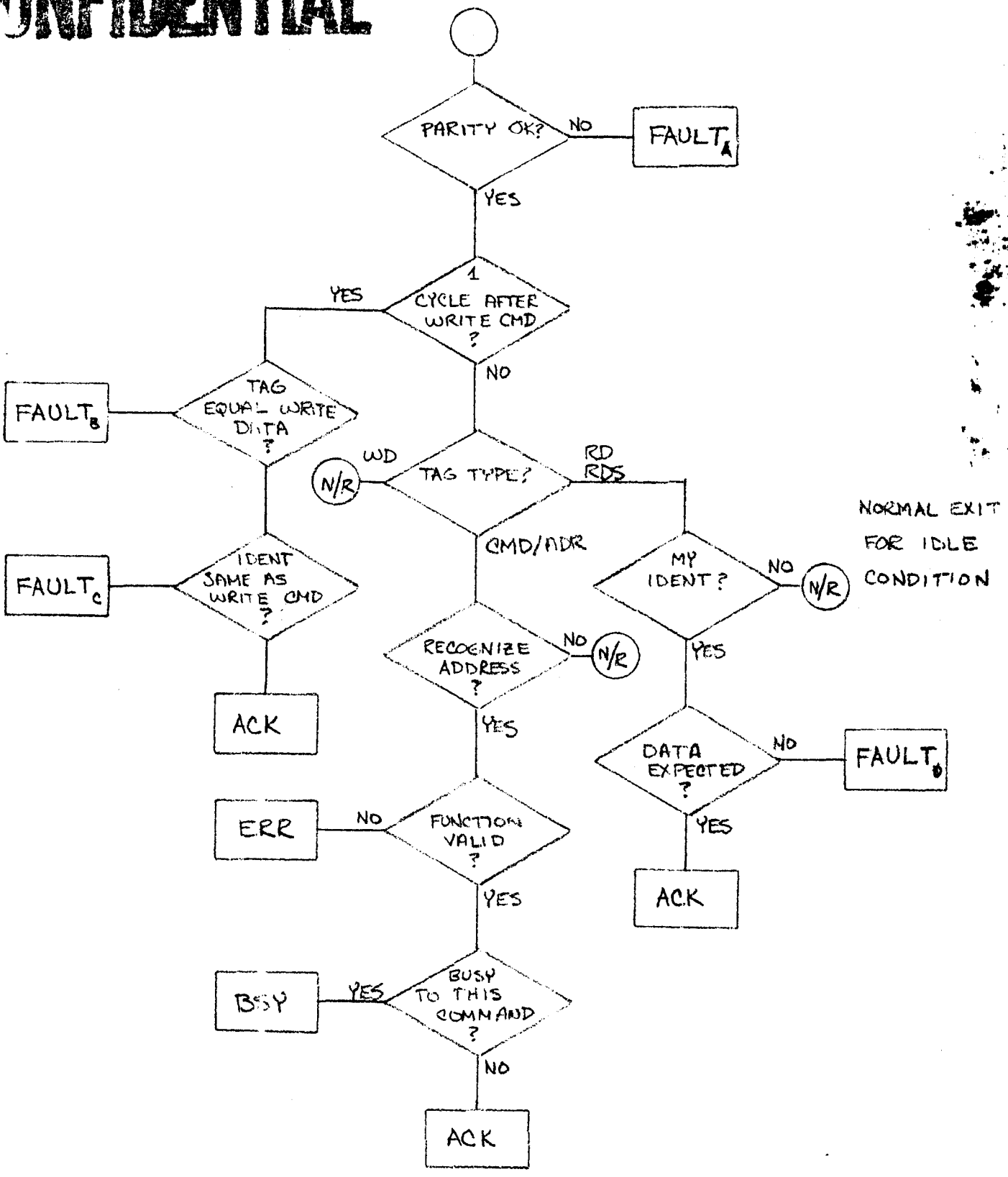
INTERRUPT SUMMARY READ



INTERRUPT SUMMARY RESPONSE

FIGURE 3.

CONFIDENTIAL



SYNCHRONOUS BACKPLANE INTERFACE

RECEIVER DECISION FLOW

FIGURE 4.

SYNCHRONOUS BACKPLANE INTERCONNECT
CLOCK TIMING

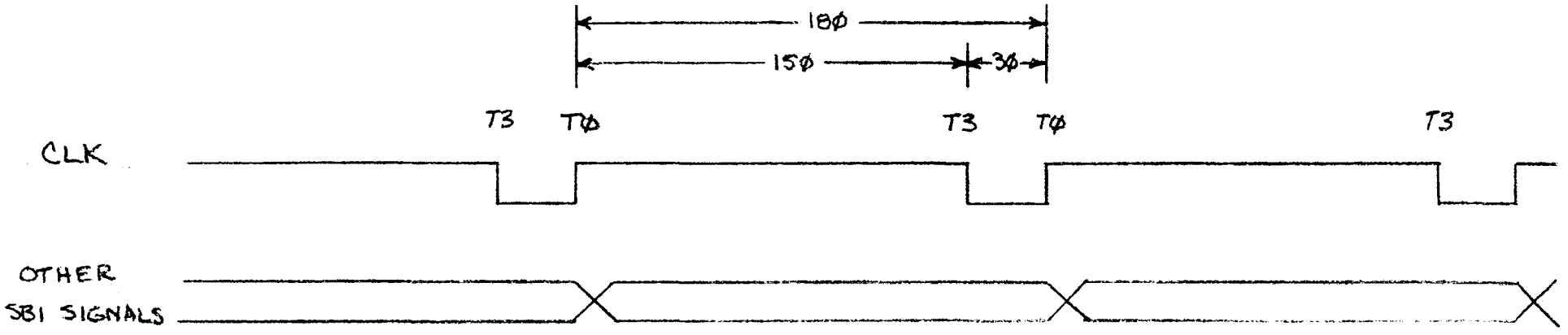


FIGURE 5.

digital INTEROFFICE MEMORANDUM

TO: Steve Teicher

cc: J. Marcus, G. Bell, D. Clayton, L. Portner, J. Bell

DATE: May 30, 1975

FROM: Peter Christy

DEPT: Software Development

EXT: 6110 LOC: ML12/A62

SUBJ: Very High Reliability Computers

You have touched on a subject very dear to my heart. In fact I think that very available systems could be a revolutionary change in our business. As I will expand upon later, a very reliable computer is a vital component of a very available system. My reasoning that VAS's could be important business derives from my experience in the medical world, but is generally applicable. It is fairly simple:

(1) We would like to sell a lot of computers;

(2) At the price range we are used to (centroid = $40K) we could sell a system to essentially every viable economic organization, from the small businessman up;

(3) The untapped market will not buy something that increases expenses;

(4) Therefore, the system must displace labor. My conclusion is that this system serves the function of a general clerk and bookkeeper, which implies the following:

(a) We provide a complete package, like the IBM System/32, which ultimately becomes the total information manager for the customer (e.g., an ideal clerk/bookkeeper).

(b) The system is a general clerical engine for the customer -- in particular, his day-to-day operation centers on the use of the system: orders are received and entered, supplies are inventoried and acquired, billing is done, payroll is computed and managed, etc.

(5) When the customer is dependent on the system then the system must have very high availability. The traditional means of achieving such availability are not sensible in this case, namely on-site system expertise and engineering. The alternative is fail-soft hardware and software.

(6) The technology for such systems is at hand, having been used extensively in diverse applications such as telephone switching (ESS) military command and control, and space control. What has never been done is cost minimization. But in this case it is clear that regardless of the incremental cost for the hardware, the system will at some point in the future be cheap enough ($40K).

The only significant question is whether the incremental cost will be borne in the marketplace for the incremental benefit -- extended MTBF. My personal conclusion is that the information assistant style of application described above, for which I believe that the ultimate markets exist, are impossible without very high availability.

The Hooker

The Hooker is that in a sense you can't get there from here. What I mean to say precisely is that low-cost, high-reliability and high-availability designs seem very much to be ends that are not achievable ex post facto: if you want a system to multiprocess effectively that had better be an initial design goal; if you want an effective fail-soft processor you had better design it that way initially. Maybe that is engineering sensibility, but the hooker part is that the demand for high availability will only come when the potential is obvious, namely when the cost-effective hardware has been built and software provided demonstrating the advantages. The "sensible" system designer will not attempt to build a system that is infeasible with today's effective technology -- to such a designer the incremental cost of fail safe hardware is a burden and not an advantage. The point here is that high availability systems are revolutionary and entail business risk. However, if the reasoning above is correct then the potential is large in terms of market size, and with the risk will come the benefits -- market share and higher profitability.

The Components

I have repeatedly referred to high availability systems rather than high reliability computers. My reasoning is simply that to the vast marketplace for computers a raw computer is essentially useless. When we contemplate the $40K system of 1980 and realize that it will have perhaps 256K bytes of central memory, then this point gets quite dramatic: that system is complex, and yet the customer will necessarily expect to be able to "use" it, without expert knowledge of computers, for a low incremental cost over the purchase price. That implies inescapably that someone has provided very complete software, peripherals, etc. I assume that someone is us, and that the total problem is system design, of which processor design is only one component. When considering high-availability system designs (a real problem in the medical world, where human life is at stake but cost-efficacy is of growing importance), the following components have seemed necessary to me:

(a) A Fault-Free Processor: I think that this is what you were discussing, namely a processor that has the characteristic of either performing correctly with very high probability (compared to say an 11/40) or self-declaring its inability to so perform. In terms of software design, it is infinitely easier to deal with the known inability of one processor to perform than it is to indirectly identify a malfunctioning processor and remedially deal with the damage that it has done.

(b) More Fault-Defensive Mass Storage: Our existing disks (such as the RK05) provide one level of fault detection in that the data is recorded with redundant coding, and the data is formatted with headers that verify the location of the data block (surface, cylinder and sector). An improvement that has been suggested is the addition of a logical key, related to the file in which the data belongs, which would help detect the next layer of errors which result from hardware or (more likely) software faults in the central system. The impact of such design improvements would be a greater bounding on the impact of the inevitable errors, again making it easier to deal with the errors, since in general they would be related logically to the faulting program rather than today's case in which a program may damage the data of a totally innocent program, causing a cancer-like spread of bad data resulting in total system degeneration.

(c) More Modular Bussing: One of the obvious problems with existing Unibus systems is that a small logic failure in a controller can render the entire bus unusable. In a fault tolerant system the bus must be designed such that the critical set of logic is minimal. Although very-reliable Unibus based systems can be built using electronic bus switching and isolation where possible (it is not possible to isolate a MassBus controller like the RH11 from the memory with a bus repeater because of latency problems), and using mechanical bus switching (like TBar switches) where not. But these solutions are inelegant and costly. A much better solution is to design the bussing and packaging from the beginning so that bus modularity and isolation are a low-cost increment and not a major perturbation.

(d) Cost Effective Packaging: In a day when the "box" becomes a more and more evident portion of the cost, the system packaging cannot be ignored. Expecially if bus switching and modular redundancy become a part of the system design then the cleverness of the packaging is likely to make a large difference in build cost. Again, one can build such systems with today's PDP-11's, but the cost saving of totally engineered designs are substantial.

(e) Multiprocessor Software and System Design: Multiprocessing is a classical example of the advantage of designing ahead. Multiprocessing is tricky but hardly impossible. In fact, a good multiprogramming system, which we know how to build, gets about 90% of the way there. But one can fail easily in the nuances, so the best solution by far is to design for multiprocessing if the chance that it will be useful exists, rather than waiting for it to be needed. In addition to software designs, in which interprocess synchronization and concurrent access to data are the critcal questions, there are meaningful system issues also. In particular bus designs, the use of cache memories, and the question of fundamental interprocessor synchronization are very real issues. None of these issues seems to pose insurmountable problems, but each merits very close consideration.

(f) Complete Software Design: One of the hardest things for Digital to do, it seems, is to build equipment for the average guy. When you like programming and logic it's hard to think of building something for someone who HATES programming and logic. The average small business man hires an accountant because he is terrified of accounting and he will buy our system if it is no more costly than the accountant and no more difficult to deal with. But that means that the software has to be complete without any human attention. That is very doable, but a tremendous challenge to those who have been building software for themselves, and for the other bright people of the world instead of building software for the average man.

Conclusion

Yes, I personally have a tremendous interest in the high reliability product!

Availability / Integrity

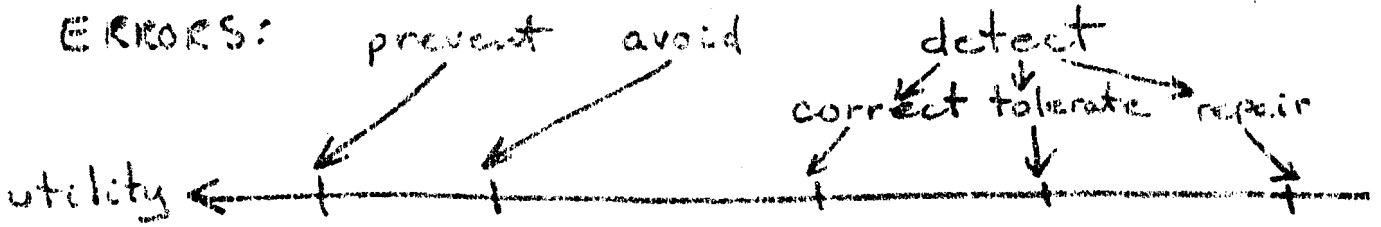
1. Availability built on
   Reliability
   Useability
   Serviceability

A/1 continued

2. Integrity is error detection and reporting or action

3. Utility of system depends on throughput/performance and availability

Effect of errors on performance/throughput/utility:



avoidance by prediction/maintenance

4. Error detection for integrity and availability thru error logging/analysis/diagnosis/repair
   - CHK nn, ss, dd
   - new traps - e.g. instruction time out
     loop detection on iterative

5. Access to error and diagnostic information
   - silos e.g. branch from's
     contexts
   - internal e.g. μproc. regs
     e.g. MDMP dd

MENDELSOHN 29 APRIL 1975

digital

INTEROFFICE MEMORANDUM

		LOC/MAIL STOP
TO:	Systems Architecture List A	
CC:	George Plowman	ML21-4/E20
	Ed Fauvre	PK3-1/E15
	Brian Croxon	ML1-5/P53
	R. Morris	ML1-5
	Ralph Platz	ML5-5/E67

DATE: April 22, 1975
FROM: Jim Bell
DEPT: R & D Group
EXT: 2764
LOC/MAIL STOP: ML3-4/E41

APR 29 1975

SUBJ:

At a recent meeting with Bolt, Baranek, and Newman on the topic of high reliability systems, they recommended that we consider computing memory parity differently on future machines. They claim tnat doing so has been very helpful to the reliability of their high speed IMP (build from Lockheed SUE's).

They suggest computing the parity of a memory location as the modulus 2 sum of all the address bits and data bits (as opposed to our current use of data bits only). The advantage is that address as well as data errors can be caught, and that more cases of bits stuck zero or stuck one are detected. Changes to error handling in operating systems and diagnostics were said by BBN to be minor.

JRB/bd

Goal./ Constraint \ Machine	11-Extended (true superset)	8-bit byte 11	EIS-only formats	Cultural 11	Microprog. Machine	
		← strict 11 is a mode →				
C1-Extend VA ≥ 24-bits	1	1	1	1	?	
C 8-no apparent train.	—	—	—	—	—	—
I 8.1 Same registers, formats	1	2	3	4	no.	
8.2 Same prog. style	1	1	2	3	no.	
8.3 Same assemblers	(1)	2	2	2	no.	
8.4 Op. Code elegance	[4]	(1)	2	3	?	
q0.1 Min. space (Need Benchmarks)	[ ]	(1?)	3	3		
0.2 Min processor time	[ ]					
0.3 Min. Cost	(1)	3	3	3		
0.4 Generality	[4]	(1)	2	3		
0.5 Min. programming	?	(1)	3	4		
0.6 Max code sharing	incompl.	Not [illegible]	incompl.	→		
q3 Wide range (1000:1)	—	—	—	—	—	—
I 3.4 Spare to expand	1	1	1	1		
q6 Enhance ISP	—	—	—	—		
6.0 More Gen funct	1	1 (worked out)	1	1		
6.1 Strings, etc.	2		1	1		
q7 RAS						
7.3 More Call Protect.	(incompl.)	Not [illegible]	(incompl.) →		[illegible]	
q9 16-bit user mode op.sys.						
9.1 16-bit compat.	Subset	← Precludes by [illegible] →				
9.3 Write 11 and 11/VAX progs	(1)	2	3	4		
q13 Min. re-programming	(1)	2	3	4		
q Min. phaseover						
I 5.3 Coding Conventions	(1)	2	3	4		

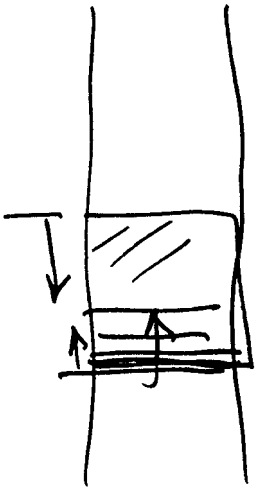
gB
May 26, 1975

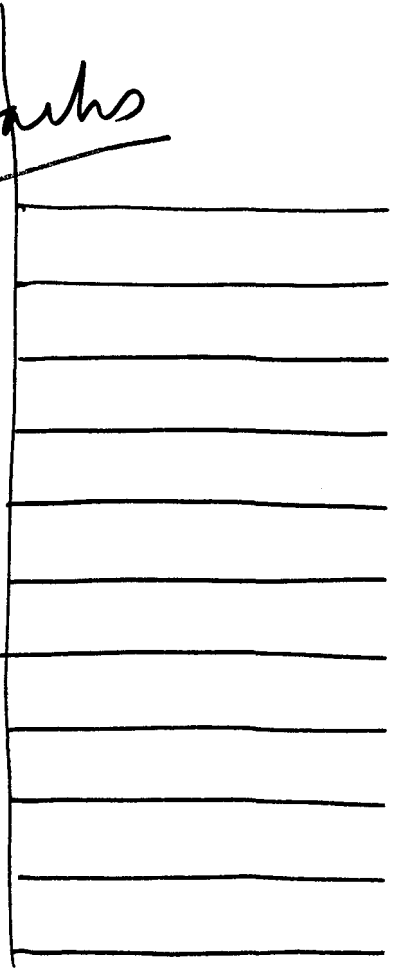
[Languages]

Marty Jacks

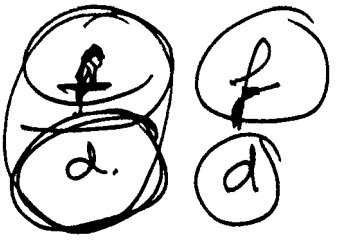
Select. data-types. < strgs. / float-pt

address depth.





- byte
- i
-



f - unpacked

f

extended ops

-b . fr / fm

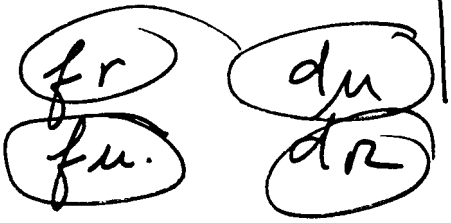
-i / a. . dr / m

- d

- Q

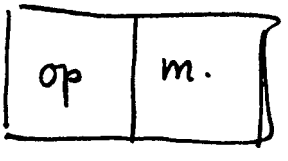
- a

i



- string.
- dec. floatg.

+
−
*
/
mod.



<-0 - 11
+1
-1
+ comp.

- ~~addr~~ i
- addresses
-

Future → ops on existy. (in effect — there are f(used × speed tradeoff).

→ data-types. | Fields ; bits

strings ::= (compare, move).

Vectors ::=

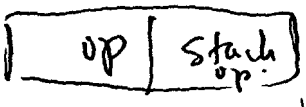
[dec. float | long decimal ..] [un-norm] (doub dec) decimal floaty. ( )

decimal ( )

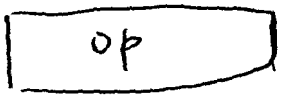
lists (move)

bi op.

s/d.



start. bi/unary



ops.

~~float~~
~~float~~

digital

INTEROFFICE MEMORANDUM

TO: LOC/MAIL STOP

DATE:
FROM: Jim Bell
DEPT: R & D Group
EXT: 2764
LOC/MAIL STOP: ML3-4/E41

SUBJ: ARRAY BOUNDS CHECKING IN MICRO-CODE

This memo represents a quick look at the costs of different styles of hardware checking of array bounds violations.

Let us consider a vector V, with lower bound VLOW and upper bound VHI. Then a (non-exhaustive) set of alternatives for insuring that the array element V(REG) is in bounds is as follows:

A. Software Method

```
CMP     REG,VLOW
BLT     error
CMP     REG,VHI
BGT     error
```

Here the space cost is 6 words in line, and the execution cost is 8 memory references. Note that in the common case where VLOW is zero and VHI is a known constant, both tests can be combined into

```
CMP     REG,#CONSTANT
BHI     error
```

or alternatively into

```
CMP     REG,#CONSTANT
ADC     R7
```

which should be even faster (though I have not actually tried it). Here the space cost is 3 words and the time cost 3 memory references.

B. New instruction for bounds checking

We can imagine a micro-coded bounds-check instruction BCHK. The argument PTR would point to the pair of bounds

```
BCHK    REG,PTR
```

B. New instruction for bounds checking (Cont'd.)

and the instruction would trap on a bounds violation. Here the space cost is 2 words in-line and 2 out-of-line (shared); the execution cost is 4 memory references.

For the common case mentioned previously the variant

```
BCHK    REG,#CONSTANT
```

could be provided. This would use 2 words and take 2 memory references to execute.

C. Descriptor addressing

A more drastic change would be to allow descriptor based addressing. We might assume no new in-line code is needed and that a vector element is referenced indirectly through a descriptor block containing the vector's address and bounds. Then the space cost is no words in-line, 3 words out-of-line, and the execution cost is 3 memory references. However, this alternative undoubtedly has very complex ramifications.

D. Other considerations

All of the above analysis should be taken with caution. In real life, the lower bounds are almost always zero or one, and the upper bound is normally a constant known at compile time.

Furthermore, a smart compiler can often move a bounds test outside of loops so that its time cost is essentially zero.

Finally, the above calculations must be modified to reflect 32 bit addresses, subscripts, and bounds.

JB/rlf

5.) Extended Instructions (WCS). Since many of the simulator algorithms involve positioning of the aircraft and other objects relative to the aircraft, there is a heavy use of trigonometric functions. It would be nice to provide some way for the simulator vendor to extend the standard instruction set to include trigonometric functions. Writeable control store was mentioned as a possibility in this area.

6.) Multiprocessors. The larger simulators today make extensive use of multiprocessors. Multiport memory was seen as the most cost effective way to tie multiple machines together. Other methods of implementing a multiprocessor system were not specifically mentioned, however, from the total conversation, I believe that the requirement is primarily one that allows easy transfer of data between machines while maintaining a simple intermachine protocol and protection scheme.

From the conversation with Phil Babel, my own personal assessment of this situation is that the data accuracy requirement could be solved by partitioning the program. At any point in time, the aircraft is only operational over an altitide and speed range for which 16 bit of resolution is more than sufficient. The functional requirement to distinguish velocity and height over a specified range is a real functional requirement of an aircraft simulator. This could easily be implemented with essentially double precision techniques where only one half of the data words need be accessed for calculations since the aircraft is operating in either one range or the other during most of the normal flight training. This approach will take some selling towards Babel. I believe the simulator manufacturers are already convinced of this point and are only hung up because of Babel's spec requirements for a longer word length.

The request for bit manipulation instructions I believe is real and should be seriously considered on a larger word length machine that we would develop. Bit manipulation instructions would allow the users to pack data more efficiently into a larger word length machine and be able to access it quicker than with the currently used masking technique. In the area of their request for additional instruction, I believe this capability would be satisfied by a writeable control store or, as an alternative, a fast high speed memory. In either case, the requirements of our general customer base will probably dictate this solution and we should not design something special for just the simulator market.

The requirements for multiprocessors is relatively primitive and if the current trend continues within our development effort, then I believe that the standard product will contain sufficient multi-processor hooks for the simulator market.

JFB:lmb

CC: Bill Demmer, ML5-5/E67

digital INTEROFFICE MEMORANDUM

TO: Larry Wade
VAX A and B distribution
cc: R. Blair, M. Jack
J. Bell
R. Grove
F. Infante

DATE: 21 May 1975 28 May

FROM: George Poonen

DEPT:

EXT: LOC:

SUBJ: PDP-11/32 Instruction Set

Roger Blair, Rick Grove, Frank Infante, Marty Jack and I have reviewed the "Byte Oriented ISP Proposal" by Bill Strecker. The questions we addressed were

a. Difficulty of generating code for this machine for FORTRAN, COBOL, PL/I like languages

b. Difficulty of transferring existing compilers viz. FORTRAN IV PLUS, COBOL to the new machine

c. Any additional facilities that might be useful for languages

Detailed comments from some of the above are attached as an appendix.

Overall it was felt that the uniformity of the architecture would enable easier code generation as well as give better code. (No attempt at evaluating the bit efficiency, etc., of the machine was made. This would be somewhat unrealistic to do. The evaluation was much more subjective in terms of overall facilities.)

It was felt that no serious problems would be encountered transferring either the FORTRAN IV PLUS compiler or the COBOL interpreter onto the 11/32. An estimate of the effort required to make the transfer is being made by Ron Ham separately (cf. conversation with R. Grove and F. Infante).

The majority of comments received were directed towards seeking clarification of existing facilities or suggesting additional ones. These are summarized below. The issues are discussed in greater detail in the appendix. The starred paragraphs are of high priority.

*Procedure Call & Entry

It is crucial that the call, parameter passing, and entry mechanism be completely defined and tried out for FORTRAN, PL/I, COBOL, BLISS, PASCAL.

*Decimal Arithmetic

Critical for COBOL and PL/I. The current proposal is sufficient although one would like to have full decimal arithmetic capability. The trade off in providing this must be investigated.

*Editing

Once again a crucial feature for COBOL. If the appropriate instructions are not available this becomes a fairly time consuming task. The EDIT instruction of the S/360 or Honeywell NPL machine appear to be good starting points.

Block Moves/Comparisons

Some form of block operations would be useful for both PL/I and COBOL. A Block move is a minimum. Other facilities such as comparison could be added if space was available.

Heap Instructions <Important for PASCAL-like Languages>

Most recent languages have a heap associated with them. Heaps are generally implemented as growing towards the stack. Since this facility is used heavily in languages that allow it, a simple instruction to allocate a specified amount of heap storage and also check for overflow against the top of the stack would be useful.

*Field Operations

In a number of applications packing of data enables great savings of space. Some languages such as PL/I give one precise control over the arrangement of data. Minimum facilities to extract and store fields within a word must be provided.

*ON Conditions

The interrupt structure of the 11/32 is not specified. In PL/I the ON condition is a very useful facility and its implementation on the new machine must be considered.

Other Useful Instructions

A number of the following suggestions are based on what is currently expensive to do on an -11 (while being fairly commonly used in some recent languages such as PASCAL)

(*) SET Operations

a. Creation of a Singleton Set.

CSS	A, B	where A is an ordinal number (<63) and B gets the appropriate bit set.

b. Membership

IN	A, B	returns a Boolean if ordinal A is in set B.

Other set operations can be conveniently done using existing operations.

Relational Operators (EQU,GEQ, etc.)

In addition to the current method it may be useful to provide relational operators that return a Boolean result. (May require 3 operands) In addition we need a TRUE and FALSE JUMP.

Save Register m to n.

Translate and Test.

Case Jump Statement

This could be done as an indirect jump. However, a more elaborate form is both useful and commonly required.

Case JMP, LBound, UBound, Table, Expr, No action Label

The above takes the no action label if the expr is not within the bound; otherwise, it does an indirect jump through the TABLE.

Load and Store Bit Strings

Useful in PL/I and COBOL.

AND Instruction (Low Priority)

Additional Instructions for Debugging

a. A window keeping track of the last n values of PC when jumps took place

b. A representation for undefined variables which may optionally cause a trap

c. The ability to specify traps for any and all op codes. (This may be expensive to implement.)

32 bit Integer in FORTRAN IV PLUS

This appears to be a problem. The problem is discussed in greater detail in the appendix.

APPENDIX I

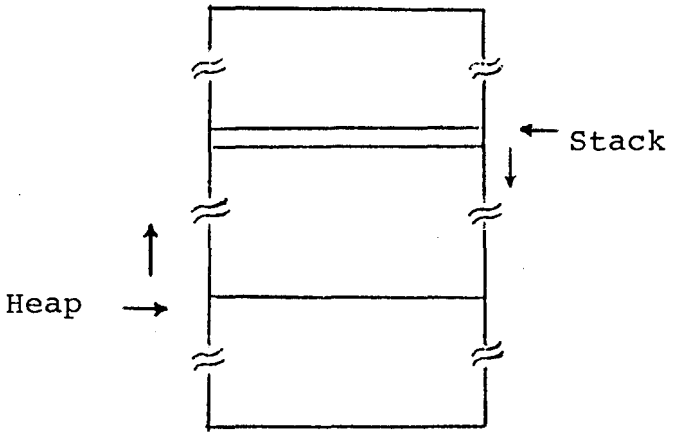
This section contains further detailed discussions of some of the requirements mentioned earlier.

Also attached are individual comments.

Further Notes

Heap:

In a number of recent languages there is a concept of a HEAP; e.g., ALGOL68, PASCAL, PL/I, etc. Typically a heap is implemented as follows



The stack grows in one direction while the heap is allocated in the other direction. In compilers written in such languages this is a frequently occurring operation; e.g., in the PASCAL compiler the dynamic allocation routine is statically the second most frequently occurring routine. No dynamic frequencies are available; however we do know that the PASCAL heap grows to about 6K words while compiling itself (in 29K). Since considerable heap storage is released after each procedure is compiled we could roughly assume about 12K of dynamic allocation was done. If on the average about 4 words are allocated each time we get $\approx$3000 calls to this routine. Finally one of the indirect effects of having a heap, as designed, is that it makes the procedure prologue more expensive. A check has to be made for heap overflow. Anyway the point of all this is

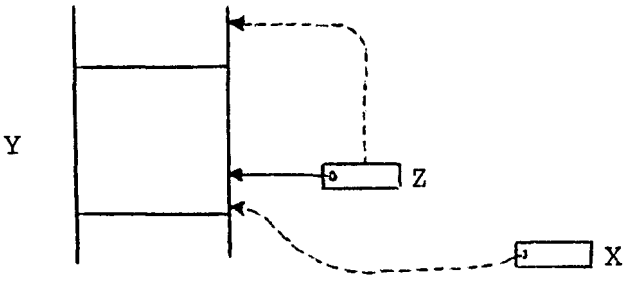
a. dynamic allocation is useful and frequently used (not only in compilers) in PL/I like languages

b. considerable savings in time and space can be achieved by appropriate instructions

Considering the above, I believe a strong case could be made for the following instruction (or some variant of it).

ALLOC X, Y, Z

Z is the heap top.



ALLOC does the following

(X) ← (Z)
(Z) ← (Z)+(Y)

[If (Z) ≥ Stack ptr. then Error (heap overflow)]

The stack pointer should signal an error if it grows into the heap.

The above could be simplified by having an implicit heap top register or location.

On the other hand it could also be generalized to a general form of auto increment and decrement instructions for a heap.

Procedure Calls, Prologue, Epilogue

Compiler implementors worry a lot about the procedure call, prologue and epilogue mechanism. If there is one thing that would greatly encourage small modular programs, it is the provision of a highly efficient procedure call and return mechanism.

The memo describing the machine does not describe the CALL mechanism so I will just mention some possibilities (see also Roger Blair's comments, attached).

The simplest approach as provided by the -ll currently is clearly the most flexible. It leaves the implementor to set up his current activation record as required.

In an ALGOL-like or PL-1 environment some of the activities that are performed are shown below (clearly there are different ways to implement the same)

1. Reservation of space in case it is a function
2. Pushing of parameters on stack (alternative mechanisms are also available)
3. Calculating Static Link - this depends on the difference between static level of calling block and declaration block
4. Transfer of control
5. Saving registers if any
6. Setting up Dynamic Link
7. Assigning value to current frame
8. Enabling any CONDITIONS (as in PL/I)

The return sequence would include

1. Restoring previous frame
2. Restoring stack pointer
3. Returning control.

Some of the above could be established using a special create activation record and release activation record instructions.

FIELD EXTRACTION

Most current languages allow one to define data up to the bit level.

On the -11 there is a distinct trade off between the space saved by packing bits and the difficulty in extracting the field.

A possible way of incorporating this in a general form is to have another mode as indicated below; if this turns out to be expensive or unfeasible then the byte pointer mechanism on the -10 will probably suffice.

If an extra mode - (packed) is provided, then the associated address could indicate the actual address, position and size; i.e.,

```
        OP (PACKED) ADR1

ADR1  :  ADDROP2        (Possibly using existing mode &
         POS , SIZE     address)

           8 bits 8 bits  (Only 6 are probably required)
```

SET Operations

The Set data type and associated operations as defined in PASCAL and some of the more recent languages are extremely useful in a variety of applications including compilers.

Associated with a SET are

1. The definition of members of a set

   Usually represented as ordinal numbers; e.g.,

   SET1 = (A, B, C, D)

   where A=0, B=1, C=3, D=4.

2. The definition of Power sets

   e.g., [A, C, D]

   usually represented by setting the associated bits; i.e., Bit 0, 3 & 4 in this case.

3. The operations associated with SETs are

   UNION, INTERSECTION, DIFFERENCE, CREATION OF SINGLETON SET, MEMBERSHIP.

   The first three can be done easily with existing -11 operations except for INTERSECTION which could use an AND instruction.

The following two need to be provided.

Creation of Singleton Set. CSS

CSS A, B ≡ convert the ordinal number A to a bit position
move result to B

The size of the SET could be limited to 64. This covers the majority of cases and is the minimum required in the standard for PASCAL.

Membership ≡ IN

IN A, B,(C) A is an ordinal number

B is a powerset represented by bits

The result is a Boolean.

E.g., if A=5 and the 5^{th} bit is set in B then the result is true.

The result may require another operand.

Relational Operators

It would be nice to have in addition to the -11 set, the operators EQU, NEQ, GEQ, etc. which return a Boolean result together with a False and TRUE jump. Currently if relational expressions are used with a Boolean expression we need to do the following; e.g.,

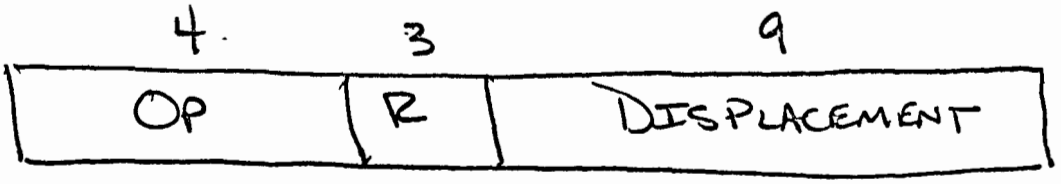
```
        CLR  R             Δ       EQU   A, B, Result
        CMP  (SP)+,(SP)    ≡
        BNE  L
        INC  R
    L:  MOV  R,(SP)
```

Clearly if an explicit Boolean result is not required we can use condition codes.

HALF WORDS, [redacted] ARE ALWAYS SIGN EXTENDED

7.- SUGGESTED OP-CODE INSTRUCTION FORMATS

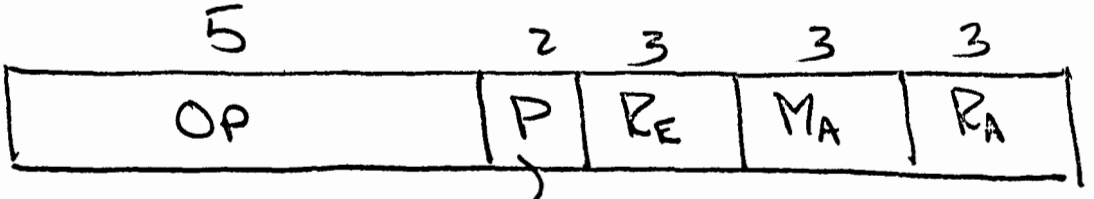
CLASS-1 LOOPING INSTRUCTIONS



SUBTRACT ONE AND BRANCH IF "GT"

NOTE: ONLY ONE INSTRUCTION IN THIS CLASS

CLASS-2 BINARY OPERATIONS



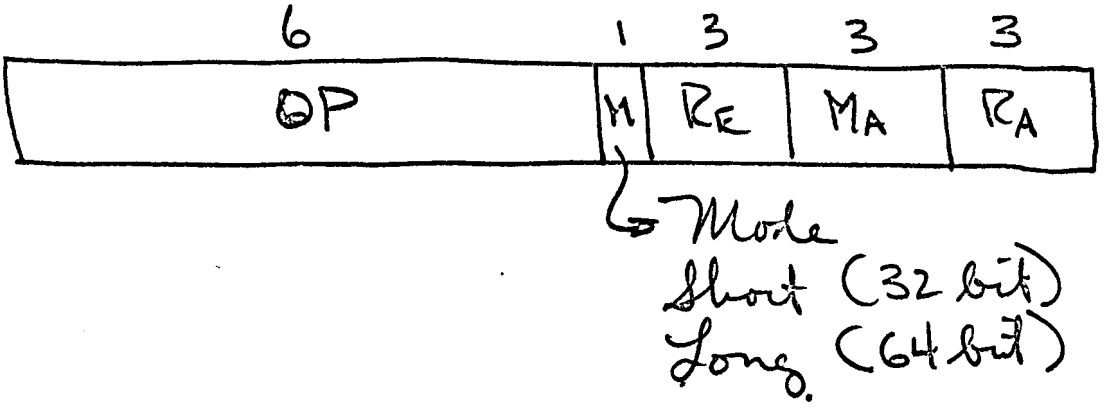
→ Partial word designator
Byte
Halfword
word
A.-Al. word

OPERATIONS
ON DOUBLE WORD [redacted] TO REGISTERS RE AND RE+1 ARE AFFECTED.

THIS INSTRUCTION CLASS CONTAINS ALL THE [redacted] BINARY ARITHMETIC OP's. e.g.

ADD	MUL	ASH	BIS	BIT	XOR	STORE
SUB	DIV	ASHC	BIC	CMP	LOAD	

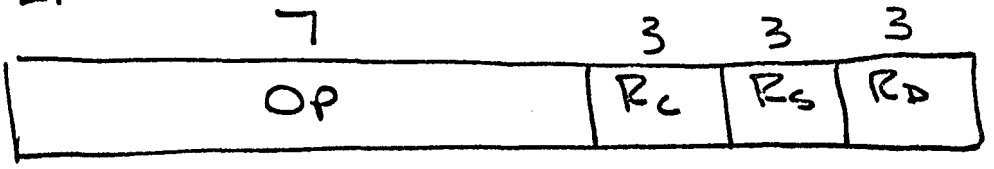
CLASS-3 FLOATING POINT OPERATIONS



NOTE: THERE SHOULD BE A MEANS TO MOVE BETWEEN FLOATING AND GENERAL REGISTERS. e.g. WORKING WITH EXPONENTS (SEE R. BRENDER & RICH GROVE)

CLASS-4 BLOCK TRANSFER + SUBROUTINE JMP + BRANCH

BLT



R_C= Byte count register
R_S= Source address register
R_D= Destination address register

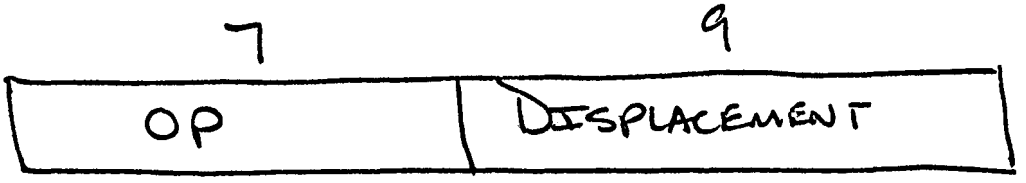
ALWAYS MOVES A "BYTE" STRING. HARDWARE CAN BE SMART AND OPTIMIZE MOVE IF DESIRED.

INSTRUCTION IS <u>ALWAYS</u> INTERRUPTABLE AFTER EACH MOVE BY SIMPLY BACKING UP THE PC AND RECOGNIZING THE INTERRUPT

JSR

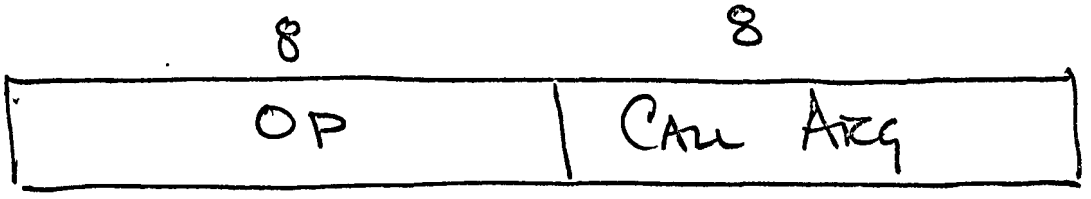
R_L= SUBROUTINE LINKAGE REGISTER

BRANCHING INSTRUCTIONS



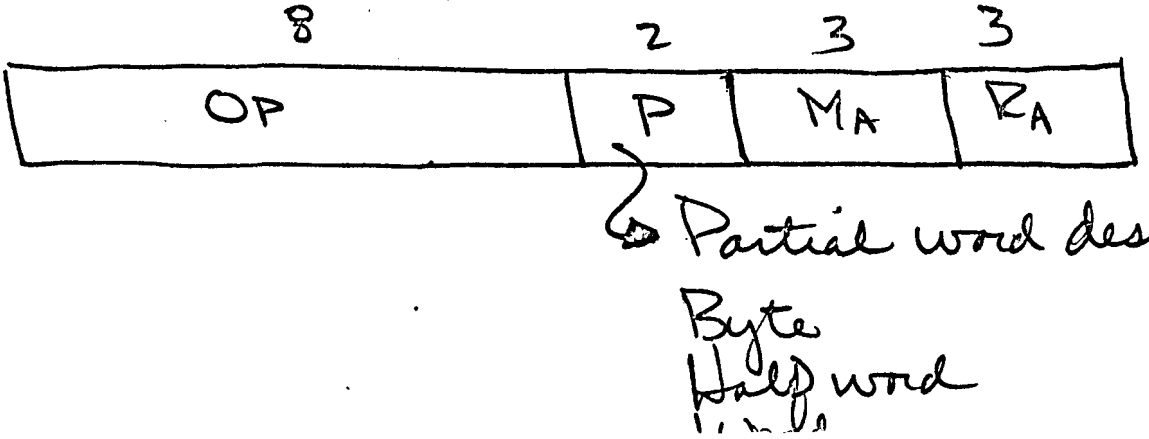
CLASS - 5 TRAP INSTRUCTIONS + UNARY OPS

EMT + TRAP + OPSYS CALL



NEW OPCODE "OPSYS CALL" IS ADDED SO THAT ALL OLD ENVIRONMENTS CAN BE EMULATED IN SOFTWARE.

UNARY OP's



UNARY OP'S INCLUDE

CLR	INC	NEG	ROR	ASR	SWAB
COM	DEC	TST	ROL	ASL	TSTS

NOTE: IT MAY NOT BE NECESSARY TO INCLUDE

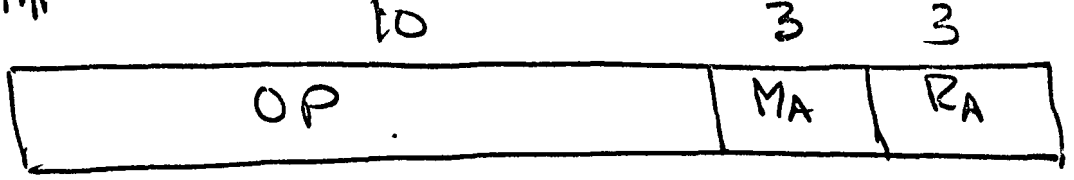
ROR ASR
ROL ASL

TSTS= TEST AND SET (CLEAR?)

THE DESTINATION OPERAND IS READ AND EITHER SET OR CLEARED. THE VALUE READ IS USED TO SET CONDITION CODES. PROBABLY FASTER TO MAKE THE INSTRUCTION A TEST AND CLEAR.

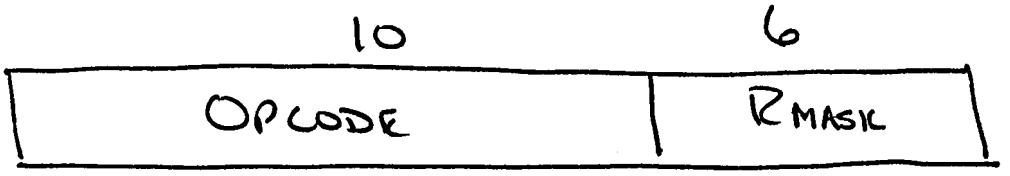
CLASS-6 MISC OTHER'S

JMP



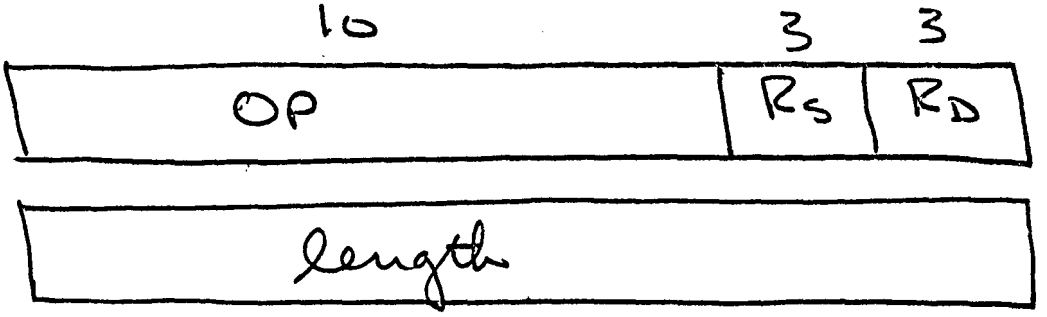
ALL OTHERS NOT DEFINED.

REGISTER SAVE / RESTORE



RMASK= REGISTER MASK - A ONE BIT IS SPECIFIED FOR EACH REGISTER TO BE SAVED/RESTORED. IMPLIED OPERAND IS STACK

Decimal Arithetic Should Be Considered in ISP. Might be of the form



Where:

R_S = Source Address Register

R_D = Destination Address Register

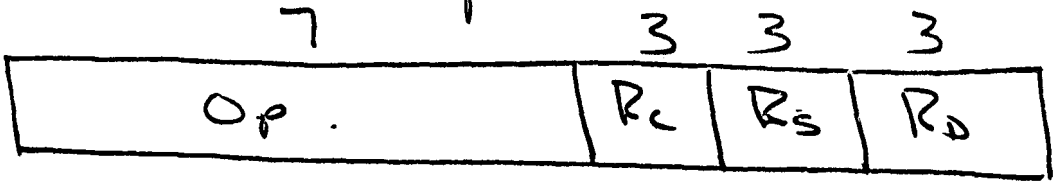
length = length in Bytes of the operation.

This format has two disadvantages

1- It is not interruptable

2- length is fixed.

Probably a better format is



D.N. CUTLER 4-APR-15

ISP NOTES:

1- MOVE TO AND FROM USER SHOULD SET CONDITION CODES ON FAULTING CONDITIONS.

2- ADDRESSING AS TO STRECKER PROPOSAL IS OKAY.

3- TEST & SET IS ONLY OP-CODE THAT SHOULD DO DATAIP UNLESS OTHER WAY TO SYNCHRONIZE ACCESS

4- DOUBLE GENERAL ADDRESS INSTRUCTIONS SHOULD BE ABOLISHED IN FAVOR OF INCREASING OPCODE SPACE.

5- ALL BINARY & UNARY OP'S SHOULD BE CAPABLE OF MANIPULATING BYTES, HALF WORDS, WORDS, AND POSSIBLE DOUBLE WORDS

6- BYTES ARE NEVER SIGN EXTENDED WHEN

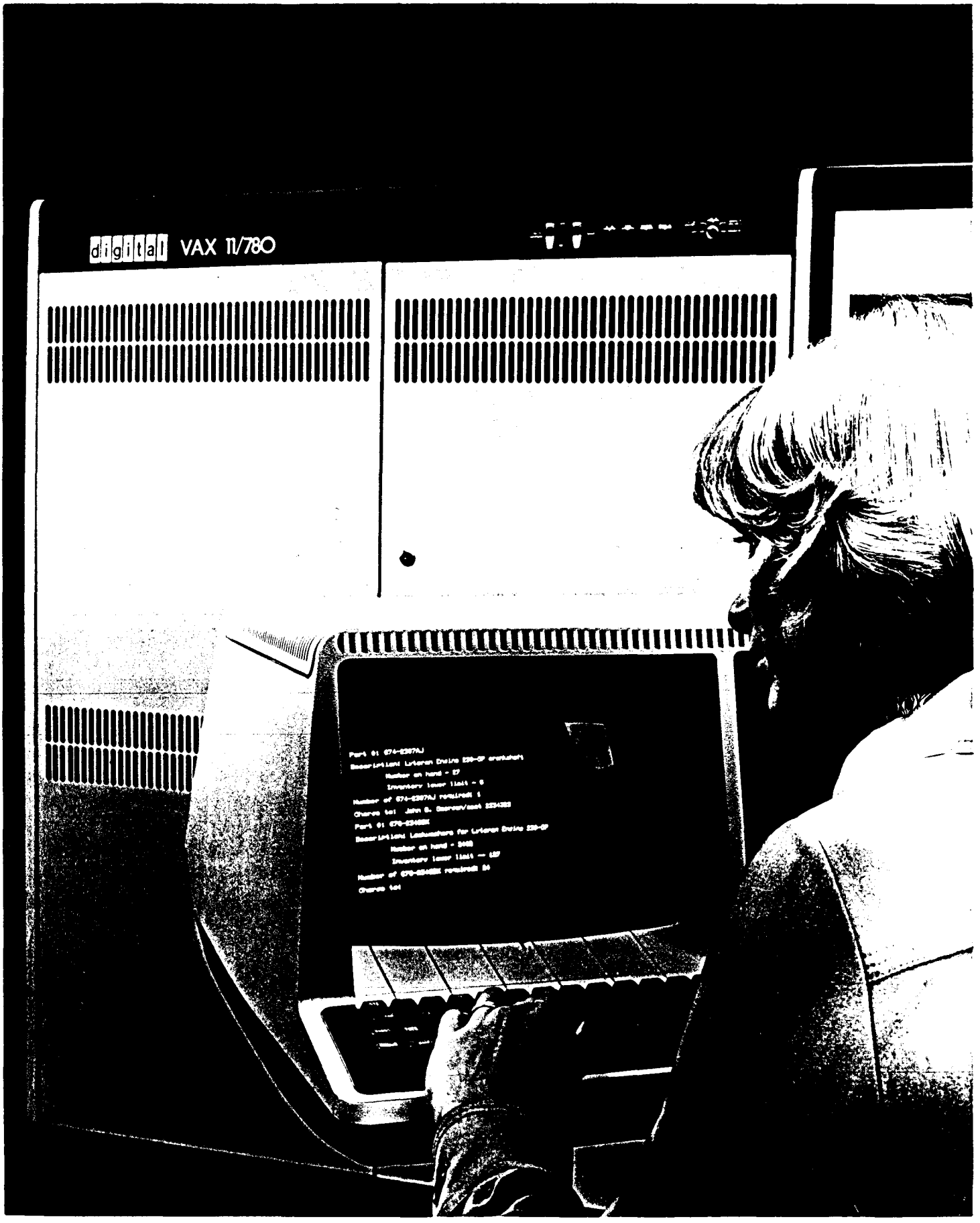

digital VAX 11/780

digital EQUIPMENT CORPORATION

Digital's new 32-bit computer features 4 billion byte virtual addressing, a virtual memory operating system, and extended compilers. Called VAX-11/780(TM), it is a high-performance, multi-user, multi-program system specifically designed for interactive applications. As the fastest-operating 32-bit computer system priced below $200,000, VAX-11/780 is suitable for scientific, commercial, control, and simulation applications. System prices start at $130,000 with deliveries scheduled to begin in early 1978.

######

For Further Information:
Stephen A. Kallis, Jr.
(617) 897-5111 Ext. 2777


digital VAX 11/780

BOSTON, Mass. -- October 25, 1977 -- "Probably the most significant interactive computer of the decade" was the description given by Digital Equipment Corporation President Kenneth H. Olsen of his firm's new VAX-11/780(TM) computer unveiled today at a press conference prior to the company's annual meeting of shareholders in Boston. The company says the new 32-bit computer system combines the full power and performance of conventional large computers with the interactive strength, flexibility and low cost of a minicomputer. The multi-user system is said to be the industry's fastest computer system priced under $200,000. Anticipated early markets for the computer are original equipment manufacturers (OEMs) and end-users in industrial, commercial, research and academic environments.

######

For Further Information:
Edward J. Canty
(617) 897-5111 Ext. 2268


digital VAX 11/780


VAX 11/780

Digital Equipment's new VAX-11/780(TM) 32-bit computer system supports up to 64 interactive users as well as multi-stream batch processing. The virtual memory operating system permits simultaneous processing of multiple large programs for scientific computation, simulation, commercial and control applications.

######

For Further Information:
McLaren Harris
(617) 897-5111 Ext. 2857


digital VAX 11/780

Digital Equipment's new VAX-11/780(TM) 32-bit computer system supports up to 64 interactive users as well as multi-stream batch processing. The virtual memory operating system permits simultaneous processing of multiple large programs for scientific computation, simulation, commercial and control applications.

######

For Further Information:
McLaren Harris
(617) 897-5111 Ext. 2857

digital EQUIPMENT CORPORATION

Interactivity, reliability and maintainability are emphasized in the design of Digital's new VAX-11/780(TM) computer system. The high-speed, multi-program system uses the 32-bit VAX-11 FORTRAN IV-Plus language as well as existing PDP-11 software including BASIC-Plus-2 and COBOL.

######

For Further Information:
McLaren Harris
(617) 897-5111 Ext. 2857

VAX11 780

NEWS RELEASE

digital

For Further Information:
Richard O. Berube
(617) 897-5111 Ext. 3046

DIGITAL EQUIPMENT CORPORATION UNVEILS ITS

FIRST 32-BIT COMPUTER SYSTEM: VAX-11/780

Called by Ken Olsen "probably the most significant interactive computer of the last decade...a milestone equal to the original PDP-11..."

BOSTON, Mass.--October 25, 1977--Digital Equipment Corporation introduced a new 32-bit computer system today which combines the full power and performance of conventional large mainframes with the interactive strength, flexibility and low cost of a minicomputer.

Called the interactive VAX-11/780, the new multi-user system is an extension of Digital's PDP-11 family and is said to be the industry's fastest computer system priced below $200,000. It features a new virtual memory operating system (VAX/VMS) which provides multi-users a direct addressing capability of over four billion bytes!

Prices for the VAX-11/780 system start at $130,000 for a minimum configuration. Deliveries are scheduled to begin in early 1978.

-more-

In announcing the new system to 400 shareholders and 100 members of the press at the company's Annual Meeting here, Digital President Kenneth H. Olsen called it "probably the most significant interactive computer of the last decade. Indeed, we think it is a milestone equal to the original PDP-11 in terms of the long-range impact it will have on the way people use computers."

In addition to its extensive interactive timesharing capabilities -- 64 separate users can work with the system simultaneously -- the VAX-11/780 also has impressive multi-user batch processing capabilities, according to Olsen. "We designed the 11/780 to be a general purpose machine which can handle a wide variety of applications in science, industry, education, and in the commercial world. And we think it will be popular with our OEMs, as well," he said.

"The best of what we've learned about interactive computers in our first twenty years has gone into this machine," he continued. "We have spent over 300 man years of intensive engineering effort in its development, and during that time I have sensed more excitement and enthusiasm among the developers of VAX than I remember seeing at any other time in Digital's short history."

Olsen described the VAX-11/780 system as an upward extension of Digital's popular PDP-11 family. "And yet, it is an entirely new and exciting computer in its own right. We have gone to great lengths to design a system that will be useful to the largest possible number of users, starting with those who use the 50,000 PDP-11s we have already installed worldwide and the thousands more we expect to sell in the future.

-more-

"Our PDP-11 customers have invested enormous resources in the form of trained personnel, peripheral devices and data bases, all built up to support PDP-11 systems. To the extent that we found it possible, we wanted to be certain that these resources would be compatible with our new VAX system," he said. "As a result, the 11/780, with its 32-bit wordlength and comprehensive operating system software, provides a natural upward migration for PDP-11 users whose applications require additional address space and functionality.

Olsen noted that Digital's PDP-11 family has the widest scope of any family of computers available today. "Starting with our LSI-11 microcomputer at the low end, the PDP-11 family forms a continuous spectrum of full system functionality which now ranges up to the new VAX-11/780," he said.

######

NEWS RELEAS

For Further Information:
Richard O. Berube
(617) 897-5111 Ext. 3046

DIGITAL'S NEW VAX-11/780 SYSTEM

TARGETED FOR DIVERSE MARKETS

BOSTON, Mass.--October 25, 1977--The 32-bit VAX-11/780 System introduced today by Digital Equipment Corporation will have widespread use among end users and OEMs in industrial, commercial, scientific and educational environments. It is aimed at satisfying the need for extended wordlength in many data acquisition and process control applications. Its unique hardware and software architecture are designed to maximize its performance of interactive, time-critical and computational tasks.

Here is what the managers of some of Digital's major market areas have to say about the VAX-11/780 system:

EDWARD A. KRAMER, VICE PRESIDENT, LABORATORY AND MEDICAL PRODUCTS GROUPS. "Government, university, and private research laboratories represent a major market for the VAX-11/780. Many scientific applications require two main capabilities in a computer system: very fast FORTRAN and the ability to handle extremely large programs. Crystallography and molecular research are just two fields where the ability to manipulate large data arrays easily is absolutely essential. The VAX-11/780 system provides these capabilities -- and more -- at a price which will make it very attractive to our customers. It is easy to use, enabling the

-more-

scientist to get his application up and running quickly. It can be connected to other PDP-11 systems and employed as a central development system in a laboratory network. The VAX-11/780's high-performance FORTRAN and floating-point accelerator option enable it to outperform any other machine in its class -- and several selling for twice the price.

"The VAX-11/780 continues Digital's leadership in scientific computing."

WILLIAM H. LONG, VICE PRESIDENT, OEM: "The new system's capabilities for rapid response and high total throughput are of significant value to original equipment manufacturers for such tasks as aircraft simulation, power monitoring and commercial processing.

"The VAX-11/780 is especially strong for time-critical applications," Long said. "Aircraft simulation, a growing market for OEMs, requires the computer system to supply all aircraft responses to pilot action in a flight simulator, using manufacturer-supplied data on several hundred elements of flight behavior and systems performance. The extended wordlength and raw speed of the VAX-11/780 are well suited to manipulating such large amounts of data in real time.

"For commercial OEM applications, the VAX-11/780's optimized design for multi-user interaction is well suited for high-volume transaction processing, while virtual memory features simplify writing and processing of large programs for inventory, bills of materials, material requirements planning and database management.

"In power monitoring, the VAX-11/780 can function as an upper level system in a hierarchical network monitoring power distribution for electric utilities," Long continued. "In this position, it would gather information from smaller systems at many distribution points, calculate load flows and compile energy distribution and system status reports for engineering and management use."

JULIUS MARCUS, VICE PRESIDENT, INFORMATION SYSTEMS: "The VAX-11/780 is a major technical achievement and milestone in computer systems design. It exemplifies our philosophy of product compatibility and provides an architectural extension to the high end of our PDP-11 family of systems. We see initial applications in our telephone industry and government markets and selected high-performance, data communications-intensive applications in the banking and transportation markets.

"Because of the critical nature of the intended applications, we placed much emphasis on development of the Reliability And Maintainability Program (RAMP) during system design. This feature enables both local and remote diagnostic analysis to expedite maintenance and repair. The expected result is substantially higher than average uptime for systems of this size.

"The initial software offering on the VAX-11/780 is extensive, providing operating systems, languages, and file systems. Major development programs are currently underway for extension of this software capability to include additional commercial software, which will round out the VAX-11/780's overall strength and applicability to all our markets involving general-purpose computing, EDP and transaction processing."

JERRY WITMORE, PRODUCT LINE MANAGER, EDUCATION PRODUCTS GROUP: "Colleges and universities are primary prospects for the VAX-11/780 educational installations. We see this system assuming a prominent role in university computation centers as the principal timesharing facility for student use. In addition, it would handle batch-oriented tasks in FORTRAN-IV-PLUS. In this way the VAX-11/780 could increase both the quantity and quality of total computer services, through its ability to serve a large number of interactive users, and its cost advantages when compared with the expense of enlarging a central mainframe. For college data processing centers the VAX-11/780 could act as the primary facility. It would

satisfy requirements for high-performance, general purpose timesharing, executing large FORTRAN programs, and running high-volume COBOL tasks for administrative processing and program development."

######

OPENING REMARKS

Before I start diving into some of the more technical characteristics of the VAX-11/780 system I would like to make a few general comments.

1. We, as a company have invested a great amount of effort and resourses in this program. Over 300 man years of effort. The industry is still technology driven. DIGITAL's customers have come to expect certain things from us; products with either a higher level of functionality and performance at the same price as existing products or new products with the same functionality at lower cost.
2. The VAX-11/780 system is not the average 32 bit computer. The system architecture, hardware and software is the result of a careful and lengthy design process. Just as for the PDP-8, the DEC 10/20 and the PDP-11, the VAX-11 architecture has to withstand the years; it must be adaptable to many user environments, some of which are not ever forseen today. Our customers depend on this type of stability and flexibility.
3. We believe that the new VAX-11/780 is a milestone in the field of interactive computers because we have combined in <u>one</u> system LARGE COMPUTER FUNCTIONALITY AND PERFORMANCE, RELIABILITY, MAINTAINABILITY, AVAILABILITY AND COMPATIBILITY WITH the most popular 16 bit computer, the PDP-11.
4. We have done this by taking a fresh look at total <u>system</u> design. In one sense we started from scratch, but in an other sense we started with the experience of 100,000 computers behind us.

A SYSTEM APPROACH

We started designing or perhaps more precisely architecting the machine from scratch with a team of hardware and software engineers working together right from the beginning.

We have made trade offs on a system wide basis.

The most obvious manifestation of this ground up, integrated design effort is reflected in the instruction set of the machine. I took two examples out of many to illustrate how hardware and software influenced each other.

ACBL, (ADD, COMPARE and BRANCH) translates exactly into one machine instruction, higher level languages constructs such as a FORTRAN DO LOOP. The PDP-11/70 which has one of the fastest FORTRAN compiler in the 16 bit computer world takes 7 instructions to translate the same DO LOOP. The result is more compact and faster compiler generated code.

INSQUE and its counterpart REMQUE lets the operating system scheduler insert or remove an entry in a doubly linked queue in one single instruction. RSX-11M an industry leader and our fastest REAL TIME MULTIPROGRAMMING operating system, takes 8 instructions to accomplish the same function. The result is faster program scheduling and less operating system overhead.

FUNCTIONALITY

The functionality of a computer system is often characterized by its operating system and associated software.

We have developed VAX/VMS, the VAX-11/780 operating system as a Single general purpose operating system to satisfy a broad range of functions.

For instance VAX/VMS supports

TIME CRITICAL APPLICATONS
64 INTERACTIVE USERS
MULTIPLE BATCH STREAMS
MULTIPLE LANGUAGES

We have applied mainframe software technology to VAX-11/780. VAX/VMS is a virtual memory operating system; it allows programs larger than physical memory to run in a fashion transparent to application programmers. I have been asked very often what is virtual memory? VIRTUAL MEMORY is memory that is not there, but the programmer does not know it! We have solved the inherent addressing limitations of the 16 bit computer by providing a very large addressing space of 4 BILLION Bytes. As a point of comparison a 16 bit architecture provides only 64K bytes or 128K bytes of addressing space.

We have built a hardware engine with a new architecture and 32 bits everywhere.
The machine is articulated around a high speed (13.3Mb/Sec) synchronous bus (a backplane in reality); some of the key features are

- 32 BIT INTERNAL BUS
- 32 BIT ARITHMETIC AND DATA PATH
- 243 BASIC INSTRUCTIONS
- 9 FUNDAMENTAL ADDRESSING MODES
- FLOATING POINT INSTRUCTIONS
- PACKED DECIMAL AND STRING INSTRUCTIONS
- PAGING WITH 4 HIERARCHICAL PROTECTION MODES
- 16 32 BIT REGISTERS
- 2Mb OF ECC MOS MEMORY (THIS IS REAL MEMORY NOW; IT'S THERE)
- UP TO 32 DISK DRIVES WITH 176Mb OF STORAGE EACH
- 800/1600 BPI TAPE
- CARD READER, LINE PRINTERS, TERMINALS

PERFORMANCE

The VAX-11/780 system can perform many functions as we have just seen. It can perform these functions very fast. Let's review a few of the factors which contribute to performance.

- A very fast interal bus
- Multiple caches for data, address translation, I/O buffering and operating system data bases
- A powerful instruction set
- A controllable and tunable paging scheme which allows time critical program to be swapped entirely (rather than paged) or locked in memory
- A scheduler with fixed and system optimized priorities
- A highly optimized FORTRAN compiler and a very fast optional floating point accelerator
  (1.4 usec for double precision ADD).
- The VAX-11/780 is positioned at the high end of the PDP-11 family in terms of performance and price, and below the DEC 2050 in terms of both price and performance.
- On average, 32 bit program execution and system throughput is roughly twice that of a comparably configured PDP-11/70.

RELIABILITY, AVAILABILITY, MAINTAINABILITY

VAX-11/780 serves a broad range of function, is fast and by design is RELIABLE, AVAILABLE, MAINTAINABLE.

Reliability, availability and maintainability features are found in the hardware architecture, the software architecture, the individual components and board design and in the packaging.

The objective: Keep the system running

If it fails find the fault quickly, fix it, get the machine up and running again

Protect the data

The list of features is long and impressive

Parity on buses, data path, control store, caches

ECC on memory

Consistency checks in hardware and software

History of bus activity

LSI-11 microcomputer for console operation, local and remote diagnostics

Floppy diskette for microdiagnostics loading and software updates distribution

Packaging with fixed back plane, cable troughs, modular power supplies, air flow and temperature sensors. On line diagnostics.

Error logging

We included these features by design. They cost money. We built them into the system, not because it is FUN, but because our customers expect it. This is the business we are in. It's just inappropriate for our machine to go down or stay down very long.

COMPATIBLE WITH THE PDP-11

VAX-11/780 is a new 32 bit machine; VAX/VMS is a new Virtual Memory Operating System.

By design the new system is enormously compatible with the other PDP-11's.

It is compatible where it counts: PEOPLE, DATA, USER PROGRAMS.

Compatibility is designed in, from the innermost to the outermost layers.

The new instruction set is not bit for bit compatible with the PDP-11's because we wanted performance and efficiency but both the PDP-11 and VAX-11 are

byte addressable machines
stack oriented

they have the same data types (VAX-11 has more)
the same instruction mnemonics
the same UNIBUS and MASSBUSES
the same PERIPHERALS

VAX-11 includes a 16 bit instruction set in its compatibility mode. The PDP-11/70 instructions are there with the exception of some privileged instructions such as HALT, I/O RESET etc.. Such instructions are not normally used by application programmers. The software or outer layer of the system has the same high degree of compatibility with the PDP-11 as the hardware.

- The ON DISK STRUCTURE is the same as RSX-11 and IAS. (VAX/VMS also has implemented extensions to it for more performance)
- The file access methods RMS are the same as RSX-11, IAS, RSTS/E
- The command languages DCL and MCR are the same as the ones found in IAS, RT-11 and RSX-11M
- The higher level languages are source compatible with their PDP-11's counterparts.
- The RSX-11M Application Migration Executive which exploits the 16 bit compatibility mode instruction set runs concurrently with other jobs under the control of VAX/VMS.
- The Application Migration Executive allows non privileged RSX-11M Tasks to execute on the VAX-11/780 with little or no modification.

We are taking advantage of this feature, ourselves - extensively. COBOL-11 and BASIC+2 as well as many utilities (perhaps as many as 200,000 lines of code) execute in compatibility mode and generate PDP-11 code.

- VAX-11/780 can be used as a host development system for RSX-11M and RSX-11S. All but the final debugging can be done on VAX-11/780.

To Conclude:
For the hundred of thousands of persons who have worked with and know the PDP-11 family, the new VAX-11/780 will be the simplest new system to learn.
For those who need more power than a PDP-11 can offer the VAX-11/780 system offers performance, functionality, up time, easy migration and great compatibility.

- A word about prices and delivery.
System prices start at $128,000 for a system with 128Kb of memory, 2 disk drives of 14Mb each, 8 asynchronous lines, a console terminal and the VAX/VMS operating system. A typical system configuration which includes 512Kb of memory. One 176Mb disk drive, an 800/1600 bpi magnetic tape, 8 asynchronous lines, a console terminal and the VAX/VMS operating system is priced at $185,000.
Deliveries are scheduled to start in early 1978; volume production is expected to be reached in mid 1978.

VAX11 780

NEWS RELEASE

digital

For Further Information:
Richard O. Berube
(617) 897-5111 Ext. 3046

DIGITAL'S VAX-11/780 SYSTEM:

A New Direction in Computer Development

BOSTON, Mass.--October 25, 1977--The new VAX-11/780 system introduced today by Digital Equipment Corporation combines the functionality, capacity and performance usually found only on large mainframe systems with the best features of minicomputers, and offers them at a remarkably low price.

Highlights of the new system include 32-bit wordlength, four billion bytes of virtual addressing space, a new Virtual Memory operation system, compatibility with Digital's 16-bit PDP-11 family and built-in reliability and maintenance design innovations.

According to Andrew C. Knowles, Digital Vice President and Group Manager, the new system "provides the high capacity, wordlength and throughput of conventional mainframes together with the interactive capabilities, design innovation and price/performance features of a minicomputer. With this level of designed-in flexibility, the VAX-11/780 is suited to a wide variety of applications in scientific/time-critical, computational, control, data processing and interactive timesharing. And it has impressive multi-user batch capabilities as well," he said.

-more-

VAX-11/780: THE HARDWARE

- 32-bit wordlength provides up to 4.3 billion bytes of virtual addressing space.

- Main memory subsystem is ECC MOS memory using 4K MOS RAM chips. Minimum system configuration provides 128K bytes of physical memory which is expandable up to 2 million bytes.

- Memory controller includes request buffer which increases system throughput and eliminates most of the need for interleaving.

- Complete and powerful instruction set consists of 243 instructions, 9 addressing modes and 5 data types. Designed for the generation of fast, efficient compiled code, the instruction set includes integral floating point, packed decimal arithmetic, character string manipulation and context switching instructions.

  As an example of efficient code generation, a FORTRAN DO loop translates into one instruction. Calls to subroutines, and return to main program combine up to 15 operations in just one instruction.

- 8K byte write-through cache memory yields effective memory access time of 290 nanoseconds.

- Optional floating point accelerator performs double precision floating point 64-bit addition in 1.4 microseconds.

- Paging memory management is supported with 4 hierarchical protection modes, each with read-write access control.

- Sixteen 32-bit general registers and 32 interrupt priority levels, 16 each for hardware and software.

- Two standard clocks: programmable real time clock and time-of-year clock with battery backup for automatic system restart.

- A Synchronous Backplane Interconnect (SBI) serves as the main control and data transfer path. It is capable of aggregate throughput rate of 13.3 million bytes per second.

- MASSBUS interfacing adapter permits connection of high-speed PDP-11 Peripheral devices (e.g. RP06 Disks and TE16 Mag Tapes); UNIBUS interfacing adapter allows connection of conventional PDP-11 peripherals (e.g. smaller disks, CRTs and printers). One UNIBUS adapter and up to four MASSBUS adapters can be connected to the backplane.

-more-

VAX-11/780: THE HARDWARE (continued)

- o MASSBUS connects to the SBI via a buffered adapter and permitting the interfacing of high performance mass storage peripherals with parity checking. Throughput rate here is two million bytes per second.

- o Adapter pathway between UNIBUS and SBI has a throughput rate of 1.5 million bytes per second.

- o Console subsystem incorporates intelligent LSI-11 microcomputer with 16K bytes or read-write memory and 8K bytes of read-only memory, single floppy disk and LA36 teleprinter.

- o The console permits simplified bootstrapping, improved distribution of software updates and fast on-line diagnosis, either local or remote.

VAX-11/780: THE SOFTWARE

VAX-11/780 boasts a new virtual memory operating system, VAX/VMS, which applies mainframe software technology by allowing programs much larger than the physical memory to be run in a way that is transparent to the programmer. Essentially, the new system will take any size program.

- o Single virtual memory operating system for multiple functions.

- o Full demand paging operation permits programs as large as 32 million bytes.

- o Memory management facilities can be controlled by the user, who can lock pages of a program in memory never to be paged out or can lock an entire program in memory never to be swapped out. This feature is particularly important for time-critical applications.

- o System supports 64 interactive users simultaneously.

- o Program development capabilities include two editors, language processors, symbolic debugger, librarian, and utilities.

- o Languages include VAX-11 FORTRAN IV PLUS, VAX-11 MACRO, PDP-11 COBOL and PDP-11 BASIC-PLUS-2, with FORTRAN and MACRO generating 32-bit native code on the VAX-11/780.

- o Operating system provides file and record management facility allowing users to create, access and maintain data files and records with full protection.

VAX-11/780: THE SOFTWARE (continued)

- o New operating system supports networking capabilities for task-to-task, access and file transfer and down-line loading.
- o Batch capabilities include job control, multi-stream, spooled input and output, operator control, conditional command branching and accounting.
- o DIGITAL command language (DCL) and MCR command languages provided.
- o 32 levels of software process priority for fast scheduling.
- o Record and file management facilities include sequential and relative file organization, sequential and random access.
- o Applications Migration Executive allows RSX-11M/S non-privileged tasks to run with minimal or no modification.

RELIABILITY, AVAILABILITY AND MAINTAINABILITY PROGRAM

The VAX-11/780 system is designed to be the most reliable, available and maintainable computer system of its class built to date, through the inclusion of reliability and maintenance design innovations. These features have been designed into the hardware architecture and software architecture, individual component and board designs and in the cabinetry.

A diagnostic console contains an LSI-11 microcomputer which provides automatic consistency and error checking to detect abnormal instruction uses or illegal machine conditions. Integral fault detection and maintenance features detect errors on memory or disks, record recent bus activity, detect hung machine conditions and allow restart recovery.

Among the several monitoring activities performed automatically are parity checking for data integrity on the synchronous backplane interconnect, MASSBUS and UNIBUS adaptors, memory cache, address translation buffer, and error detection and correction (ECC) on memory. Also performed are operating system consistency checks, redundant recording of critical information, uniform exception handling, on-line error logging, on-line diagnostics and unattended automatic restart.

-more-

PDP-11 COMPATIBILITY

According to Digital's Bernard LaCroute, product manager for the VAX-11/780, "one of the design goals for the new systems was compatibility with other PDP-11's. The result is an instruction set for the new 32-bit system that is extremely rich, through the use of microprogrammed logic. The new instruction set has the same mneumonics as the PDP-11. The system also includes a compatability mode which provides the PDP-11 instruction set, with the exception of privileged and floating point instructions.

Like other PDP-11s, VAX-11/780 uses both DCL and MCR command languages and implements the same FORTRAN-IV-PLUS, BASIC-PLUS-2 and COBOL languages. FORTRAN generates native 32-bit code on the VAX-11/780, and can concurrently execute a subset of the PDP-11 instruction set in its "compatibility" mode.

The VAX-11/780 system can also be used as a host development system for RSX-11M and RSX-11S operating systems running on PDP-11s. Like other PDP-11s, the new 11/780 uses a UNIBUS for connecting to peripherals, and like the PDP-11/70 it uses integrated MASSBUS adapters for interfacing high-speed peripherals.

The on disk structure is the same as RSX-11, and IAS; the RMS file access methods are the same as RSX-11, IAS and RSTS/E.

"In sum, our conscious design of the new system to be compatible with the 50,000 PDP-11 systems installed throughout the world will make it simple for the hundreds of thousands of people who have worked with the PDP-11 family to make an easy migration up to the VAX-11/780 system," he said.

-more-

SYSTEM CONFIGURATIONS

For end users, the VAX-11/780 system will be offered in three standard system configurations:

- Minimum system configuration consists of VAX-11/780 CPU with 128K bytes of ECC MOS memory, LA36 DECwriter II console terminal, two RK06 14-megabyte disk drives and a multiplexer that provides eight EIA terminal connections and VAX/VMS operating system. Price: $128,000.

- CPU with 256K bytes of ECC MOS memory, one RM03 67-megabyte high performance disk, one TE16 mag tape drive, and 8-line multiplexer and VAX/VMS operating system. Price: $153,000.

- CPU with 512K bytes of ECC MOS memory, one RP06 176-megabyte high performance disk drive and one TE16 800/1600bpi magnetic tape drive, an 8-line multiplexer and VAX/VMS operating system.

- All systems have provisions for additional memory and peripherals.

- System components will be available to OEMs.

VAX-11/780 QUESTIONS and ANSWERS

For Press Conference and Stockholder Meeting

PDP-11 and DEC 20

Q - How is this machine going to impact the PDP-11 business? What is the future of the PDP-11.

A - The VAX-11/780 is an extension of the PDP-11; it offers more functionality and performance for those customers who need it while complementing the PDP-11 offerings. PDP-11 hardware and software will continue to be aggressively enhanced to maintain their price performance leadership in the 16 bit world.

Q - Do you expect in five years to have the same ratio of business between VAX and the PDP-11 as you have today between the PDP-11 and the PDP-8?

A - The ratio is not that important; what counts is the right set of products at the right time to meet our customers needs.

Q - How is this machine going to impact the DEC 10/20 business? What is the future of the DEC 10/20?

A - The VAX-11/780 is an extension to the PDP-11 and will be sold for the same type of applications as well as new ones which require more functionality. Just as for the PDP-11 the DEC 10/20 will continue to be enhanced. VAX-11/780 complements the PDP-11 and DEC 10/20 offerings.

Q - Who would want to buy a PDP-11/70 when there is only a $30,000 difference between the two systems?

A - Those customers who do not need the 32 bit functionality of the VAX-11/780. Why would somebody pay more to get something they don't need?

PDP-11 and DEC 20 (cont.)

Q - Why didn't you choose a 36 bit architecture for your new machine? Is 36 bit obsolete?

A - We wanted the new machine to key off the PDP-11 for compatibility reasons. This has nothing to do with our 36 bit architecture obsolescence.

Q - What do you anticipate the average VAX configuration to be? How does it relate to the PDP-11/70 and the DEC 20?

A - The VAX-11/780 system configurations start at $128,000. A configuration with ½ Mb memory, a 176 Mb disk drive and tape is priced at $185,000.

The PDP-11/70 average system is around $200,000; the average DEC 20 system is around $450,000. We expect the average VAX-11/780 configuration to be in the $250,000 - $300,000 range.

Q - Did you cut your PDP-11/70 prices to make room for this new machine?

A - No - PDP-11/70 price reduction reflects our increased manufacturing efficiency; so does the pricing of the VAX-11/780.

VAX-11/780 Specific

Q - Are you now spending most of your R & D dollars on VAX now?

A - No. No comments about any figures.

Q - You have acknowledged spending a lot of resources developing this machine. When is the next VAX machine to be expected and where is it going to be positioned?

A - Just as for the PDP-8, PDP-11, DEC 20 we will have the right product at the right time.

Q - Is this machine a minicomputer or a mainframe?

A - It is what you want it to be; mainframe capabilities at minicomputer prices.

Q - How many machines are you building in the first year?

A - The number is expressed in hundreds.

Q - What is the anticipated yearly number of machines at volume production?

A - As many as we need to satisfy our customers demand.

Q - Who do you anticipate your customers/markets to be?

A - The traditional PDP-11 customers and applications plus those new applications which need 32 bit word length in the OEM, Scientific, Realtime computation market place.

Q - Are you going after the commercial market with this machine?

A - In the same sense as we are with the PDP-11 today.

Q - When are you going to add COBOL?

A - We do have PDP-11 COBOL on the machine.

Q - What is the performance of the machine?

A - On average twice the speed of the PDP-11/70 for 32 bit programs and operations.

Q - Is there any new technology in the machine (ECL, LSI etc.)?

A - No, we are using conventional Schotky TTL logic.

Q - When are you shipping your first machine?

A - We will be shipping our first machine to a customer testing environment this coming month. Production machines will follow in early calendar 1978 and full production in mid-calendar 1978. (Do not disclose customer names).

VAX-11/780 Specific (Con't'd.)

Q - Have you identified your first customers?

A - Yes, as part of our test marketing effort. (do not disclose customer names).

Q - How many machines have you built so far?

A - Nine. Several more are to follow shortly.

Q - Where is the machine going to be manufactured?

A - In our New Hampshire facilities.

Q - Will you manufacture VAX in Europe? Where? When?

A - The machine will be built initially in New Hampshire. We will consider manufacturing it in Europe later if it makes sense.

Q - Why are you making such a big announcement? It is not the traditional DEC approach.

A - To make sure that we get you the right information about the significance of VAX.

Q - Why didn't you use the acronym PDP?

A - The PDP acronym is implied in the name; VAX-11 stands for Virtual Address Extension to the (PDP) 11.

IBM and COMPETITION

Q - Does this new machine signal DEC's entry in the "IBM World"?

A - NO. VAX-11/780 is an interactive computer designed to serve our traditional markets as well as new applications which require greater word length functionality.

Q - Which IBM machine are you competing (and/or comparing) with?

A - VAX-11/780 is an interactive computer designed to serve our traditional markets as well as new applications which require greater word length functionality. IBM machines are primarily Batch oriented; we can't calibrate ourselves against them.

Q - Is this machine going to replace installed IBM equipment?

A - See answer to previous question.

Q - How do you stack up against the IBM 370 line in terms of raw computer power - or which IBM 370?

A - Very well; but our customers don't use the machine the same way. Ours are interactive, IBM is Batch oriented

Q - The VAX-11/780 has a PDP-11 emulation capability. Are you also planning to emulate the IBM 370?

A - NO; we have never considered this in our design.

Q - Why are you so late with your 32 bit machine, some of your competitors (SEL, INTERDATA) have had one for two years?

A - We don't think we are late; we have the right product at the right time. If you want to compare let's discuss performance and functionality!

Q - When do you expect Data General to follow suit with their 32 bit machine?

A - We don't know; we have the product today, they don't.

Q - How does this relate to the recent 32 bit WANG announcement?

A - We have not had time to look at it.

AGENDA

VAX Press Conference/Stockholders Meeting

Tuesday, October 25, 1977

Dorothy Quincy Suite, John Hancock Building

VAX PRESS CONFERENCE (Conference Room)

9:45am -- Welcome and Introduction: WIN HINDLE

9:50am -- DEC product philosophy, family evolution
VAX design goals: GORDON BELL

10:05am -- VAX Technical presentation: BERNIE LA CROUTE

10:25am -- VAX Markets and Applications: ANDY KNOWLES

10:35am -- Questions and Answers (Hindle, Knowles, Bell, LaCroute, Demmer, et al)

10:55am -- Press adjourn to Shareholders Meeting

SHAREHOLDERS MEETING (Dorothy Quincy Suite)

11:00am -- Call to order:)
)
11:15am -- Informal Remarks:) KEN OLSEN
)
11:30am -- Stockholder Q&A:)

12 noon -- Adjourn Annual Meeting
VAX Demo for Stockholders: TOM RARICH, et al

12:10pm -- KHO press conference (conference room)

12:40pm -- Adjourn KHO press conference
(Press return to the Dorothy Quincy Suite for VAX Demo)

1:15pm -- Press luncheon (Top of the Hub at the Prudential)

JULY 6, 1978

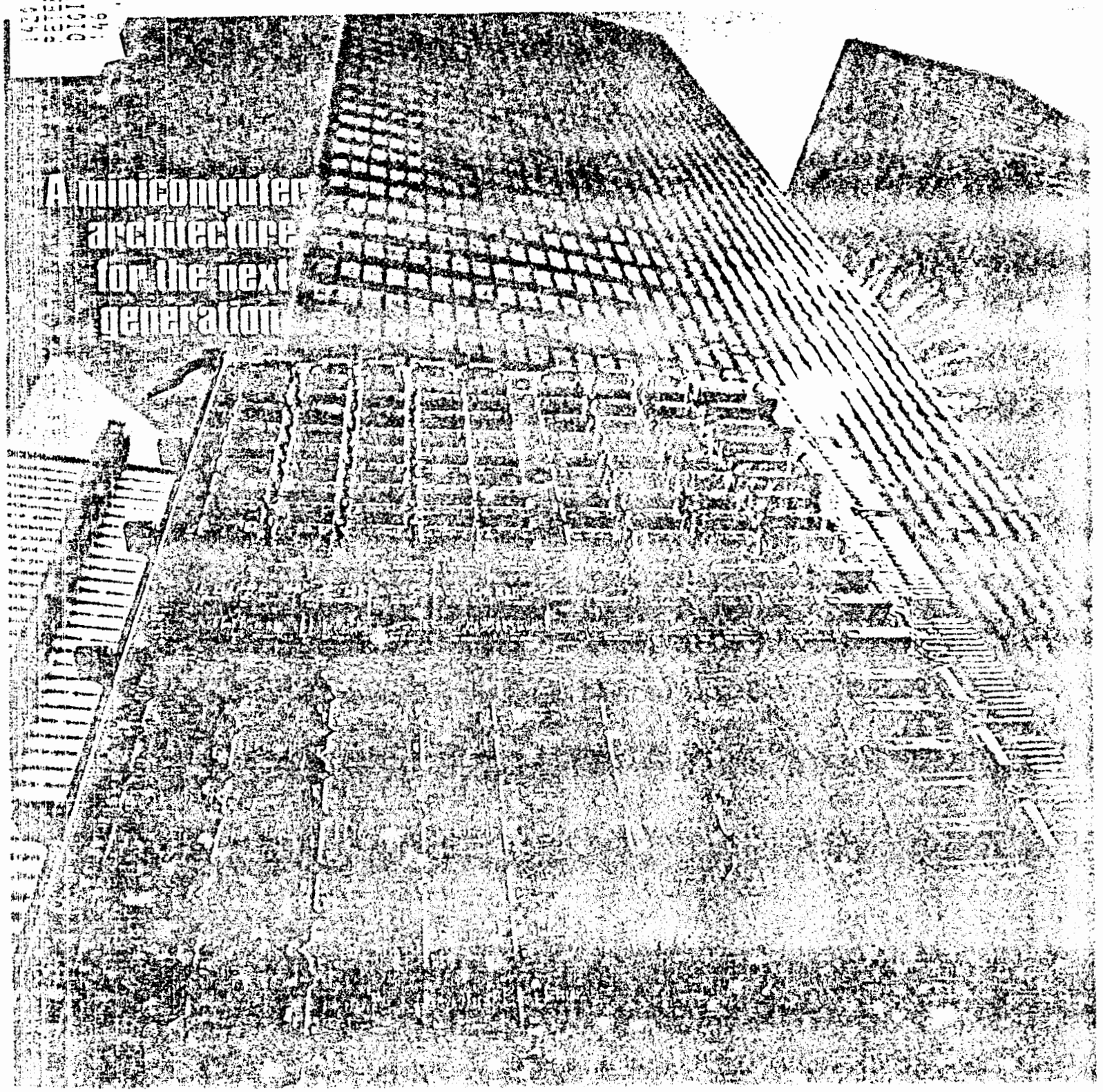
A NEW CONSUMER INDUSTRY: ELECTRONIC PHONES/81

C-MOS erasable PROM uses single power supply/106

A guide to thermal resistance measurements in ICs/121

FOUR DOLLARS A McGRAW HILL PUBLICATION

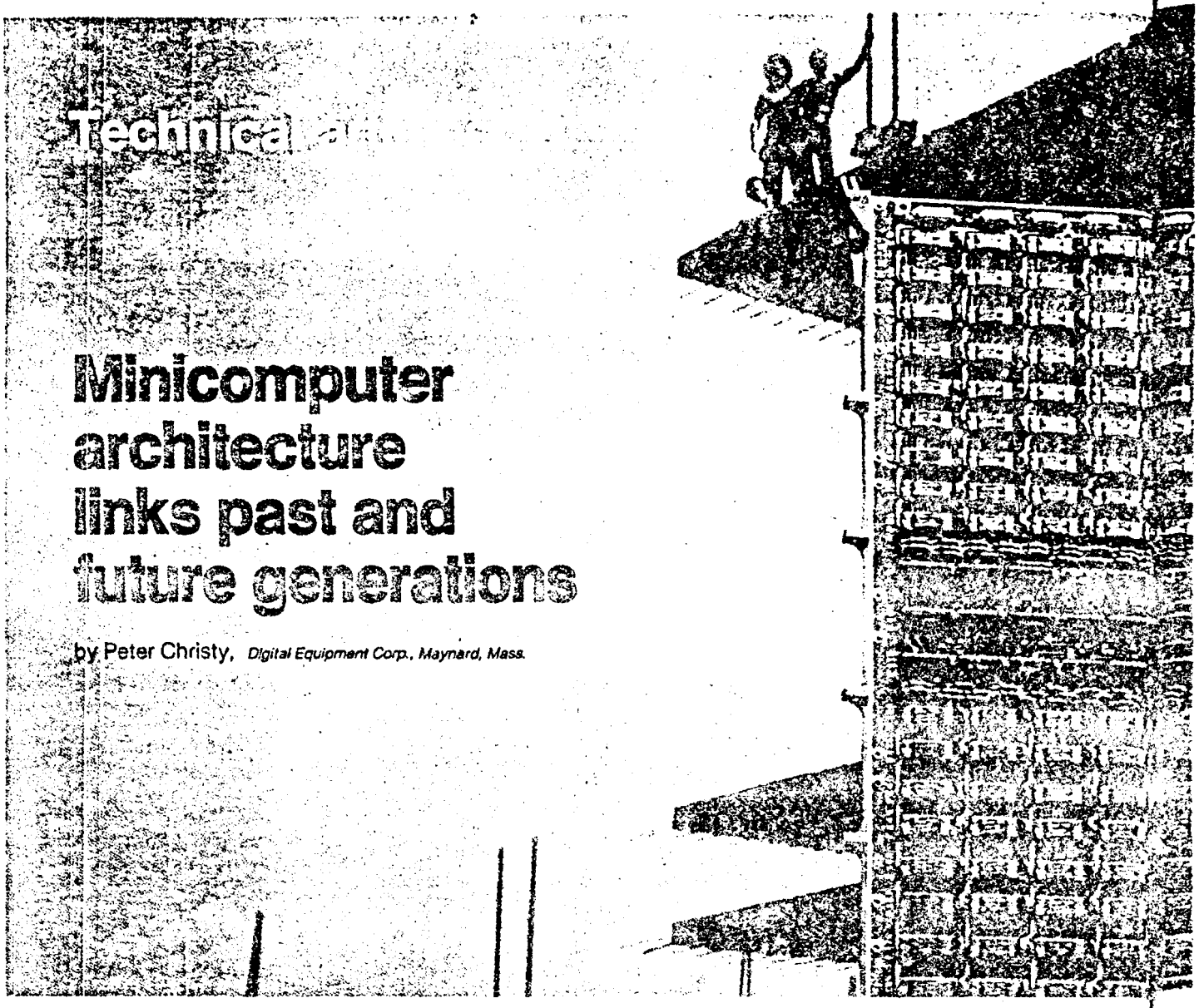
Electronics



Technical

Minicomputer architecture links past and future generations

by Peter Christy, *Digital Equipment Corp., Maynard, Mass.*



☐ The design and planning of a new series of minicomputers is a difficult problem, especially for a company with a large installed base of a highly successful family. And the problem is greatly magnified when the proposal is for the new family to overstep what were earlier regarded as a minicomputer's limits.

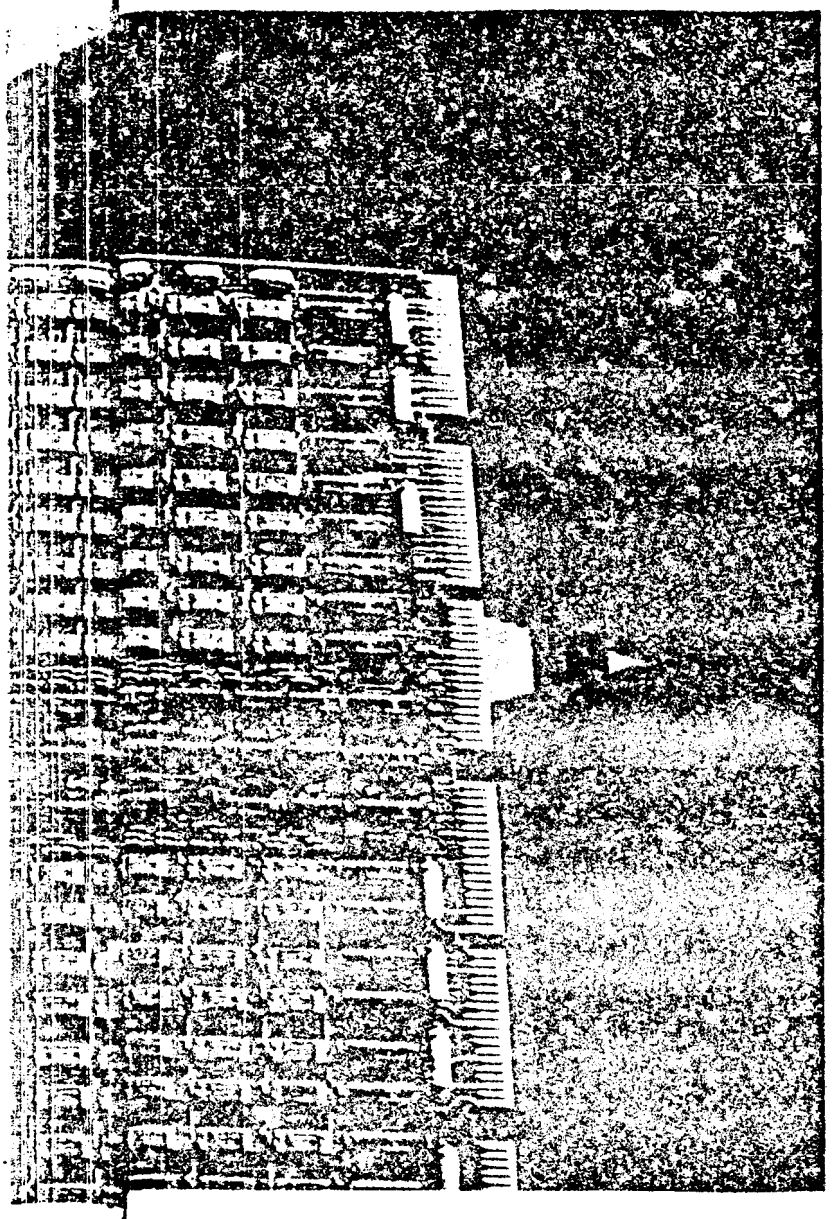
Thus, when Digital Equipment Corp. decided to extend its line into 32-bit mainframe territory, it set in motion a series of complex design decisions requiring a thoroughgoing reexamination of minicomputer architecture in the light both of likely user needs through the 1980s and of likely technological progress through the same period.

This article addresses some of the issues that guided the development of the VAX-11 architecture, the VAX-11/780 computer system, and the VAX/VMS operating system. The goal was to preserve compatibility with the existing large software investment in the PDP-11 minicomputer family yet to reflect future system needs, in particular by enlarging virtual address space to a huge 4 gigabytes. VAX, in fact, stands for virtual address extension, and VMS stands for virtual memory system.

The VAX-11/780 32-bit minicomputer system is the high end of the new family. A typical configuration costs between $150,000 and $200,000. Initial benchmarks show the machine's Fortran performance, using its fast-floating-point option, to be comparable to that of a modern upper-to-middle-range mainframe costing several times as much. Yet processor, optional floating-point unit, up to 1 megabyte of metal-oxide-semiconductor random-access memory, a Unibus medium-speed input/output controller, and two high-speed (Massbus) I/O controllers come in a single cabinet measuring 47 by 60 by 30 inches (Fig. 1). More memory and various options can be added in extender cabinets.

Parts

All this was implemented with conventional Schottky transistor-transistor logic and standard large-scale integrated memory circuits. Indeed, it was the ready availability of fast, high-density read-only memory that made it possible to design a complex processor without resort-



ing to anything more expensive than microcoding techniques. Besides the ROM control store, the central processing unit includes 12 kilobytes of RAM control store, which is used for diagnostic functions, some special instructions, and field microcode changes. A further 12 kilobytes of RAM control store is available as an option.

Other RAM parts are used throughout the system to increase performance. The CPU includes some in the form of an 18-kilobyte cache, which keeps the most recently used instructions and data quickly accessible to the processor. Also included in the CPU is a 128-entry address-translation buffer, which is functionally analogous to memory-mapping hardware: it keeps the most recently used translations between virtual and physical memory in high-speed registers, greatly reducing the memory management overhead. RAM is also used throughout the memory busing and I/O subsystem to increase the efficiency of the major busing mechanisms.

As for the new VAX/VMS operating system, it provides the VAX-11/780 minicomputer with the kind of functions previously available only to mainframe computers. Examples are full virtual-memory management, demand paging, indexed data-access methods, and extensive interjob protection and sharing capabilities. VAX/VMS supports up to 64 on-line users simultaneously developing and executing programs in assorted high-level languages. In particular, a compiler for DEC's Fortran IV-Plus language (a superset of ANSI Fortran) has been developed to take full advantage of the extended instruction set of the VAX-11 architecture.

Compatibility with the PDP-11

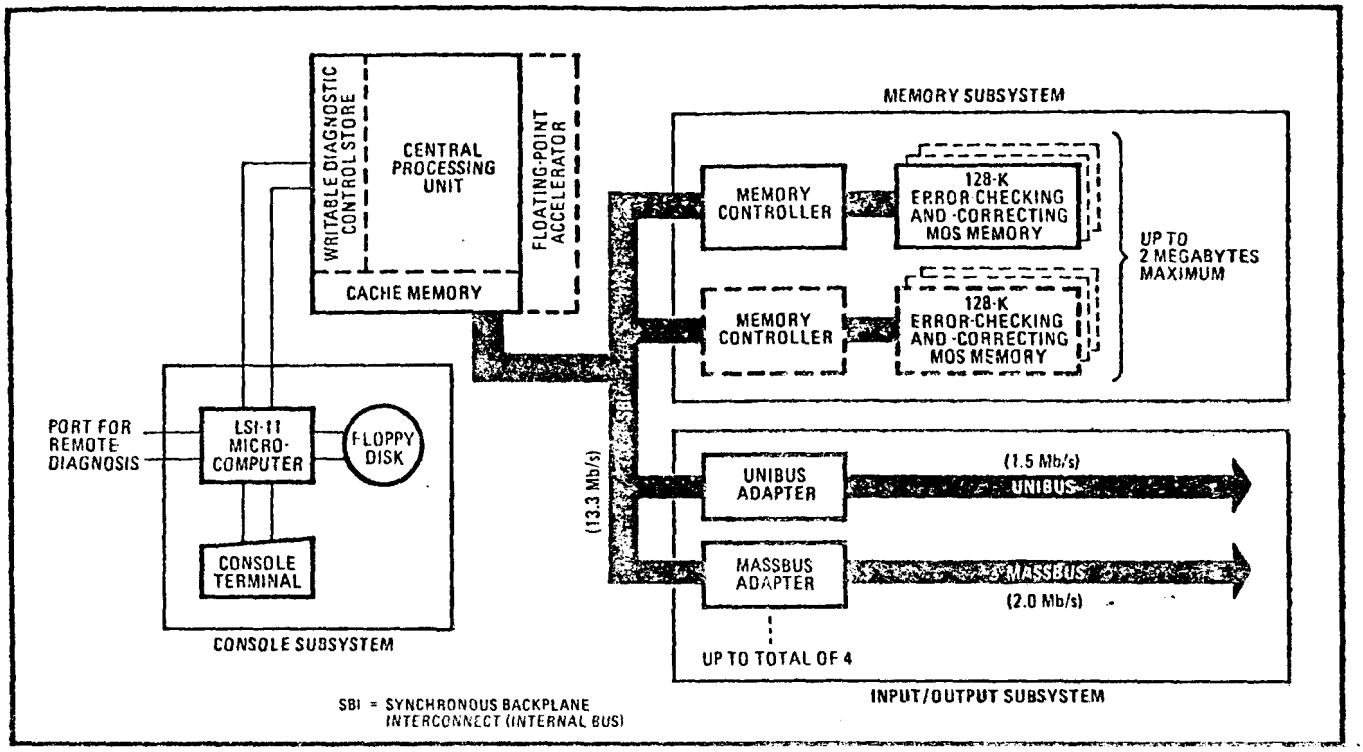
Though the VAX-11/780 is not a 32-bit PDP-11, for reasons that will shortly be gone into at length, cost-saving compatibility with the PDP-11 and its associated software has been achieved as desired at the six most relevant user levels:

- Cultural compatibility, if such a term may be used to describe the stylistic similarity of the machines. Because of it, PDP-11 programmers can produce high-quality VAX-11 native-mode code with little training, and language compiler designs that generate PDP-11 code can be adapted to generate efficient VAX-11 code.
- Operating-system compatibility. Many of the VAX/VMS operating-system functions were modeled after the PDP-11 RSX operating systems (for example, in the type and form of system calls, the way in which tasks synchronize with each other and exchange data, etc.). Key VAX/VMS functions, such as the file system and record management facilities, are functionally identical to their RSX counterparts.
- High-level language compatibility. VAX-11 languages are designed to be compatible with the existing PDP-11 compilers. A calculation program written in Fortran IV-Plus for the PDP-11 runs unchanged on VAX-11.
- Direct processor support for user-mode PDP-11 programs. The VAX-11 architecture includes a PDP-11 compatibility mode, in which the processor behaves just like a user-mode PDP-11, except that it can simultaneously run 32-bit code jobs.
- Data compatibility. All PDP-11 data formats were brought forward to the VAX-11 architecture.
- Data-file compatibility. VAX/VMS is able to create and access disk files that are compatible with the PDP-11 RSX-11 operating systems.

Why not a 32-bit PDP-11?

During planning of the VAX-11, one idea that received serious consideration was in fact a 32-bit PDP-11. Most of the PDP-11 architecture is already independent of word length, and recent architectural studies had demonstrated that the PDP-11 is a bit-efficient architecture, even compared with mainframe architectures. In short, a PDP-11-like machine with an extended virtual address space would evidently be an attractive computer, today and tomorrow.

The most distinctive attribute of the PDP-11, and the basis for its architectural power, is the flexible way in which its registers can be used to form addresses. This flexibility permits the machine to be used effectively for many different types of computing, unlike most previous architectures, which tended to be good for one style of



1. New generation. The VAX-11/780 computer system consists of the central processing unit, the console subsystem, which serves as an operating system terminal and system or diagnostic console, the main memory subsystem, and the I/O subsystem. All major hardware components, implemented by Schottky TTL and standard MOS memory devices, are connected through the SBI, an internal synchronous bus.

processing but poor for another. For example, a design that has many central registers but not stack-like characteristics is good for scientific calculation but poor for complex subroutine structures. Conversely, a machine designed around a stack architecture is good for program control but inefficient for intensive calculation. But the PDP-11 is able to take on either set of attributes, and others, whenever a task demands it.

However, a 32-bit PDP-11 would have meant extending the register width to 32 bits but keeping the instruction formats and encodings unchanged, and this turned out to be an impossibility. The idea would have been a machine that could execute existing PDP-11 machine code intermingled with 32-bit code that made full use of the 4-gigabyte virtual address space. But a careful examination of a 32-bit version of the PDP-11 uncovered some unsurmountable obstacles.

There turn out to be many ways in which a programmer can implicitly design the address length into a program. For example, before control is passed to a subroutine, parameters may be pushed onto the stack. The subroutine call itself leaves the return address on the top of the stack. Within the subroutine, the parameters are accessed with respect to a known displacement from the top of the stack. But unfortunately, changing the address length from 2 to 4 bytes makes these known offsets invalid. This and many similar problems ruled out the possibility of executing 16-bit code unchanged in a larger address space, or of automatically translating 16-bit programs into a 32-bit form.

Given the difficulties of directly extending the PDP-11 design to a 32-bit form, the next alternative was to see what improvements could be gained by a bit-level-incompatible, but otherwise highly similar, design. The result was the VAX-11, a substantially better design that, though not precisely like the PDP-11, is "culturally compatible" with it. Hardware and software were also developed that permit a large subset of existing PDP-11 programs to execute without any changes on a VAX-11 system, as described earlier.

Architectures and word lengths

The description of the VAX-11 as a 32-bit minicomputer and the PDP-11 as a 16-bit minicomputer implies that the essential difference between them is their word length. But any significant difference in the architectures would presumably be measurable in terms of their comparative bit efficiencies on important applications. As it turns out, the bit efficiency of the PDP-11 is excellent, and in most respects the PDP-11 is not restricted to a 16-bit word length.

The problem is that the term "word length" has too many meanings to be useful without qualification. In a typical computer system, many different word lengths can be identified. In this context, therefore, it is necessary to eliminate from consideration the word lengths that represent engineering decisions for specific implementations and to consider only those that are intrinsic to an architecture and affect all its implementations in the family.

Instruction length is a possible candidate here. But both the PDP-11 and the VAX-11 have instructions of variable length, ranging from 16 to 48 bits and 8 to 296 bits respectively. In both cases, the variable-length instruction format offers better bit efficiency than an equal-length format because common instructions can

The importance of bit efficiency

A good architecture is reflected in a computer's static and dynamic bit efficiency. Bit efficiency is a quantifiable measure of how well the investment in the computer system's components pays off in application-level throughput. In other words, if two systems are built with the same technology and the same complexity, then the one with the greater bit efficiency will be more cost-effective (assuming that the bit-efficient instructions can still be rapidly decoded and executed by the processor).

Static bit efficiency is the relative size of a program compared with the size of a program coded for an architecture defined as a standard. A good static bit efficiency reduces the requirements for central memory and program file storage and streamlines the tasks involved in program-moving overhead, such as initial program loading, fetching overlays, paging, or swapping.

Dynamic bit efficiency is a comparative measure of how many program bits must be fetched from memory to the processor to execute a program. If all machine instructions were used with the same frequency, then static and dynamic bit efficiency would be the same. In practice, some instructions and data types occur often and others occur rarely. Good dynamic bit efficiency reflects the fact that the most frequent instructions (such as loop control instructions) have particularly good encoding.

The ideal way to compare bit efficiencies would be to take a specific set of application programs and measure their actual bit efficiencies on different architectures. Unfortunately, such an approach is impractical for a computer vendor because customers have many disparate applications and many architectures of interest are hypothetical.

Fortunately, there are ways to characterize typical applications. Those coded in common high-level languages, such as Fortran, Cobol, and Basic, may be related to studies of typical program behavior, which show that in each of these languages different statements and data types have a characteristic frequency of occurrence. With these statistics and with an understanding of the machine code generated for each common statement, it is possible to estimate the bit efficiency of real or hypothetical architectures.

Bit efficiency is a good general test of architectural effectiveness, since it diminishes with any difficulty in machine-level programming or compiler code generation. Good static bit efficiency reflects effective use of system components; good dynamic bit efficiency reflects effective use of memory system bandwidth.

have shorter encodings. No architecturally useful definition of word length can be derived from instruction length, therefore.

Both the PDP-11 and VAX-11 are byte-address machines, since all data types are addressed in main memory by the byte address at the beginning of the data item, regardless of whether the data is a 1-byte character or an 8-byte double-precision, floating-point number. So memory addressing is also no help in defining the architectural difference between the machines.

But the PDP-11 has 16-bit general registers, whereas the VAX-11 has 32-bit general registers. In both architectures the registers can be used for arithmetic on data items that are shorter than the register size (8-bit integers on the PDP-11, 8- and 16-bit integers on VAX-11) or can be used in multiples for data items that are longer than the register size. Register length as used in arithmetic is not an invariable word length, therefore.

Register length

However, register length as used in instruction address formation is another matter: it does define the essential difference between the PDP-11 and the VAX-11, since it determines the size of logical storage that a program can instantaneously address—the size of the virtual address space. Thus the PDP-11's 16-bit byte address creates a virtual address space of 65,536 bytes, whereas the VAX-11's 32-bit byte address creates more than 4 billion bytes of virtual address space. What's more, virtual address space limitations can affect bit efficiency.

At the time the PDP-11 was designed, it seemed unlikely that any minicomputer would need more than 65 kilobytes of *physical* (as opposed to virtual) memory. In retrospect, the designers realize they failed to anticipate how rapidly central memory costs would decline.

Early in the evolution of the PDP-11 family, hardware memory mapping was added to the top-range machines. PDP-11 mapping logically divides the 64-K virtual address space into eight 8-K pages, each of which can be located independently in physical memory and protected independently (Fig. 2).

The addition of mapping offered two major benefits:

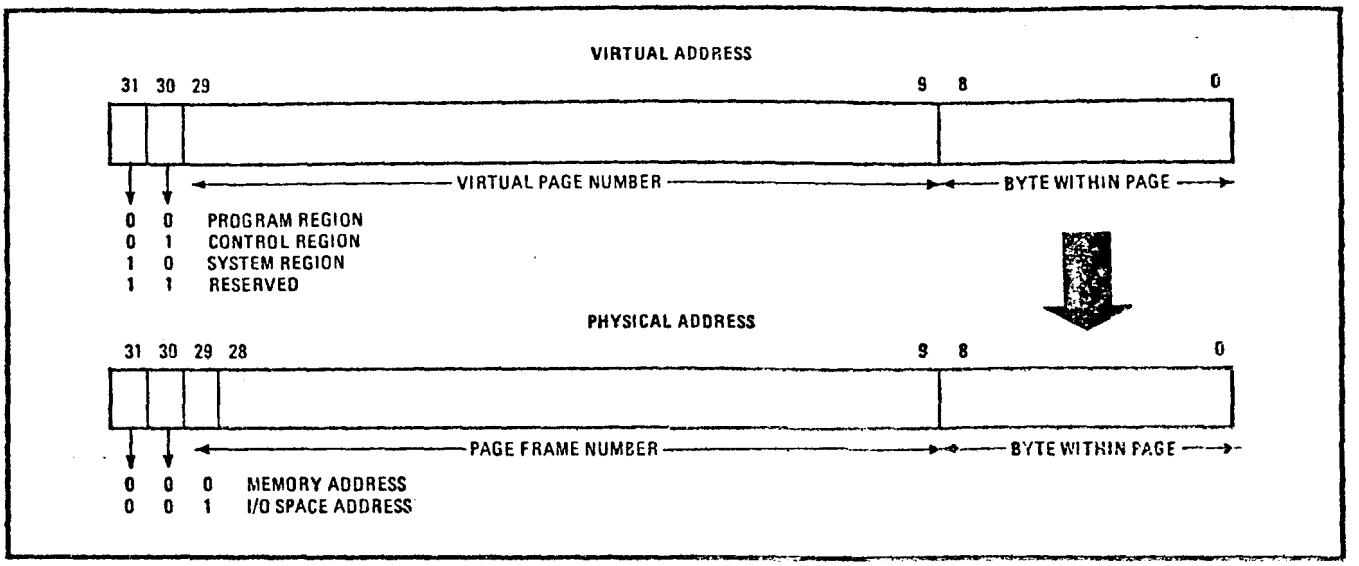
- It permitted the design of multiprogramming software systems in which a user program is prevented from damaging another user program or the operating system code, since each program can address only those parts of central memory allocated to it.
- It permitted the design of configurations with more than 65 kilobytes of physical memory, since the mapping hardware can translate 16-bit addresses into physical memory addresses of arbitrary length. Thus a PDP-11/70 may have in excess of 4 million bytes of central memory by developing 22-bit addresses.

For most applications, the remapping overhead is insignificant. But there are calculation applications in which it induces noticeable bit inefficiency. Also, although today most minicomputer programs and their data fit naturally into a 65-kilobyte address space, the trend to larger central memories will surely lead to larger program sizes as well.

The need for large address space

Consequently, though the immense marketplace success of the PDP-11 and other minicomputer architectures demonstrates that limited address space has not so far been felt as a restriction, it might become one in the future. Since the need for large address space is felt first in large configurations, the VAX-11/780 was designed as a top-of-the-line minicomputer.

The fundamental need of VAX-11 was to solve the



2. Address formats. Virtual addresses are 32 bits long in format shown at top. The virtual address space (memory space as it appears to a given process) is mapped onto the physical address space (lower format), which is the actual memory space existing in hardware. Memory-management logic controls the mapping and maintains in physical memory only portions of virtual memory being used.

addressing problem. Given the byte orientation of the PDP-11 and the need to do address arithmetic conveniently, the obvious address lengths to explore were 24 bits (16+8), 32 bits (16+8+8), etc.

Memory costs are roughly halving each year, and 1 more bit of physical address per year is needed. A 24-bit virtual address seemed too small. For VAX-11 the choice was 32 bits, representing a 4-gigabyte virtual address space (see Fig. 2 again). The next decision was to make this address space linearly addressable, meaning that there would be no further segmentation (many mainframes offer comparably large virtual address spaces but break them into many segments). This is large by any standard today, including mainframes, and should also allow a good decade of growth before the size of typical physical memories comes anywhere near the virtual address space. The very large, linear, virtual address space will also permit flexible evolution in software system design techniques, should that prove needed, for example, in advanced file or data-base management system designs.

Picking the virtual address size was a key decision, but a simple one compared with the total question of virtual memory design. Other issues included:

- Feasibility. It had to be possible to implement without adding a large cost penalty to the processor.
- Functionality. It had to be able to support sophisticated data-processing applications.
- Efficient use of control memory. The VAX-11 architecture had to exhibit a wide range of system performance without requiring large amounts of central memory for control tables.
- System efficiency. A virtual memory design would have to work well in real-time applications, yet be able to switch rapidly from one program to another while responding to external requests.

A 32-bit virtual memory design is totally different from a 16-bit one. For example, mapping a 65-kilobyte PDP-11 program consists of initializing eight internal processor registers. In contrast, mapping a 4-gigabyte address space would mean specifying 8 million page relocations (on VAX-11, each page or separately mappable unit of virtual memory is 512 bytes). Obviously, no VAX-11 processor will have 8 million internal relocation registers!

Similarly, the operating system on a PDP-11 does little in the way of virtual memory management. Because programs are relatively small, they are typically swapped in and out in their entirety. But the large virtual address space on VAX-11 encourages programmers to build large, logically connected programs that may well be much bigger than the physical memory of the system. For example, a large computation that had previously been structured in overlays will simply be a big program on VAX-11. But making the virtual memory useful to the application programmer means that the operating system has to be much more active in deciding which pieces of an application program should be kept in central memory and which should be kept on disk-backing storage (called working-set management). This in turn means that the mapping mechanism must be efficiently controlled by operating system code and must provide as much useful information about the dynamic usage of virtual memory as possible.

Sharing virtual memory space

A key feature of the VAX-11 virtual memory design, which leads to efficient use of central memory and low overhead during execution, is the way in which the operating system shares virtual memory space with user processes. The entire 4-gigabyte virtual address space is logically divided into halves. The user process is limited to the use of the bottom half, and most of the operating-system code resides in the top half (Fig. 3). (Remember that this is just virtual memory; the decision about what virtual memory is made resident in central memory, including operating-system code, is made dynamically on the basis of actual need.) The mapping of each user process may be unique, or user processes may share program and data pages with one another, but in either

case the same virtual operating system is mapped with each user process.

Putting the operating system into a single address space rather than having pieces of the system code in multiple address spaces minimizes the need for system mapping control tables and makes intersystem communication more efficient. Having the operating system share the virtual address space of each user process simplifies requesting services from the operating system. The high-speed processor translation caches, which store the most recently used mapping translations, treat system and user mapping separately so that system mapping translations stay in the cache when the operating system switches to another user process, but the user translations are flushed out.

In a simple memory scheme, it would be risky to put the user programs in the same address space as the operating system, since their malfunctions could affect the system operation. In VAX-11, the system is totally protected from this by a separate access control mechanism. The processor executes in one of four modes:

- Kernel, for interrupt processing, physical I/O control, processor scheduling, and the like.
- Executive, for file management and similar functions.
- Supervisor, for functions such as interactive command processing.
- User, in which user programs are executed.

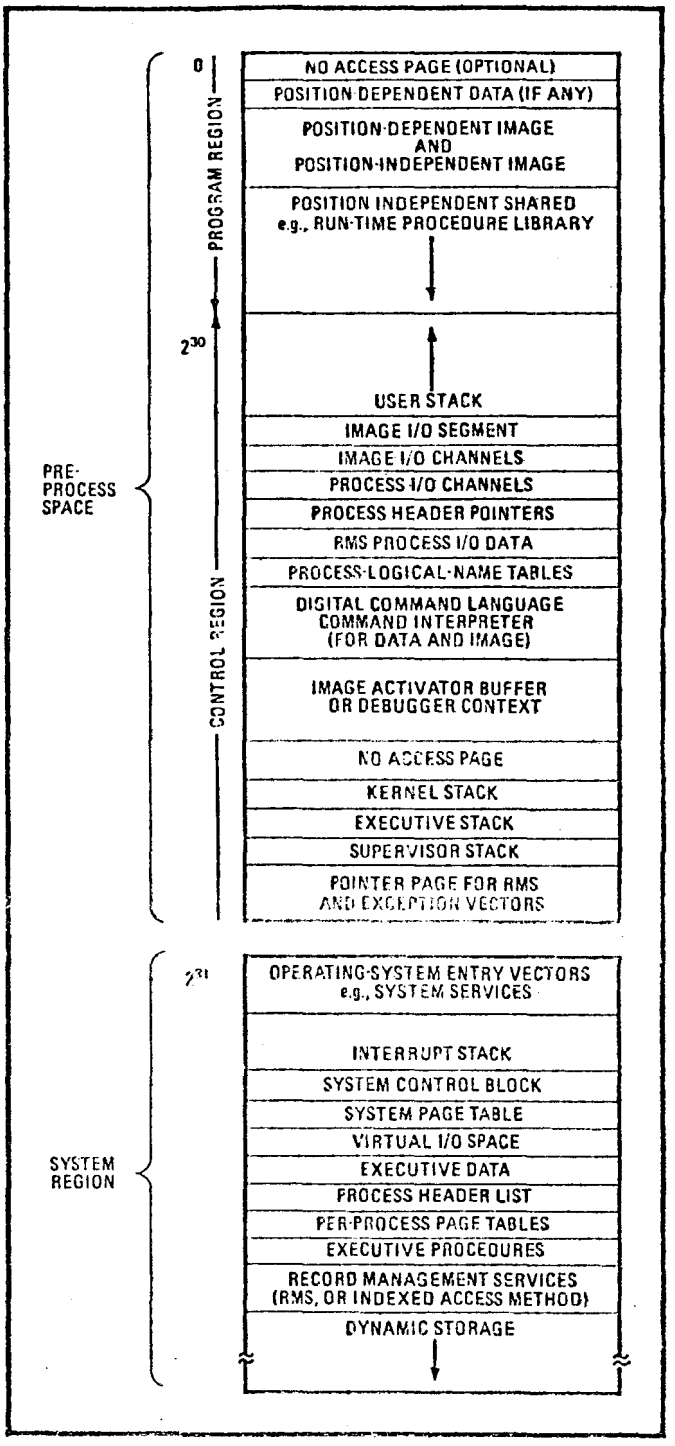
The accessibility of each page in the virtual memory space (whether the page can be read, written, or both) can be controlled for each of the processor execution modes. Thus the operating system can keep critical data in the user's address space and can access that data freely during execution of an operating system service (during which the processor runs in a more privileged mode), and yet the user's program may be restricted from reading the data, if that is inappropriate.

Compatibility mode

The VAX-11 user mode can be put into a compatibility mode, which makes it capable of executing many PDP-11 programs often faster than the PDP-11/70, the top-of-the-line PDP-11. The efficiency of PDP-11 emulation is due to the strong cultural compatibility between VAX-11 and the PDP-11. Thus a processor designed to perform VAX-11 instructions efficiently can also perform PDP-11 instructions well. Instruction execution of the VAX-11/780 is implemented with microcode (as is true of most computers today); PDP-11 compatibility-mode emulation was primarily implemented with a 10% increment of microcode.

This VAX-11/PDP-11 compatibility mode is worth exploring in some detail. Some earlier "compatible" emulation modes required that the computer be used only in one mode at a time. But although 16- and 32-bit code cannot be freely mingled within a single program on the VAX-11, compatibility-mode jobs and native jobs can run at the same time, sharing the resources of the VAX/VMS multiprogramming system. The two kinds of jobs can even cooperate with each other by exchanging messages or by sharing files.

Emulation modes have also been used in the past in the place of software development for the new architecture. However, in the case of VAX-11, the power of the enormous virtual address space and new instructions were intrinsic to the value of the system. So there seemed to be little value in a hardware-supported compatibility mode that would execute a complete PDP-11 operating system. Instead, as already indicated, the compatibility mode is limited to user-mode programs. Those operating-system utilities that are insensitive to the size of the address space have been taken from the earlier family's



3. Allocation. Virtual address space is divided into halves. The lower address locations are limited to user processes, the upper to operating-system code. User processes may share program and data pages, but virtual operating system is always the same.

DATA TYPES HANDLED BY VAX-11

Data type	Size	Range (decimal)	
Integer		Signed	Unsigned
Byte	8 bits	−128 to +127	0 to 255
Word	16 bits	−32,768 to +32,767	0 to 65,535
Long word	32 bits	-2^{31} to $+2^{31}-1$	0 to $2^{32}-1$
Quad word	64 bits	-2^{63} to $+2^{63}-1$	0 to $2^{64}-1$
Floating point		$\pm 2.9 \times 10^{-37}$ to 1.7×10^{38}	
Floating	32 bits	approximately 7-decimal-digit precision	
Double floating	64 bits	approximately 16-decimal-digit precision	
Packed decimal string	0 to 16 bytes (31 digits)	numeric, two digits per byte sign in low half of last byte	
Character string	0 to 65,535 bytes	1 character per byte	
Variable-length bit field	0 to 32 bits	dependent on interpretation	

RSX-11M operating system and execute in compatibility mode transparently to the user.

The application migration executive is a subroutine package provided with VAX/VMS that emulates RSX-11M operating support for PDP-11 programs running in the VAX-11/PDP-11 compatibility mode. VAX/VMS has been designed to transfer control to the AME within 50 microseconds on the VAX-11/780 when a PDP-11 compatibility-mode program requests operating-system services. The AME executes as a VAX-11 program, in a 32-bit address space that includes the 16-bit address space of the PDP-11 program. The AME determines which RSX-11M system call is being requested by a PDP-11 program, translates it into VAX/VMS format, and issues the request to VAX/VMS. When control returns to the AME, it translates the results into RSX-11M format, stores the result in the compatibility-mode program data, and then returns control to the PDP-11 program via a VAX/VMS service.

Translators

Use of the AME permits a large collection of the RSX-11M programs to run unchanged on VAX-11 systems under VAX/VMS. Although the AME translates RSX-11M system calls, a similar program could be written to translate calls of other PDP-11 operating systems. A single VAX/VMS system could, theoretically, have translators for multiple PDP-11 operating systems.

The efficiency of compatibility-mode program execution under such a translator depends on how heavily the program uses operating-system facilities and how different the emulated operating system was from RSX-11M and VAX/VMS. Although the translation adds some overhead, the typical VAX/VMS service is faster than RSX-11M (run on a PDP-11/70) because of the increased functionality of the VAX-11. On balance, emulated PDP-11 programs run about as fast as they would in the PDP-11/70 under RSX-11M.

The PDP-11 was designed with 8 general-purpose registers. Since then, the cost increment of additional processor registers has gone down dramatically, and the VAX-11 was given 16. Apart from cost, the penalties for additional registers are a need for extra system overhead to perform context switching and a reduction in bit-efficiency, since more bits are required to address a register. Nevertheless, these extra registers do provide better compiler optimization of generated code, lower overhead in subroutine usage, and efficient design of complex instructions.

The strength of the PDP-11 architecture is its inclusion of the best features of stack, multiple-register, and memory-to-memory designs because of the versatile way in which its general registers can be used to develop addresses. VAX-11 added to these addressing modes to increase the efficiency of program indexing into those tables that list multiple-byte data items like 4-byte floating-point values or 8-byte integers.

In the last decade the processing capacity of minicomputer systems has increased to the point where they are patently unsuited for very few applications. To support efficiently all likely forms of processing, new data types (forms of data for which processor instructions exist) were added to VAX-11, as shown in the table. VAX-11 implicitly does 32-bit address arithmetic, and instructions were added for explicit 32-bit integer arithmetic and Boolean logic.

Decimal arithmetic

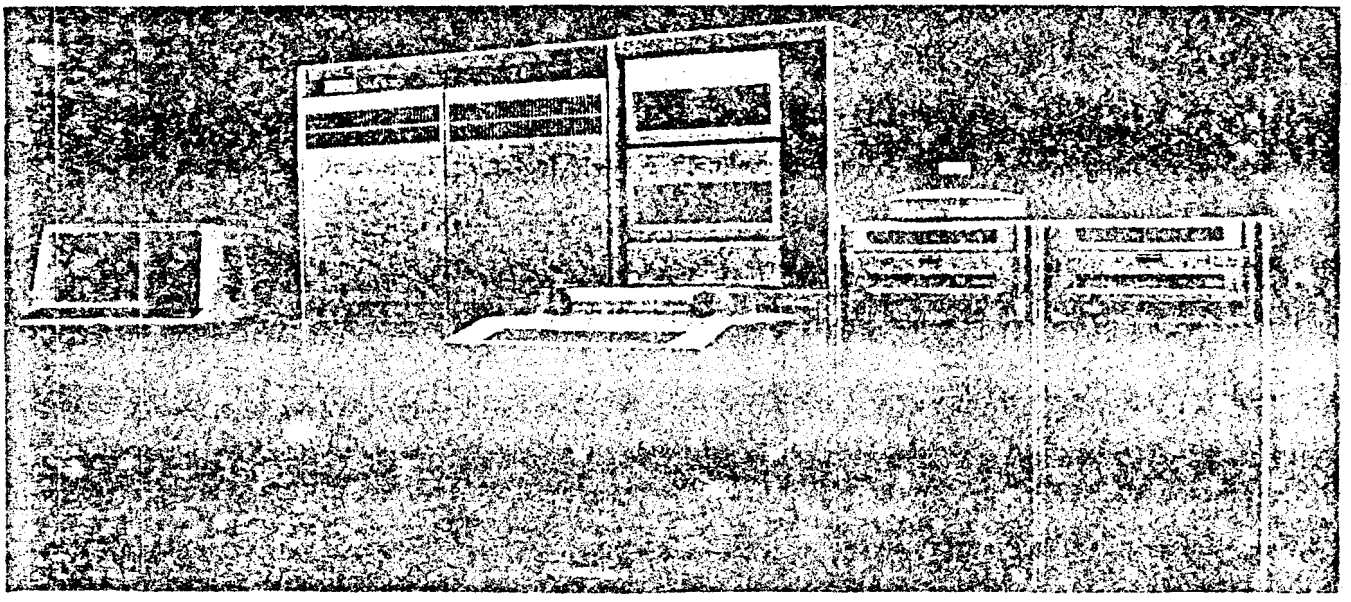
Thirdly, VAX-11 permits arithmetic to be done directly in decimal form, instead of requiring that it be converted to binary form, to ensure that full data precision is retained, and because such data is more often moved intact between data records than used for calculation. Still other new data types are test string manipulation (where strings of characters can be moved, translated, and searched with specific instructions) and a complex editing instruction (to provide for the kinds of manipulations common in generating the typical data-processing report—for example, editing out leading zeros or adding a dollar sign). On VAX-11, direct processor support for 1- to 32-bit data fields has been implemented, increasing the bit efficiency of critical

The economics of compatibility

The last thing a computer user wants to do is to rewrite an existing program for a new hardware design. He would rather develop new software for all the profitable new applications it opens up. After all, if the hardware is costing him less, the cost of programming is still as high as ever. Worse yet, it may even be rising, since the number of computers to be programmed appears to be growing faster than the population of skilled programmers.

In view of this, a major design focus for the VAX-11 architecture was compatibility with the PDP-11. Accurate statistics are not available, but if as little as $20,000 has been spent for software for each of the 50,000 PDP-11s produced this far, then the total investment is $1 billion. Surely, this is a conservative estimate when it is remembered that $20,000 buys only a small amount of code.

The size of the entire existing investment in software is incomparably larger, and the need to preserve it might already have halted innovation in the computer industry were it not for the phenomenal rapidity with which computer technology is evolving. The new markets and applications continuously being opened up force architectural changes that compel some level of innovation even at the price of a devalued software investment. Nevertheless, the point has seemingly been reached at which no new computer—mainframe, mini or micro—can be designed without careful examination of the compatibility issue. Barring some remarkable breakthrough in software engineering, compatibility will continue to grow in importance.



Compatibility. Design goal of the VAX-11/780 32-bit minicomputer system shown here was to preserve compatability with the software developed for the existing PDP-11 family while anticipating future needs. It features greatly extended virtual address space.

operating-system code and like programs. Since field-bit position is specified by a 32-bit integer, very large (512-megabyte) structures can be linearly bit-addressed.

Although most of the added instructions were in support of the data types listed above, many other special instructions were added for operating system support, user programming support, and specific computation needs. For example, an instruction, POLY, has been provided to compute polynomial equations of the form:

$$y = C_1 + C_2x + C_3x^2 + \ldots$$

in a single instruction, with the loop overhead handled in microcode. This instruction substantially speeds up the calculation of standard numerical approximation, such as the calculation of sine and cosine functions within a Fortran run-time library.

Future minicomputers

The implications of all these capabilities for the future of the VAX-11 minicomputer family can readily be assessed. The yardstick may be inferred from the apparently constant rates of improvements in base technologies like semiconductors and magnetics and from computer designers' rules of thumb. For example, in a typical system the number of bytes of central memory is about equal to the number of instructions per second by the central processor. Such a system must also be capable of performing 1 bit of I/O for each instruction executed (these rules are sometimes attributed to Gene Amdahl). For any year in the near future, the technologies' price prediction for computer subsystems may be combined with the designers' predictions of the appearance of a balanced system, the constant price definition of minicomputers applied, and the range of expectable system configurations thus delimited.

Following such logic, a minicomputer priced at $50,000 in the early 1980s should look much like today's mainframe in gross capability, having on the order of a million bytes of central memory, hundreds of millions of bytes of disk storage, and so on. That prediction, made some years ago, led to the VAX-11 design project and the development of the VAX-11/780. □

digital

JUN 18 1979

INTEROFFICE MEMORANDUM

TO: Distribution

DATE: 14 June 1979
FROM: George Hoff
DEPT: LSG Engineering Operations
EXT: 231-6524
LOC/MAIL STOP: MR1-2/E78

SUBJ: VENUS TECHNOLOGY REVIEW - 06 JUNE 1979

Attached is a summary of the issues discussed, positions taken, and decisions made. I have also attached a copy of the slides presented and handouts prepared for the meeting. The minutes are highly condensed in order to get this information distributed in a timely manner. If anyone has corrections or additions please contact me.

GH/dmc
attachments

digital

INTEROFFICE MEMORANDUM

TO: Distribution

DATE: 14 June 1979
FROM: George Hoff
DEPT: LSG Engineering Operations
EXT: 231-6524
LOC/MAIL STOP: MR1-2/E78

SUBJ: SUMMARY OF DISCUSSION AND DATA REVIEWED AT VENUS TECHNOLOGY REVIEW 06 JUNE 1979

Attendees

O^2D:	Gordon Bell, Jim Cudmore, Ulf Fagerquist
LSG:	Sas Durvasula, George Hoff, Vic Ku, Jud Leonard, Pat Sullivan, Bill Walton, Sultan Zia
Microproducts:	Russ Doane
MSD:	Brian Croxon
Purchasing:	Dan Hamel, Steve Kavicchi
O^2T:	Alan Kotok

Goal for Meeting

Review tradeoffs between three technology alternatives for Venus: MCA (Motorola), Siemmens 100K ECL array and 100K MSI (Fairchild).

Tradeoff Review included:

1. Prime Vendor Status and second sources
2. Burdened Part Cost (Cost of each type of IC mounted on tested module)
3. Venus Cost Estimate with each technology
4. Schedule differences
5. Development Cost

Review of Positions

Decision on how to proceed

Review of Data (see slides attached)

The data presented indicated that with a average module cost estimated ($350 for 8 layers) the MCA approach yielded a pro-

cessor kernel cost of $9695 (5% above goal), versus $11592 (20% above goal) for 100K MSI. Siemmens was significantly more expensive than the MCA and provides less functionality (36 cells vs. 48 for the MCA) and was excluded.

The cost difference between 100K MSI and the MCA was noted as not highly significant in the cost of the total system. Pat Sullivan further projected the 100K MSI cost could be brought down to the MCA cost by reducing module cost to $120 and application of 24 pin dips in lieu of chip carriers. Pat was alone in believing that this could be accomplished. Another issue discussed was the value of chip replacement (instead of modules) which Field Service has estimated to be worth $20 million over the life of the product. The MCA with I^2L diagnostic logic and sockets maximizes our chances of reaching this goal.

A review of schedule differences between the 100K MSI approach and the MCA indicated only a slight advantage for 100K MSI (1 month). This was not generally accepted and the consensus appeared to be more like 3-6 months based on Comet experience.

The development cost for Venus with MCA was estimated at $14.3 million vs. $12.9 million for 100K MSI.

Position Summary

The vendor/second source situation is not optimum for any of the technologies. Motorola (MCA) looks better than Fairchild (100K MSI) primarily due to a potential 2nd U.S. source and a history of more stability as a volume vendor. The fact that we will not have a qualified part until October is a major concern. (Hamel) Gordon suggested we exploit this decision process to secure commitments from Motorola at the senior management levels.

The data we have been able to generate to date indicates only a moderate cost advantage (20%) for the MCA, however, to a designer the actual potential looks greater. As the designers learn to use the MCA the cost reduction yield relative to 100K MSI will increase. The MCA is the choice of the designers because it is the most competitive solution with the greatest potential and we have a running start in the CAD tools area. The MCA also reduces the level of module interconnect required which should enhance our chances of volume module build by Digital -- this is critical to Venus. (Durvasula, Hoff, Leonard, Kotok)

Pat Sullivan's position that 100K MSI was a better solution in terms of schedule, risk, and potential cost was not changed as a result of the discussion above. Pat suggested that a Hybrid approach of 100K MSI and Siemens arrays for control might be optimum. Gordon rejected this proposal on the grounds that it resulted in maximum risk; i.e. we would need to develop two volume technologies to get Venus to market.

The potential application of 100K MSI for the 2080 was also discussed. Since the 2080 is less cost sensitive, has lower volume and critical time to market requirement a different technology choice may be appropriate. The 2080 could possibly use multiwire for production relieving the requirement for a fine line multilayer 100K module. A multiwire 100K MSI approach for the 2080 would eliminate conflict for both chip layout and module layout resources between Venus and 2080 in Marlboro.

We need to build up our knowledge in complex design tools and processes. The oxide isolation process is a critical "next step" in our bi-polar process development. We must go forward into advanced technology and drive prices down. (Croxon, Cudmore, Doane)

The real tradeoff is between the short term and long term. All indications are that the future is gate array and beyond. We must stage this product to build the knowledge base for the next product. We can not skip a technology step and expect to make a double jump in the next generation machine. The Comet experience indicates we can expect incremental development cost and longer time to market (3-6 months), however, this is the risk we must take to meet the competition. The Venus schedule and budget must be tested versus what happened on Comet. (Bell, Fagerquist)

Decision

Gordon's position was to proceed with the design using the MCA. O^2D approval and Operations Committee approval must be attained before the decision is final. Gordon will recommend approval of the MCA technology for Venus. Gordon also recommended that we actively pursue putting the Bipolar RAMs also in the sockets as this would yield a substantial savings in the field service replacement costs.

digital

INTEROFFICE MEMORANDUM

TO: Ulf Fagerquist

CC: Distribution

DATE: June 11, 1979
FROM: Bill Green
DEPT: LSI Mfg. & Eng.
EXT: 2220
LOC/MAIL STOP: ML1-4 B34

SUBJ: I.C. Technology for Venus

The long range technology objective of the LSI Group may be summarized in two statements:

1. To develop silicon processes that are close to those of the industry leaders. This requires continued improvements in device density, speed, and power.

2. To develop design processes and tools of a structured (CAD-able) nature that will allow low cost, quick turn around custom chip design.

In addition to supporting these objectives with PL97 and 98 funds, experience dictates that we explore actively opportunities to integrate these efforts with product programs. In such programs the reality of the market place more effectively refines our efforts than is possible elsewhere. This memo will relate the technology objectives above to the choices you are making with respect to Venus and to a lesser extent 20/80.

The alternatives you are examining for the Venus CPU are:

1. 100K MSI
2. Siemens gate array + MSI
3. Motorola gate array + MSI.

Alternative 1 does not in any way support the technology objectives stated above. Furthermore, it does not leave any residuals — insofar as we can judge — for future machines. We believe, in fact, that it merely delays an inevitable move to LSI and probably makes that step steeper when finally taken. If either alternative 2 or 3 produces an equivalent system, they are much more supportive of our objectives.

Both alternatives 2 and 3 support the LSI technology objectives. With regard to silicon processes, both the Motorola and Siemens processes represent a substantial advance over the Comet process. A detailed comparison from available sources shows the two to be literally identical step by step except for the resistivety of the starting material. Thus the work already expended on the Motorola process should contribute equally well to progress on the Siemens process. Until we know more details, it is not equally clear whether the equipment ordered for the oxide isolation step is also useful. Given the fact that both will take DEC to the same end point of performance, the maturity of the Siemens process recommends it.

To illustrate the advantages for future DEC products accruing from the acquisition of the process, a simple comparison is made below between the Comet array and a proposed Siemens TTL array.

	Comet	Siemens
Gates	480	700+
Pins	48	64
Power	1.6-2.0 W	1.5 W
Speed	3-7 ns.	1 ns.

Such an array might be useful for a next generation Comet.

With regard to CAD process technology, both gate arrays will lead to useful progress. The efforts already expended on the Motorola array have been substantial and are essentially complete. No such work has been done in DEC on CAD for the Siemens array. This will require additional effort to achieve an equivalent posture.

A table is attached to compare the technology for business arrangements possible with Siemens and Motorola. The Motorola agreement is known in detail. The Siemens proposal, while remarkably complete on short notice, is yet to be negotiated in detail. Note that though the Siemens agreement calls for a royalty while the Motorola one does not, there is some equivalent offset since we believe we experience some small price increase on guaranteed business. From a business point of view neither the cost nor obligations are sufficiently different to discriminate between the two.

Summary and Recommendation

The execution of Venus in 100K MSI does not enhance the DEC technology position in LSI nor leave any residuals toward future systems. Either gate array proposal supports both LSI silicon and design process objectives and builds residuals toward future systems. While both silicon processes are in principle identical, the Siemens process is substantially more mature and proven by a reasonable amount of production. As an offset, the CAD to support logic design is more advanced with Motorola. The business positions with both vendors are nearly equivalent with Motorola representing a broader potential as a partner. In summary, the LSI group finds little sharp distinction between the support for its programs between Motorola and Siemens and will support equally either choice.

WBG:cg
Attachment (1)
Distribution:

CC: Gordon Bell
Pat Buffet
Jim Cudmore
Dan Hamel
George Hoff ✓
Ruth Rawa
Rod Schmidt
Jack Schneider
Bobby Snow
Joe Zeh

Attachment 1.

	Motorola	Siemens
Provides gate array design	yes	yes
Provides ECL process	yes	yes
Provides technical consultation	yes	yes
Gate array business	FY81 70% FY82 50% FY83 50%	about 50%
Cash payment	$150,000	about $100,000
Royalty	no	about 3% on I.C.'s produced by DEC
Standard Device Business	$5M	no
Unibus license	yes	no
Other	preferential qualification on new standard device	unknown
Provides CAD programs	no	yes
ECL RAM business	40% FY81-83	not clear

TECHNOLOGY STATUS REVIEW

1) MCA: TWO U.S. SOURCES
   - (a) MOTOROLA (b) NATIONAL SEMI
   - (a) MOTOROLA'S WORKING PARTS - SEPT 79
     COMITTED TO VOLUME - SEPT 80
   - (b) NATIONAL SEMI — NO DATA YET

2) SIEMENS ARRAY: TWO EUROPEAN SOURCES
   - a) SIEMENS
   - b) RTC
   - (a) SIEMENS PART IS HERE IN QUANTITES
   - (b) RTC IS ONE YEAR AWAY FROM DELIVERING OPTIONS TO SIEMENS.

3) 100K SSI, MSI: TWO U.S. SOURCES
   ONE EUROPEAN SOURCE
   - a) FAIRCHILD HAS 35 PART TYPES
   - b) NATIONAL WILL HAVE 17 TYPES IN TWO YEARS
   - c) RTC WILL HAVE 27 TYPES IN TWO YEARS.

——— x ———

	MCA	SIEMENS 36 ARRAY	100K Logic A.S.P
DEVICE COST	$ 33.00	$ 42.00	2.05

ASSUMPTIONS IN THE COSTING OF THE MODULE

FOR MCA & GATE ARRAY :

1. 8 LAYER (4 SIGNAL) EXTENDED HEX.
2. 32 MCAs OR GATE ARRAYS / BOARD
3. DEVICES SOCKETED
4. CHIP LEVEL ISOLATION
5. ASSOCIATED TERMINATORS & CAPACITORS

FOR 100K LOGIC FAMILY :

1. 8 LAYER (4 SIGNAL) EXTENDED HEX
2. 162 DEVICES IN CHIP CARRIERS
3. SURFACE MOUNTED COMPONENTS WITH REFLOW SOLDER PROCESS
4. BOARD LEVEL ISOLATION
5. ASSOCIATED TERMINATORS & CAPACITORS.

PRODUCT COST COMPARISON IS DONE BY:

1) CALCULATING THE COST OF A FULLY POPULATED MODULE FOR EACH OF THE THREE TECHNOLOGIES.

2) CALCULATE THE COST OF A FULLY LOADED DEVICE BY DIVIDING THE COST OF THE MODULE BY THE NUMBER OF DEVICES

3) CALCULATE THE PRODUCT COST BY MULTIPLYING THE LOADED COST / DEVICE BY THE NUMBER OF DEVICES IN THE CPU.

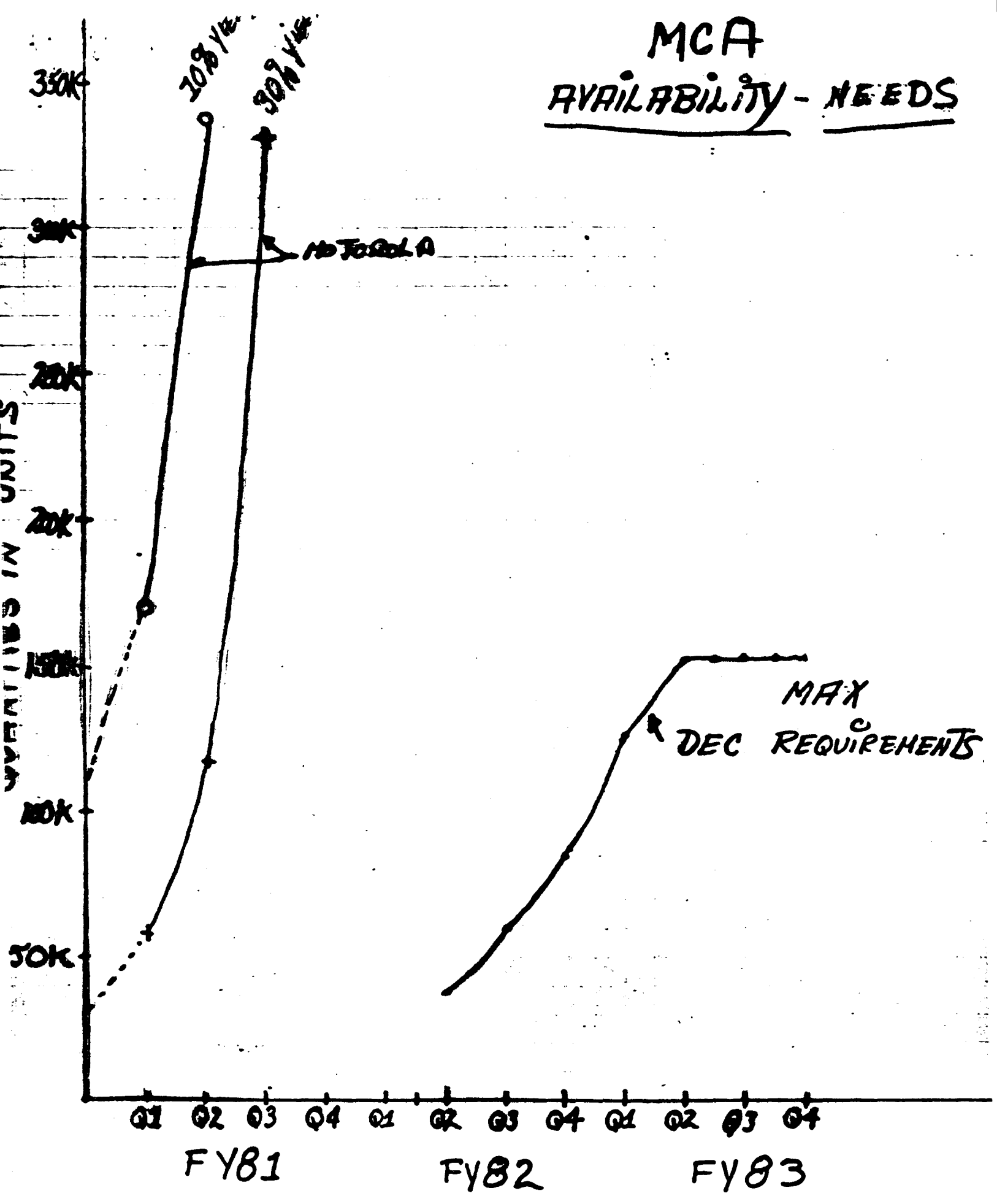
LOADED COST/DEVICE ANALYSIS

	NOT	SIE-24	SIE-36	100K IN CHIP CARRIER	100K IN 24 PIN DIP*
P.C. BD. (8 LAYER)	350	350	350	350	120*
SOCKET	96	96	96	———	----
CAPS	25.7	25.7	25.7	32	32
RESISTORS	24	24	24	32	47
ASSEMBLY/TEST	70	82	82	160	97
DIAGNOSTIC LOGIC	-----	176	176	75	{ 349
I.C.'s	1056	1088.	1344.	328	{ ---
INCOMING INSPECTION	106	109	134	32.8	35
TOTAL $	1727.7	$1950.7	$2231.7	$1010.00	$680.*
COST/DEVICE	54.0	61.0	69.74	6.23	4.53
COST OF POWER/DEVICE	5.0	3.0	4.0	.80	.80
TOTAL COST/DEVICE =	59.0	64.0	73.7	7.0	5.3

ASSUMPTIONS:

1. 32 MCAs PER EXTENDED HEX
2. 32 SIEMENS ARRAY PER EXTENDED HEX
3. 162 SSI, MSI PER EXTENDED HEX
4. 8 LAYER (4 SIGNAL LAYERS) P.C. BOARD.
5.* COST BASED ON POSSIBILITY OF MOUNTING 171 24 PIN DIPS ON 6 LAYER EXTENDED HEX.

MCA
AVAILABILITY - NEEDS



DATA PATH TRIAL DESIGNS

	MCA	SIEMENS	100K	RATIO MSI/GATE ARRAY MCA	SIEMENS
VENUS IBOX DATA PATH					
CHIP COUNT	16 (13)	20	130	8.1 (10)	12
DELAY (NS)	50 (57)	49	58	1.16(1.02)	1.18
VENUS EBOX DATA PATH					
CHIP COUNT	13		80	6.2	
DELAY	50		57	1.14	
VENUS MBOX DATA PATH					
CHIP COUNT	8	8	90	11.3	11.3
DELAY	30		29	.96	
DOLPHIN MBOX DATA PATH					
CHIP COUNT	17	26+7msi	505	29.7	18.7
DELAY	76	60			

CONTROL TRIAL DESIGNS

MULTI-MEM CONTROL					
CHIP COUNT	.4	.6	11	27	18
DELAY	24	22	25	1.04	1.14
KL10 MB CONTROL					
CHIP COUNT	.5	1.0	19	38	19
MICROSTACK CONTROL					
CHIP COUNT	.26		5.4	21	
IBOX VALID CONTROL					
CHIP COUNT	.5		16	32	
DOLPHIN BUS INTERFACE					
CHIP COUNT	1		17	17	

WEIGHTED MEAN FUNCTIONALITY RATIOS

DATA PATH: 1x MCA = 8.1x 100K MSI (excluding Dolphin MBOX)
1x MCA = 1.3x SIEMENS

CONTROL: 1x MCA = 26x 100K MSI
1x MCA = 1.8x SIEMENS

CPU DESIGNERS POLL

"WHAT PROPORTION OF CHIPS IN A TYPICAL CPU ARE USED FOR DATA PATH AS OPPOSED TO CONTROL?"

PAUL BINDER: 1/3 TO 1/2 IS CONTROL

JEFF MITCHELL: ABOUT HALF CONTROL

ALAN KOTOK: ABOUT 50 - 50

KL10 MODULES:	30% RAM
	37% CONTROL
	33% DATA PATH
780 MODULES:	22% RAM/PROM
	44% CONTROL
	33% DATA PATH

IN ORDER TO COMPARE COSTS, WE WANT
THE OVERALL RATIO (R) OF MSI TO GATE
ARRAY CHIPS TO IMPLEMENT EQUAL FUNCTION

LET F_c = FRACTION OF MSI CHIPS NEEDED FOR CONTROL

$F_d = 1 - F_c$ = FRACTION OF MSI CHIPS FOR DATA PATH

R_c = RATIO OF CONTROL MSI PER GATE ARRAY

R_d = RATIO OF DATA PATH MSI PER GATE ARRAY

THEN

$$\frac{F_c}{R_c} + \frac{1-F_d}{R_d} = \frac{1}{R}$$

$$R = \frac{R_c R_d}{R_d F_c + R_c(1-F_c)}$$

EXAMPLE: GIVEN A 1000-CHIP MSI CPU, HALF CONTROL AND HALF DATA PATH, HOW MANY MCA'S ARE REQUIRED TO REPLACE IT, AT A RATIO OF 8.1 FOR DATA PATH AND 26 FOR CONTROL?

DATA PATH MCA'S = 500 / 8.1 = 62

CONTROL MCA'S = 500 / 26 = 19

TOTAL MCA'S 81

OVERALL RATIO (R) = 1000 / 81 = 12.3

LOGIC COST COMPARISON: GOAL = $9200

MCA IMPLEMENTATION
USING MARCH 10 ESTIMATES

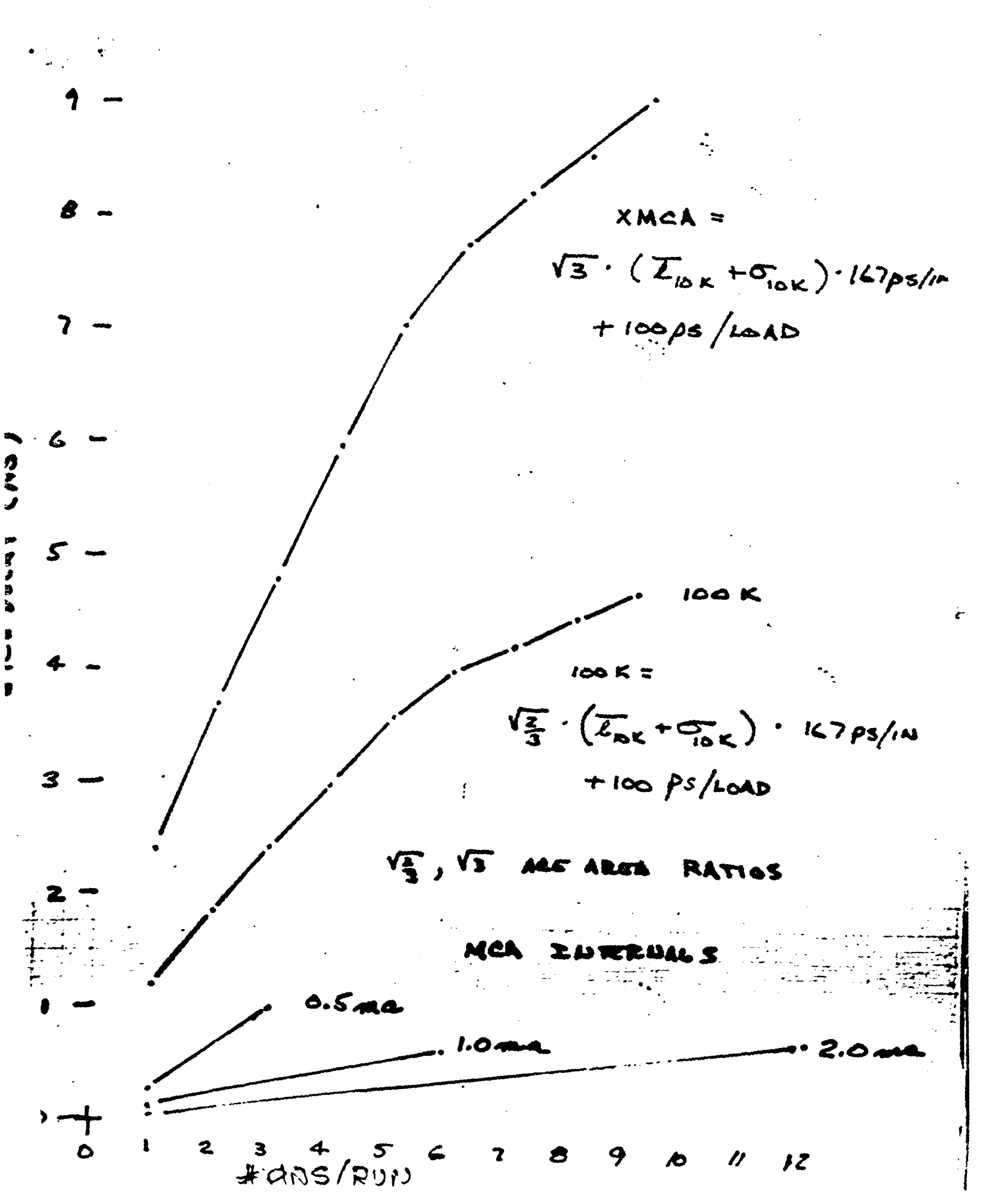
70 MCA @ $59	$ 4130	
235 4K RAM @ $19	4465	BURDENED
100 1K RAM @ $11	1100	$5/CHIP
	$ 9695	

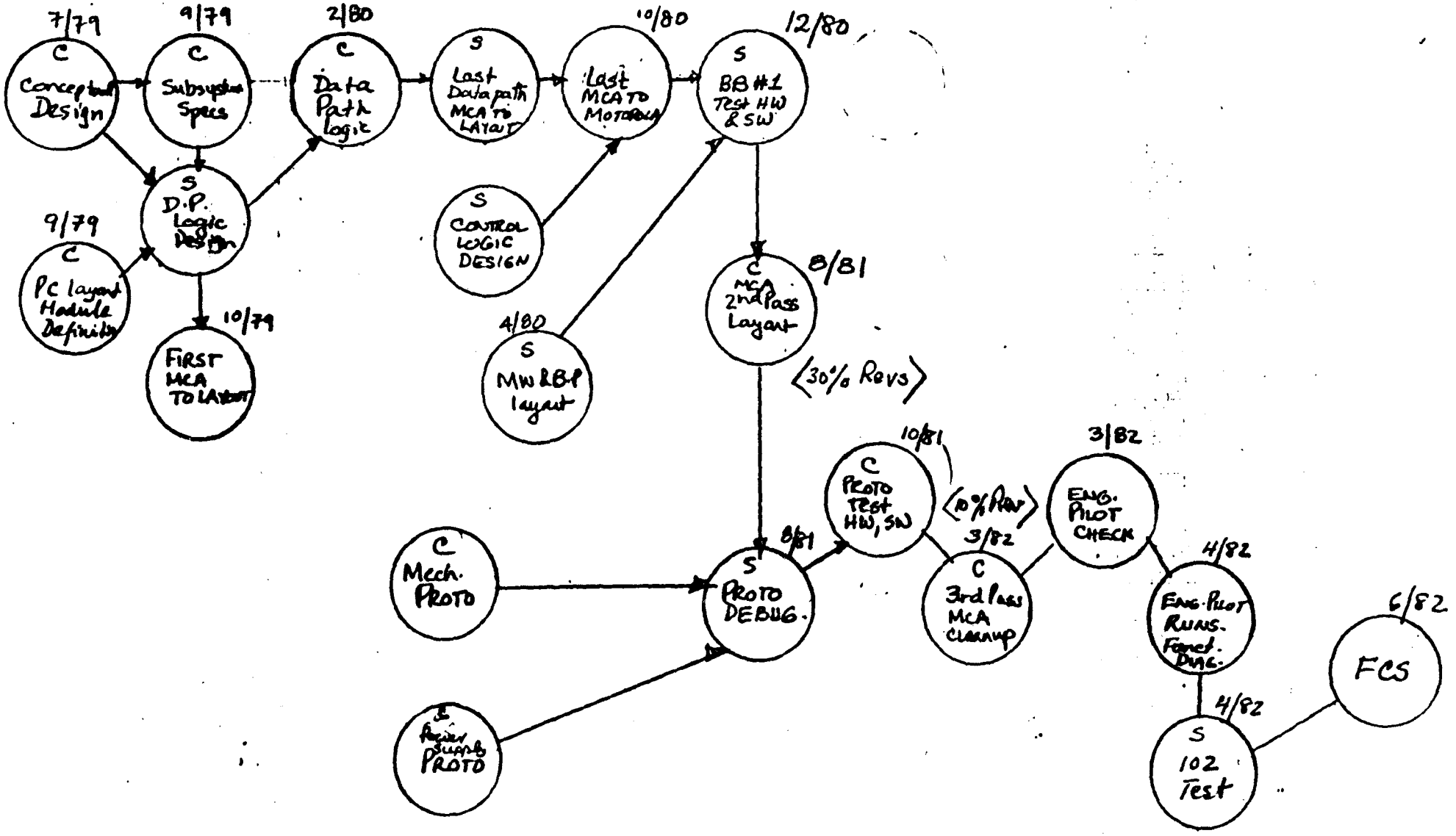
9 bds

100K EQUIVALENT IMPLEMENTATION
@ FUNCTIONALITY RATIO..= 12.3

861 MSI @ $7	$ 6027
TOTAL RAMS	5565
	$ 11592

11 bds


XMCA =
$\sqrt{3} \cdot (\bar{t}_{10K} + \sigma_{10K}) \cdot 167 ps/in$
+ 100 ps / LOAD
100 K
100 K =
$\sqrt{\frac{2}{3}} \cdot (\bar{t}_{10K} + \sigma_{10K}) \cdot 167 ps/in$
+ 100 ps/LOAD
$\sqrt{\frac{2}{3}}$, $\sqrt{3}$ MC AREA RATIOS
MCA INTERVALS
0.5 ma
1.0 ma
2.0 ma
#ONS/RUN

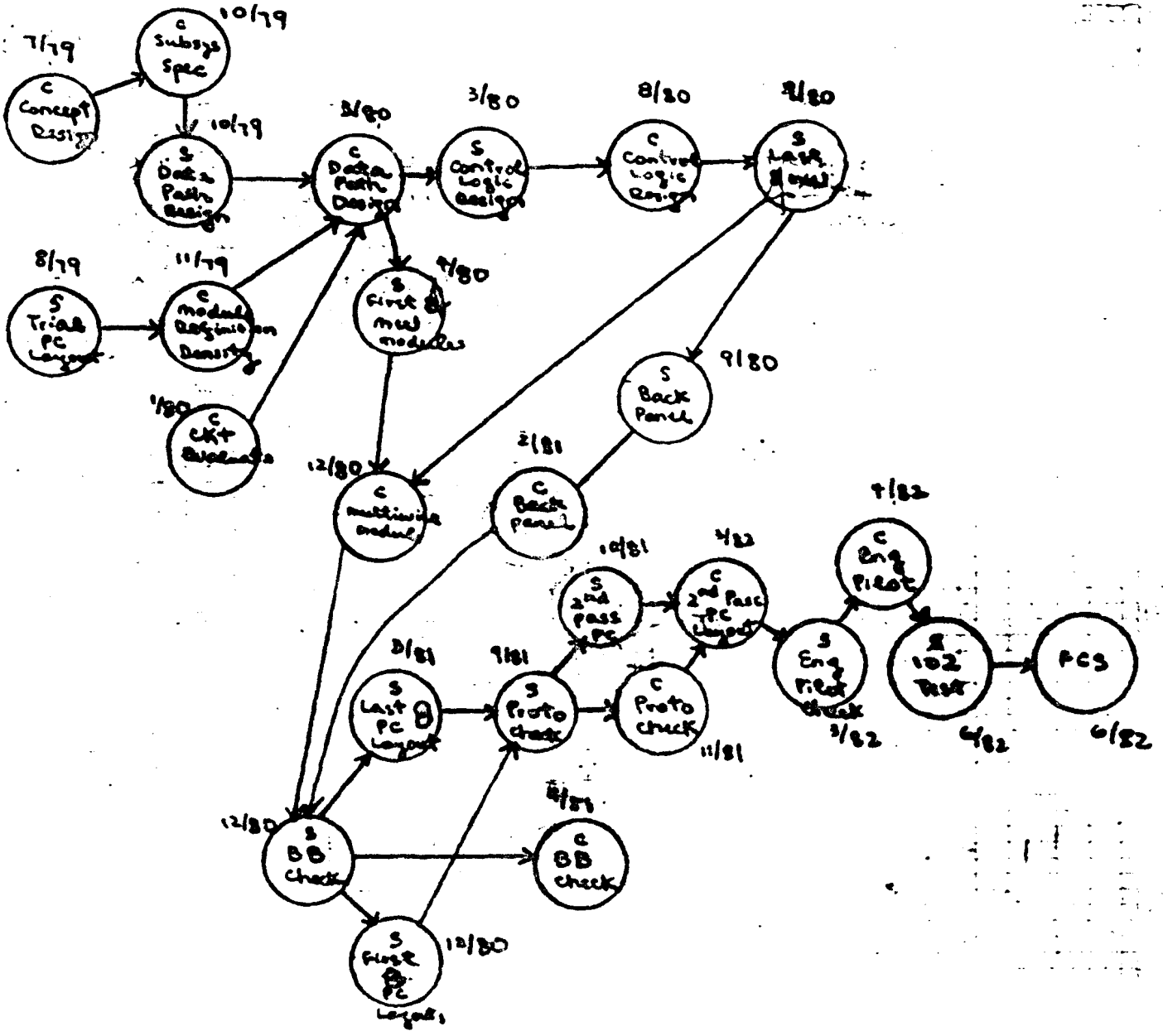


MCA BASED VENUS DEV. PERT TO FCS

MCA VENUS PERT

ASSUMPTIONS:

1. MCA, 10K Technology
2. Do all MCA design for breadboard with emulator backup for failing MCAs.
3. Maximum of 40 MCA types to do the design.
4. FY80, 81, 82 loading 10 Engineers and 7 Technicians for MCA design.
5. Start first MCA layout by 15 OCT, 4 pipes in layout.
6. 6 weeks layout time, 4 weeks for calma operation, 12 weeks in MOTOROLA for chips.
7. Breadboard is multiwire and proto is etch.
8. 3 passes to get MCAs right for FCS.
9. No schedule improvement in MOTOROLA during the first two years.
10. MCA development cost 80K/MCA.



Uses Perk using 100K

Vic Kn
4/12/79

Assumptions =

1) 100K SSI and MSI chips

2) Total chip count = 11/780 = 2087 chips (excluding RAMS and ROMS)

3) 160 chips/module, 4 signal layer/module

4) Total CPU module count = 16

5) 20 weeks per layout, 6 IC designers

6) BBs are multiwire, Prototypes, Pilots, FCS Systems are etched

Vic Ku
6/12/79

VENUS PROJECT MAJOR MILESTONES

	with MCA	with 100K	
Conceptual Design Done	7/79	7/79	
Specifications Available	9/78	9/79	
Complete Data Path Design	2/80	3/80	
Complete Control Path Design	5/80	5/80	
BB Power On	12/80	9/80	
Complete BB Checkout	4/81	12/80	
Proto Power On	8/81	6/81	
Complete Proto Checkout	10/81	8/81	
Eng. Pilot Power On	2/82	12/81	
	3/82	1/82	
DMT, 102 Complete	5/82	3/82	
FCS (50%)	5/82	3/82	(1)
FCS (90%)	3/83	9/82	
Volume FCS (50%)	11/82	9/82	

(1) If PC layout time is 10 weeks instead of 20 weeks, this date will be one quarter sooner.

Vic Ku
6/12/79

VENUS HARDWARE DEVELOPMENT COST (K) - MCA

	FY80	FY81	FY82	FY83	Total (K)
CPU	1600	2600	2200	700	7100
Memory	327	600	300	70	1297
Technology	1016	1050	900	200	3166
MCA/tools	500	50	30	--	580
Release Eng.	46	58	200	100	404
IPA/HSC50	167	240	70	50	527
Hydra Comm/UR	40	72	50	50	212
SBI Adaptor	143	263	70	48	524
CAD	103	120	140	100	463
				Total	14.3M

Vic Ku
6/12/79

VENUS HARDWARE DEVELOPMENT COST (K) - 100 K

	FY80	FY81	FY82	FY83	Total (K)
CPU Design	1500	2100	1700	700	6000
Memory	327	600	300	70	1297
Technology	1200	1100	1000	200	3500
Release	46	58	200	100	404
IPA/HSC50	167	240	70	50	527
Hydra Comm/UR	40	72	50	50	212
SBI Adaptor	143	263	70	48	524
CAD	103	120	140	100	463
				Total	12.9M

Vic Ku
6/12/79

DECISION MATRIX

CRITERIA	MOTOROLA			SIEMENS			FSC/100K		
	SCORE	WEIGHTING	TOTAL SCORE	SCORE	WEIGHTING	TOTAL SCORE	SCORE	WEIGHTING	TOTAL SCORE
1. SYSTEM COST	10	2	20	6	2	12	9	2	16
2. A. DESIGN RISK ARRAY	7	3	21	10	3	30	10	3	30
B. SCHEDULE RISK DUE TO INTERNAL PROBLEMS	8	3	24	6	3	18	10	3	30
C. SCHEDULE RISK EXTERNAL	5	3	15	9	3	27	9	3	27
3. SECOND SOURCE	8	1	8	6	1	6	8	1	8
4. MANUFACTURABILITY									
A. YIELD	5	1	5	7	1	7	9	1	9
B. CAPACITY	8	1	8	8	1	8	9	1	9
5. SUPPORT CIRCUITS	3	1	3	10	1	10	10	1	10
6. EASE OF ENG. INTERACTION	9	1	9	5	1	5	5	1	5
7. DESIRE TO SELL	10	1	10	7	1	7	8	1	8
8. RELATIONSHIP W/DEC	10	1	10	5	1	5	4	1	4

DECISION MATRIX (CONT'D)

CRITERIA	MOTOROLA			SIEMENS			FSC/100K		
	SCORE	WEIGHTING	TOTAL SCORE	SCORE	WEIGHTING	TOTAL SCORE	SCORE	WEIGHTING	TOTAL SCORE
9. LONG TERM BUSINESS DIRECTION	10	1	10	10	1	10	5	1	5
10. EASE OF FUTURE ECL DESIGN BUSINESS	10	1	10	7	1	7	0	0	0
11. TECHNOLOGY TRANSFER	10	1	10	8	1	8	~~0~~	1	~~0~~ N/A
12. DEC PENETRATION* (LOW IS GOOD)	8	1	8	5	1	5	6	1	6
13. CONTRACTUAL EASE	10	1	10	6	1	6	6	1	6
TOTAL SCORE			181			171			172 ~~182~~

30% OF MOTOROLA G/A PRODUCTION
50% OF SIEMENS' G/A PRODUCTION
35% OF FSC'S 100K PRODUCTION
W/ 100K ONLY

DAN HAMEL
11 JUNE 79

+---------------+
! d i g i t a l ! I N T E R O F F I C E M E M O R A N D U M
+---------------+

DATE: May 2, 1979
REV 1: JUNE 13, 1979
FROM: JOHN HACKENBERG
DEPT: I.C.E.G.
EXT: 6106
LOC: MR1-2/E47

TO: Distribution list

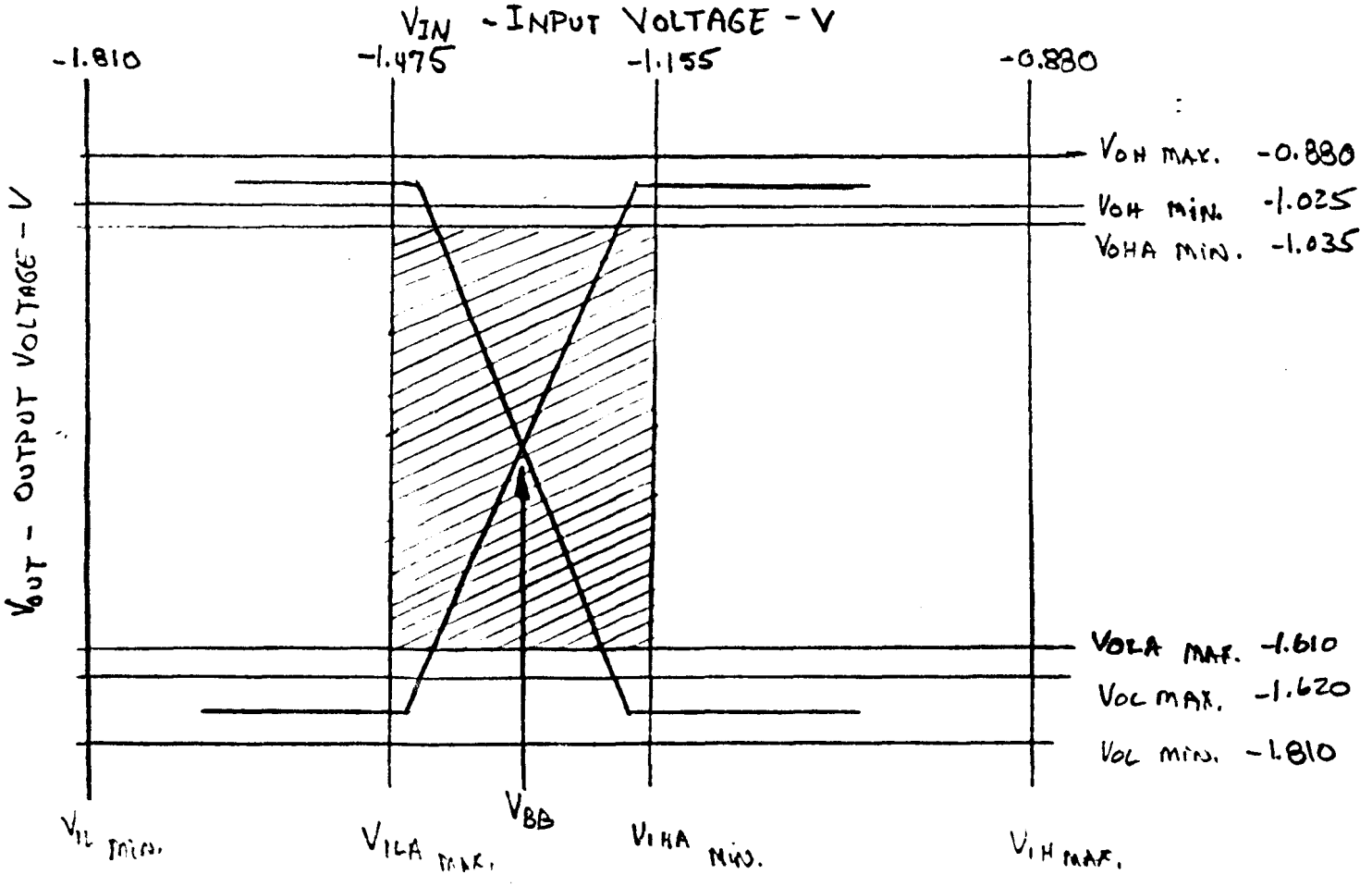
SUBJ: Noise Margin for 100K ECI

This memo shows the noise margin when using 100K and/or 10K ECI parts. This study takes into account voltage and temperature differentials that may exist in a ECI system.

The Siemens Gate Array will incorporate voltage compensation and temperature compensation.

The following noise margin study is based on the 100K ECI noise marging information supplied in the Fairchild ECI Data Bookpublished in 1977. This memo will follow the same outline as the memo of Nov. 2, 1978 on " System Noise Margin for the Dolphin ECI logic".

The following graph shows the points for determining noise margin for the 100K ECI gates:



Guaranteed noise margin (NM) is defined as follows:

(1) $NM_{\text{high level}} = V_{OHA\ min} - V_{IHA\ min}$

(2) $NM_{\text{low level}} = V_{ILA\ max} - V_{OLA\ max}$

The above equations do NOT take into account temperature and voltage varitions in a system. For 100k ECL parts, the tracking rates for the supply voltages are:

$\Delta V_{OHV} = 35$ mV/V, $\Delta V_{BBV} = 52$ mV/V, and $\Delta V_{OLV} = 70$ mV/V

ΔV_{OHV} = The change in V_{OH} due to a change in V_{EE}.

ΔV_{OLV} = The change in V_{OL} due to a change in V_{EE}.

ΔV_{BBV} = The change in V_{BB} due to a change in V_{EE}.

Temperature:

The 100k series operates in both still or forced air systems where an ambient temperature of 0 °C to +85°C is maintained. The required cooling is determined solely by this ambient temperature requirement. There is no need to maintain a constant temperature throughout the system. The 100k devices are relatively insensitive to variations in junction temperature. No power warm-up or moving air cooling is required to assure the specified device characteristics.

The requied system voltage at Vee pins is -4.2 to -5.7 Volts. With a load on all outputs of 50 ohms to -2.0 V and a 0 °C to a +85 °C temperature range, the device dc parameter are guaranteed for a nominal Vee of -4.5 V.

For 100K ECL,

$NM_H = 120$ mV and $NM_L = 135$ mV

These numbers are over the temperature range 0 °C to +85 °C. If there are supply voltage differentials between packages, the above noise margin numbers change.

The magnitude of the output levels and the bias level is larger as the magnitude of the supply voltage increases. The effects of temperature changes are specified in the data sheets. As the temperature increases, the magnitude of the input and output levels get smaller. From these facts, equation (1) and (2) are modified as follows:

(3) $NM_H = /V_{IHA\ min}$ at smallest magnitude of V_{EE} and the highest temp/

$-/V_{OHA\ min}$ at largest magnitude of V_{EE} and the lowest temp/

(4) $NM_L = /V_{OLA\ max}$ at smallest magnitude of V_{EE} and the highest temp/

$-/V_{ILA\ max}$ at the largest mag. of V_{EE} and the lowest temp/

Equations (3) and (4) may be rewritten in terms of temperature and voltage.

(5) $NM_H = /V_{IHA\ min} / -[(\Delta V_{BBV})(\Delta V_L) + (\Delta V_{IHAT})(T_H - 25^{\circ}C)]$

$-/V_{OHA\ min} / -[(\Delta V_{OHV})(\Delta V_H) + (\Delta V_{OHT})(25^{\circ}C - T_L)]$

(6) $NM_L = /V_{OLA\ max} / -[(\Delta V_{OLV})(\Delta V_L) + (\Delta V_{OLT})(T_H - 25^{\circ}C)]$

$-/V_{ILA\ max} / -[(\Delta V_{BBV})(\Delta V_H) + (\Delta V_{ILAT})(25^{\circ}C - T_I)]$

T_I = Ambient temperature of incoming air to system

T_H = Temperature of air exiting system.

$T = T_H - T_I$ = max. temp differential between packages.

V_H = The largest magnitude of V_{EE}.

V_I = The smallest magnitude of V_{EE}.

$V_H = V_H - 5.2$ (or -4.5 V)

$V_I = 5.2$ (or -4.5 V) $- V_I$

NM_H = High voltage noise margin in mV.

NM_I = low voltage noise margin in mV.

The tracking rates can be inserted into equations (5) and (6) to obtain noise margin equations for for 100K ECI. Neglect Temperature variations in 100K ECI logic.

1) 100K ECI driving 100K ECI

$$NM_H = 120 - 52(\Delta V_L) - 35(\Delta V_H)$$

$$NM_l = 135 - 70(\Delta V_l) - 52(\Delta V_H)$$

The following is a chart of Noise margin (in mV) for different supply voltages:

Voltage Tolerance		NM_H	NM_l
-4.5	+/- 7%	97.56mV	101.9mV
-4.5	+/- 5%	100.4	107.55
-4.5	+/- 3%	108.2	118.5
-5.2	+/- 7%	88.3	90.6
-5.2	+/- 5%	97.4	103.3
-5.2	+/- 3%	106.4	115.97

The following is a comparison between 10K ECI non-compensated, 10K ECI compensated, and 100K.

	10K ECI Non-Compensated	10K ECI Compensated	100K ECI
Logic levels			
Voh max.	-0.810 V	-0.810 V	-0.880 V
Voh min.	-0.960 V	-0.960 V	-1.025 V
Voha min.	-0.980 V	-0.960 V	-1.035 V
Vola max.	-1.630 V	-1.630 V	-1.610 V
Vol max.	-1.650 V	-1.650 V	-1.620 V
Vol min.	-1.850 V	-1.850 V	-1.810 V
Vih max.	-0.810 V	-0.810 V	-0.880 V
Viha min.	-1.105 V	-1.105 V	-1.155 V
Vila max.	-1.475 V	-1.475 V	-1.475 V
Vil min.	-1.850 V	-1.850 V	-1.810 V
Noise Margin			
NM h	125 mV	125 mV	120 mV
NM l	155 mV	155 mV	135 mV
NOTE: Voltage differentials and temperature tracking are NOT taken into consideration here.			
Noise Margin			
-4.5 V 7%			
NM h	-	-	92.56 mV
NM l	-	-	101.9 mV
-4.5 V 5%			
NM h	-	-	103.1
NM l	-	-	111.4
-4.5 V 3%			
NM h	-	-	108.2
NM l	-	-	118.5

CONTINUED FROM PAGE 5

	10K ECL Non-Compensated		10K ECL Compensated		100K ECL
-5.2 V 7%	Δ10°C	Δ20°C	Δ10°C	Δ20°C	
NM h	53.3 mV	-1.9 mV	-	-	88.3 mV
NM l	41.3	-13.9	-	-	90.59
-5.2V 5%					
NM h	70	58	101 mV	89 mV	97.4
NM l	46	40	127	121	103.3
-5.2V 3%					
NM h	87.4	75.4	105.8	93.8	106.4
NM l	87.1	81.3	135.2	129.3	115.97
-5.2V 2%					
NM h	96	84	107	95	-
NM l	108	102	138	132	-
logic swing Voh max. - Vol min.	1040 mV		1040 mV		930 mV
Voh min. - Vol max.	690 mV		690 mV		595 mV
Rise Time 20% to 80%	1.1 NS		1.0 NS		0.5 NS
0% to 100%	1.84 NS		1.67 NS		0.84 NS

Summary of Noise margin study:

The following is the noise margin for each ECL type operated at it's nominal operating voltage +/- 3% for power distribution, power supply regulation and filtering on modules.

Nominal Operating Voltages	10K ECL Non-compensated	10K ECL Compensated	100K
+/- 3%	87.1 mV (Δ10°C)	105.8 mV (Δ10°C)	108.2 mV

DISTRIBUTION LIST:

GORDON BELL	ML12-1/A11
RON BINGHAM	MR1-2/E85
STEVE CAVICCHI	WB
DICK CLAYTON	ML12-2/E71
BRIAN CROXON	TW/CC4
JIM CUDMORE	ML1-5/E30
BILL DEMMER	TW/D19
RUSS DOANE	ML1-4/E34
SAS DURVASALA	MR1-2/E47
ULF FAGERQUIST	MR1-2/E78
BILL GREEN	ML1-4/B34
DAN HAMEL	WB
GEORGE HOFF	MR1-2/E78
BILL JOHNSON	ML12-3/A62
ALAN KOTOK	ML3-5/H33
VIC KU	MR1-2/E47
JUD LEONARD	MR1-2/E47
JOHN MEYER	ML12-1/A11
LARRY PORTNER	ML12-1/T32
GRANT SAVIERS	MR3-6/E94
BILL WALTON	MR1-2/E47
SULTAN ZIA	MR1-2/E47

VENUS PRODUCT REQUIREMENTS

MAY 1979

AUTHOR: Marilyn S. Ressler
TW/A08
247-2421

DIGITAL EQUIPMENT CORPORATION

TABLE OF CONTENTS PAGE

ACKNOWLEDGMENT

This VENUS Product Requirements Document is based extensively on a November 1978 preliminary draft written by Peter Conklin and Bernie Lacroute. For this update of VENUS requirements, much-needed technical and marketing information and several good ideas were contributed by Peter Conklin, Bernie Lacroute, Dave Rodgers, Steve Jenkins, Ed Slaughter, Kathryn Norris, Ed McHugh, Al Avery, and Al Ryder.

Similar product requirements documents created in the Mid-range Systems Product Management group by Kathryn Norris (VAX/VMS R2.0) and Lou Philippon (NEBULA) were used as guides for the structure and content of this VENUS document.

The Product Line Marketing section of the VAX-11/780 Sales Guide was the source of details included here on coverage of the market segments.

The MSD Red Book for FY79 provided information on corporate and MSD strategy and on the VENUS product relative to other 32-bit products under development.

This entire document was prepared on a WS102 word processing system by Carol Hicks, Mid-range Systems Product Management.

PREFACE

This is a Product Requirements document for VENUS, the VAX-11/780 replacement product. Several other documents on the VENUS product will be published. Among these are:

1. Preliminary Product Summary (PPS)/Product Contract
2. Project Plans
3. System Plan
4. Product Description and Specification
5. Business Plan

The objectives of this Product Requirements document are:

1. to specify the attributes of the VENUS product;
2. to identify the marketplace for VENUS;
3. to describe how the VENUS product, its marketplace, and its product strategy are compatible with the corporate strategy and objectives;
4. to provide product and market information to every DIGITAL group involved in the VENUS development and marketing;
5. to establish a precedent for creation and subsequent use of this document and several other product-related documents that are necessary for short- and long-term review and control of VENUS product development and marketing.

Initially this document reiterates the DIGITAL strategy and objectives for developing and marketing computer products with a 32-bit architecture. Then the market need for a VENUS product is explained. As part of this, the principal market segments are identified along with their respective product requirements. Further breakdown of the market segments is presented relative to DIGITAL's current organization of the Product Lines. The main competitors of each Product Line are also noted.

Next the product strategy and objectives are given, that is, the plans for the VENUS product, what its principal characteristics are, how it fits in with other DIGITAL products. This is followed by a list of product requirements with development priorities specified.

Finally a product assessment denotes the product's market fit, its competitive goodness, and its positioning and compatibility with other DIGITAL products. Then the assumptions and risks of product development and marketing are given.

The last portion of this document, Appendices A through E, contain general information that is relevant to the VENUS development and marketing.

Note that this Product Requirements document is being published now to assist in the transfer of VENUS product development from Tewksbury to Marlborough. As such, the document is somewhat premature. Release of a fully updated requirements document (based on a preliminary draft authored by Peter Conklin and Bernie Lacroute in November, 1978) was anticipated following the completion of several key information gathering tasks being done in Mid-range Systems Product Management. This work has been suspended temporarily and, instead, all effort has been given to preparation of this interim document based on data available as of April 1979. The information gathering tasks will be resumed by LSG after transfer is complete.

1.0 OVERVIEW

The corporate strategy is to converge on a 32-bit architecture by FY85 with the center of business in systems having MLP less than $250,000. Focus will be given to development of systems for distributed processing and for high availability.

Between now and the mid-1980's the marketplace will demand computer system products that are cost- and performance-effective, easy to use, secure, highly reliable, and family oriented. The products must support distributed processing with interconnections to systems of many vendors and to packet switched networks. They must be laden with rich software (languages, data management, utilities, applications).

The market for VENUS-based systems meeting these product requirements is divided into five segments:

1. scientific computation
2. real-time computation
3. transaction processing
4. general purpose commercial EDP
5. general purpose timesharing

Digital's Product Line Groups are currently organizied to serve these segments as follows:

TECHNICAL GROUP --

TOEM: scientific and real-time computation
LDP : scientific and real-time computation
MSG : real-time computation, general purpose commercial EDP
ESG : general purpose timesharing, scientific computation
ECS : general purpose timesharing
GSG : all segments

COMMERCIAL GROUP --

COEM: general purpose commercial EDP
CSI : general purpose commercial EDP, transaction processing, general purpose timesharing
MDC : general purpose timesharing, real-time computation
T&UG: real-time computation, general purpose timesharing, general purpose commercial EDP

No analysis has been done for the Product Lines in the Computer Group and the Customer Services Group.

VENUS is a product at the high end of the VAX family pyramid during the FY82-85 timeframe. It will meet the market's requirements with product development priorities as follows:

#1 - design center at $180K MLP with performance at 3.5 times VAX-11/780
#2 - new I/O architecture based on ICCS, HSC50, and MERCURY
#3 - SBI capability for -11/780 migration
#4 - FCS in Q1FY82; volume in Q2FY82
#5 - entry level system at $99K MLP
#6 - significant RAMP improvements
#7 - system options
#8 - large system

The VENUS system product is an excellent offering to the traditional Digital markets --- scientific, real-time, timesharing. For transaction processing and for general purpose commercial EDP, VENUS will also be a strong product with the continued development of software products appropriate to those market segments.

The VENUS system product will be compatible with the VAX family architecture. It will use the single VAX family operating system, VAX/VMS. VAX-11/780 migration is supported, and PDP-11 compatibility mode is maintained.

2.0 DIGITAL'S CORPORATE STRATEGY AND OBJECTIVES

The corporate strategy is to develop and market computer products intended for distributed processing systems and for high availability systems. The center of the corporate business will be single-processor systems with a purchase price below $250,000. While current products are based on 8-, 16-, 32-, and 36-bit architectures, there will be convergence to a single 32-bit architecture by 1985. The products based on the latter architecture will be developed and marketed in a manner that provides maximum protection of our existing PDP-11, DECsystem-10, and DECSYSTEM-20 customer base.

3.0 THE MARKETPLACE

3.1 Market Requirements

In the FY82-FY85 time frame, the computer system manufacturers in the EDP industry must offer products that meet the following requirements:

- Cost-/performance-effective computing engines
- Distributed computing capability
- Systems with a high degree of data integrity, internal security and protection
- Effective system interconnection (DEC to DEC, DEC to IBM and other mainframes, X25 for packet switched networks)
- Non-stop computing capability, fault tolerant computing (HYDRA-like configurations)
- Highly reliable systems
- System packaging/operation suitable to an office environment
- System familiness whether or not the individual systems are physically tied into a network
- Highly approachable, easy-to-use systems whether dedicated to a single application or to several different modes of operation
- Training and documentation suited to a wide variety of end users
- Richness of software (languages, data management, utilities, applications)
- Availability of skilled services from the computer system vendor (maintenance, system design, applications programming help)

It will be necessary for vendors of these products to provide complete and accurate cost-of-ownership data to the prospective customers in each market segment. Since the distributed processing style of computing (with more computing on more data by more people) will be emphasized for these products, the amount of mass storage and the number of end user terminals to be purchased will greatly increase from present-day levels.

It will also be necessary to provide prospective customers with comprehensive system performance data that is specific to their particular use of the computer system product. As an example, for a transaction processing application this could be data on system throughput in terms of the number of transactions processed per hour based on such variables as 1) the number of characters in each transaction, 2) the number and speed of the communications lines, 3) the number of multi-drop terminals on each line, and 4) the number of disk accesses per transaction.

A major challenge will be to provide these highly cost-/performance-effective computing engines for systems which require neither a sizable staff of support specialists nor a special operational environment. This means that the systems must be easy to build (dock merge), easy to install (customer merge), easy to use, and easy to repair (self-diagnosis, some customer maintenance). Similarly, the desire of customers to utilize the computer system as a resource to do a specific job (rather than to learn how to be system programmers) and to increase worker productivity puts a great emphasis on 1) the availability of programming languages, data management facilties, utilities and applications software, 2) the reduction (and even elimination) of system and sub-system downtime, and 3) the familiness of systems to ensure that no learning is required when upgrading to a more powerful configuration or when adding another family member to a network.

3.2 Market Segments

VENUS-based systems will be capable of meeting the marketplace requirements outlined above. To provide focus for VENUS product development and marketing, several key market segments can be identified according to how the VENUS-based systems are used, what the characteristics are of computer system products utilized in these market segments, and who these users are.

From the perspective of system use, the market segments for the VENUS product are:

1. Scientific Computation
2. Real-time Computation
3. Transaction Processing
4. General Purpose Commercial EDP
5. General Purpose Timesharing

There are characteristics common to the computer systems utilized in all these segments. These include:

- Easy-to-use, highly approachable, friendly systems for various levels of users
- Easy-to-use program development tools
- Documentation, commands, prompting, error messages in the language and style of the end user
- System HELP facilities
- Internal system security and protection
- Large capacity, high-speed mass storage
- Fast backup/restore between disks and tapes
- Storage hierarchies
- Data management, data integrity
- Support (programming tools, file exchange utilities) for transfer from other current DIGITAL products, migration to future products; system familiness is critical
- Ease of connection to other DEC (DECnet), IBM, CDC, other networks (X25)
- Network transparency to applications programs and to terminal users
- One general purpose operating system with sufficient extensibility/adaptability to serve the entire range of market segments
- Systems configurable/tunable to effective use by a specific market segment (and its changing needs)
- Ease of adding applications packages to the system
- Variety of programming languages
- ANSI-standard languages with validated compilers
- Common for all programming -- call standards, data types, record management, exception handling, run-time library, symbolic debugger
- High system reliability -- extensive H/W, S/W RAMP support; solid quality assurance testing

In addition to the above common characteristics, each market segment has very specific needs to be fulfilled by the VENUS product. By market segment these are:

3.2.1. Scientific Computation

- Multi-user systems
- Good interactive performance
- High-speed processing
- Accuracy
- Fast, mainframe FORTRAN, PL/1
- FORTRAN IV PLUS to match IBM Level H FORTRAN
- Global optimizer (optional during program development)
- APL with file system
- Vector processor with language support
- Large programs
- Sharable programs
- Reliable systems to run large programs to completion
- Mainframe off-loading (ANSI-standard high performance FORTRAN and PL/1, ANSI-standard mag tape, virtual address space)
- Multi-system, high-speed interconnects to other DEC; to IBM, CDC, and UNIVAC
- DECnet, X25
- Batch processing (especially when replacing outmoded IBM systems)
- Data management (for large data files)
- Software routines for graphics displays and plotters
- Applications packages (statistics, project management/control, math library)

3.2.2. Real-time Computation

- High-speed processing
- Rapid context switching
- Low scheduling overhead
- Fast interrupt handling (response to, service of interrupts)
- Fast I/O
- Accuracy
- DECnet (for distributed data acquistion systems)
- Microcoded math functions
- High availability through redundancy, ease/transparency of switchover
- File exchange utility for other DEC products (especially -11M systems)
- Fast, highly-optimized FORTRAN (optimizer optional during program development)
- PASCAL, PL/1, ADA, CORAL-66 (United Kingdom), PEARL (Germany)
- Fully-supported end-user tools for UCB and RMC-11 or equivalent
- Ease of interfacing and supporting special devices.
- Tools for performance measurement, system tuning

3.2.3. Transaction Processing

- Many (10-500) terminals on-line simultaneously in a network
- Terminal cluster controllers with down-line load, up-line dump, data entry/verify, interim storage for off-line data entry
- Intelligent communications subsystems (MERCURY)
- TP concurrent with program development
- Multi-drop terminals
- Intelligent terminals with down-line load
- Fast COBOL, PL/1
- Data management
- Distributed data base management, data base integrity
- Hierarchical data storage, archiving
- Message control
- Forms definition language (compiler, debugger)
- Batch
- Connect to DECnet, IBM, X25
- Fast, reliable communications
- Network transparency to user terminals, applications programs
- High availability (HYDRA-like configuration)
- Journalling
- Shadow recording
- Transaction roll forward/roll backward
- System-/network-wide data directory and dictionary
- Tools for network performance measurement, load balancing, tuning, reconfiguring
- Easy switchover of TP terminal to general purpose use (program development, data inquiry)
- Additional operators' consoles

3.2.4. General Purpose Commercial EDP

- Multi-user system (production EDP runs concurrent with interactive program development, word processing, on-line applications, RJE to mainframes)
- Tools for computer-assisted program documentation
- File design assists
- Support for a family of terminals (dumb to intelligent)
- Applications packages (broad range of capabilties; ease of installation, use, support)
- Multi-volume disk files
- Data management
- Distributed data base management
- Data integrity
- Inquiry language, report writer
- Forms definition language (compiler, debugger)
- Limited transaction processing
- Journalling
- Shadow recording
- Transaction roll forward/roll backward
- Communication with other mainframes (IBM, CDC, UNIVAC); connect to DECnet, X25
- Industry standard languages
- Fast, mainframe COBOL, PL/1
- Interactive BASIC
- RPG II, BASIC, MUMPS, APL with file system
- SORT with MERGE option
- Tools for migration from IBM to DEC
- IBM tape handling (including EBCDIC data)
- ANSI-standard tapes (labels, formats)
- Disk allocation controls and reporting
- BATCH (scheduling, resource allocation, reporting)
- Job class scheduling
- System resource accounting
- System-/network-wide data directory and dictionary

Office automation (connect to remote word processors, backup storage for large documents, document interchange utility, electronic mail)
Support of typeset terminals (SCRIBE editor)
Hierarchical data storage, archiving
System security and protection (e.g., terminals limited to running a single application)

3.2.5. General Purpose Timesharing

Multi-user system (512 edu terminals active sumultaneously)
Good interactive performance (especially for on-line applications; for edit, compile, link/task build, debug/test process)
Large programs
Sharable programs
Interactive BASIC
Multiple languages (FORTRAN, COBOL, PL/1, BASIC, PASCAL, ADA, RPG II, BLISS, APL, MUMPS, ALGOL)
Fast compilers
Fast syntax checkers for all languages
Flexible BATCH, spooler queues
File exchange utilities
Data management
Inquiry language
Report writer
Resource allocation, quotas, scheduling
System resource accounting
Dynamic working set size selection
Applications packages (especially CAI, school administration, project management/control; broad range of capabilities; ease of installation, use, support)
Tools for host development of small systems (DEC) software
Software for graphics displays and plotters
Connect to other mainframes (IBM, CDC, UNIVAC)
DECnet, X25
Scientific computation
Office automation

3.3 DIGITAL'S Marketing Groups

If we now consider the current organization of DIGITAL'S Product Line Marketing Groups, it is possible to identify several of the users (customers) within each of these segments and to denote those customers with requirements spanning two or more market segments. Mainly the Technical and Commercial Groups have been studied because of the applicability of VENUS to their Product Lines. Further study should be done especially with the Word Processing, Computer Special Systems, and Graphic Arts Product Lines.

3.3.1. TECHNICAL GROUP

TOEM, Technical OEM.

Their users are involved with

a. flight training simulation -- real-time computation
b. power monitoring -- real-time computation (high-availability systems)
c. seismic exploration -- scientific computation
d. aerospace systems -- real-time and scientific computation (familiness is required)

Potential new users are those in industrial automation and in telephone and data communications (both real-time computation).

TOEM sees competition from SEL, Interdata, Harris (all in-flight simulation), Modcomp (power monitoring), and PRIME and the mainframe vendors (in seismic and aerospace).

LDP, Laboratory Products

Their users, primarily doing scientific and real-time computation, are involved with

a. U.S. and foreign government research
b. university research
c. energy research
d. industrial research
e. simulation (sensor-based and modeling)

Competition is mainly from SEL, Harris, and Interdata. CDC will remain strong competition since there are many systems (6400, 6600, 7600s especially in energy) to be off-loaded or even replaced by one or more VENUS systems.

MSG, Medical Systems

Their users are

a. medical OEMs -- real-time computation (high availability systems)
b. medical administration -- general purpose commercial EDP (patient billing, report generation, financial applications)

With medical OEM's the main competition comes from IBM, Data General and Honeywell. Hewlett-Packard is becoming very active in this area. In medical administration the strongest competitors are Hewlett-Packard, IBM, and NCR.

ESG, Engineering Systems

Their users specialize in

a. aerospace design
b. automotive design
c. chemical engineering
d. electronic/electrical design
e. engineering consulting
f. architectural and engineering design

A general purpose timesharing system is required with strength in the area of scientific computation.

Competition comes mainly from Data General, Hewlett-Packard, PRIME, CDC, and IBM. Sometimes Harris and Interdata are seen.

ECS, Education

Their users, always conscious of system price, prefer a general purpose timesharing system for large numbers of users with specialties as noted:

a. school district students -- BASIC programs (small-scale problem solving, little I/O); good text editors, fast compile, debuggers for program development; not production.
b. university students -- technically sophisticated; more FORTRAN than BASIC; heavy text editing; large, complex FORTRAN and COBOL programs.
c. university departments -- variety of languages; some large FORTRAN programs.
d. school administration -- RJE required to installed main frame; high performance COBOL, some data management; administrative usage is growing; on-line, interactive applications; need application packages.
e. private college administration and students -- good COBOL and data management plus applications packages for administration; students need BASIC, FORTRAN, COBOL.

The competitors of ECS are IBM, Harris, PRIME, Hewlett-Packard.

GSG, Government Systems

Their users are from every organization within Federal governments around the world. The needs of these users cover every market segment described above. In the United States, particular emphasis is given to users in

a. military intelligence -- scientific and real-time computation; extensive PDP-11 experience to migrate.
b. civilian agencies -- with competitive procurements and long systems lifetime (5-8 years), systems require wide range of capabilities and low life cycle cost; interface to other mainframes (IBM, CDC, UNIVAC).

Generally, the competitors of GSG are IBM, Univac and Honeywell. SEL also does considerable U.S. Government business. For governments in countries outside the U.S., competition is usually seen from any vendor native to the country.

3.3.2 COMMERCIAL GROUP

COEM, Commercial OEM

Their users acquire systems intended for operation with specialized applications packages, e.g., business office management (wages and payroll, accounts payable, general ledger), customer billing, order entry and inventory control, sales analysis. Many of these systems are sold to small manufacturing and distribution companies and to lawyers, accountants, physicians, and dentists.

Competition for COEM comes primarily from IBM, Wang, and Basic Four in addition to DG and HP oems.

CSI, Commercial Service Industries

Their users work in

a. banks, insurance companies, financial institutions
b. data services companies
c. transportation companies
d. state and local government
e. retail business
f. service business

Many of these users require general purpose commercial EDP systems. A low-cost subset of the general purpose system can be geared to users in the small business community. Transaction processing is becoming extremely important to these users as is real-time computation for communications and sensor-based applications. Data service companies are interested in general purpose timesharing.

Competition is very strong from IBM at the account level. Other competitors are Hewlett-Packard, Data General, Honeywell, Tandem, NCR, PRIME, Computer Automation, and Burroughs.

MDC, Manufacturing Distribution and Control

Their users are found in

a. manufacturing companies
b. process industries

Required is a general purpose timesharing system to be used mainly for host development of software for smaller -11 based production systems running RSX-11M or RSX-11S. This general purpose system must have a good FORTRAN with an easy-to-use file system plus COBOL for data processing activities such as inventory control and materials scheduling. Real-time computation is becoming a requirement for more of these users.

Competitors for MDC are IBM, Tandem, SEL, Perkin-Elmer (Interdata), and Harris. Hewlett-Packard is beginning to enter this market area.

T&UG, Telephone & Utilities

Their users are involved with

a. telephone equipment manufacturers -- real-time computation, general purpose timesharing for development of specialized applications (e.g., traffic monitoring, billing data collection, repair order administration)
b. operating telephone companies -- general purpose commercial EDP with emphasis on communications with IBM mainframes.

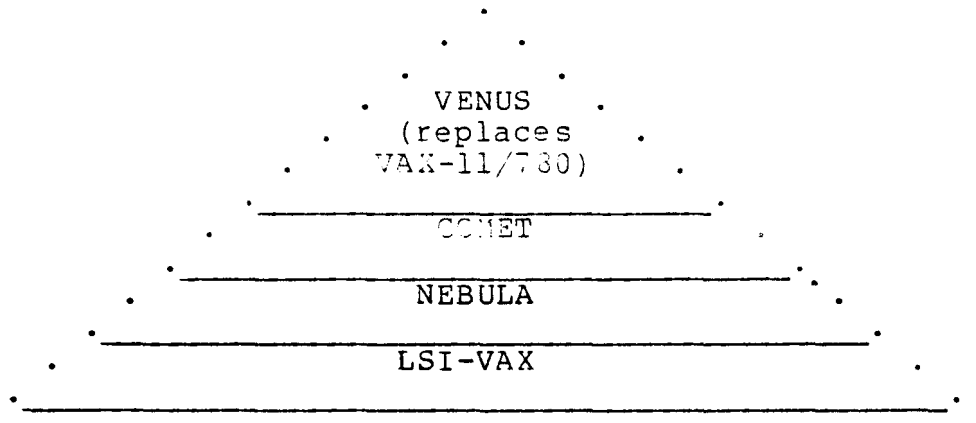
Competition comes from Interdata, especially at Bell Labs. IBM and Hewlett-Packard are competitors in the operating companies. Tandem is a potential competitor.

4.0 PRODUCT STRATEGY AND OBJECTIVES

VENUS is a high performance computer system product which implements the VAX-11 architecture and runs the VAX/VMS operating system with its associated layered software products. The MLP ($) design center of VENUS is the same as that of the VAX-11/780. Increased system performance is achieved by applying state-of-the-art technology (e.g., Mosaic ECL Array Technology) to the VAX-11 architecture.

In the FY82-FY85 timeframe VENUS is at the top of a pyramid of VAX family distributed data processing products in terms of

- general purpose computing capability,
- cost and performance
- ease of interconnection to and strict compatibility with all other members of the pyramid.



VENUS will remain as a product offering until FY87 although a new, higher performance processor might be introduced in the FY85/86 timeframe.

As the VAX-11/780 replacement, the VENUS product is consistent with the corporate strategy which calls for a single 32-bit system architecture by 1985. Furthermore, VENUS will augment the VAX family of 32-bit products in a fashion which is consistent with the corporate goal calling for concentration on systems priced at $250,000 or less.

In a single processor configuration or in distributed processing configurations, VENUS provides the functional base for scientific and real-time computations, transaction processing, general purpose commercial EDP, and general purpose timesharing. VENUS can also be utilized in HYDRA multi-processor configurations (high availability, non-stop computing systems).

VENUS systems comply with the constraint of having VAX/VMS serve as the single operating system for the entire VAX family. A vast array of layered software products (such as COBOL, FORTRAN, DATA BASE MANAGEMENT), system options (such as TRANSACTION PROCESSING, HIGH AVAILABILITY modules), and applications will be offered. These will be VAX/VMS system add-ons in much the same fashion as, for example, disk and tape hardware sub-systems are field add-ons at an existing customer installation.

5.0 PRODUCT REQUIREMENTS

The VENUS system product will satisfy the market requirements outlined above. Priorities to guide the development of this product are given below (in descending order of importance):

- Design center at $180K MLP with performance at 3.5 times VAX-11/780
- New I/O architecture based on ICCS, HSC50, and MERCURY
- SBI capability for -11/730 migration
- FCS in Q1FY82; volume in Q2FY82
- Entry level system at $99K MLP
- Significant RAMP improvements
- System options
- Large system

The capabilities and functions to be developed for VENUS-based systems according to these priorities are illustrated in the following chart.

VENUS PRODUCT REQUIREMENTS

	ENTRY LEVEL SYSTEM *	DESIGN CENTER SYSTEM *	LARGE SYSTEM	MAX. CONFIG.
CPU with CIS warm floating point(includes G and H)	yes	yes	yes	yes
ECC MOS Memory	1MB	4MB	16MB	32MB
Vector Processor	none	none	1	1
Disks	2 x 40-50MB, removable	600MB, fixed or fixed/ removable	4 x 600MB, fixed or fixed/ removable	4 x 600MB, fixed or fixed/ removable
Mag tape	none	1600/6250 bpi, 125ips	2 x 6250 bpi, 200ips	4 x 6250 bpi, 200ips
Console with terminal plus dual load device	yes	yes	yes	yes
Asynch lines	8	8	128 (MERCURY)	512 (MERCURY)
Line Printer	none	none	1	1
Card Reader	none	none	1	1
SBI	none	none	none	2
ICCS	1	1	2	4
Remote diagnostics, console port	yes	yes	yes	yes
Cabinetry, power supplies	single cabinet w/exp.space	single cabinet w/exp.space	TBD	TBD
On-line diagnostics, UETP	yes	yes	yes	yes
VAX/VMS	yes	yes	yes	yes
Languages	1	any (not in $180K)	any	all
MLP	$99K	$180K	TBD	TBD

*Note expansion possibilities under discussion of Priority #1 and #5.

PRIORITY #1: Design center at $180K MLP with performance at 3.5 times VAX-11/780.

With an MLP of $180K, the system transfer cost is $40K (based on MU=4.5).

The basic components of this dock merge system product are:

- CPU with CIS and warm floating point (including G and H)
- 4MB ECC MOS memory
- 600MB disk mass storage; fixed or fixed/removable media
- 1600/6250 bpi, 125 ips magnetic tape subsystem
- Console, including terminal and dual load device for software patches, software distribution
- 8 asynchronous lines
- ICCS I/O bus
- Remote diagnostics with console port
- Cabinetry, power supplies
- On-line diagnostics, UETP
- VAX/VMS operating system (language licenses not included in the $180K)
- Expansion space in this single cabinet for
  - 4MB ECC MOS memory (additional)
  - 24 asynchronous lines (additional)
  - 1 line printer
  - 1 card reader
  - 6-8 synchronous lines
  - 1 accelerator (FORTRAN or COBOL)
- Note that the $180K MLP covers the pre-wiring for these expansion components only and not the components themselves.

The configuration rules for this design center system and for all other VENUS-based systems must be easily stated and must be subject to easy verification by all sales people.

Performance at 3.5 times VAX-11/780

FORTRAN and COBOL --

The best FORTRAN and COBOL performance must be

FORTRAN = 3032 = 370/168

COBOL = 3032 = 370/168

This performance can be achieved via accelerators or any other engineering option.

When these high-performance capabilities are removed from the system, the performance is at the 3031 or 370/158 level.

FORTRAN is measured using the Whetstone and SP1111 benchmark programs. The performance for data types F, D=G, and H should each meet these goals compared to the corresponding IBM data types.

COBOL is measured using the U.S. Steel and Profile benchmark programs. The performance for display, binary, and index subscripts and for trailing overpunched and packed decimal data should each meet these goals compared to the corresponding IBM measures.

Real-time --

Times for context switching, CALL, and response to/service of interrupts must be at least 3 times faster than the speedier of COMET and VAX-11/780.

Throughput --

Memory bandwidth for VENUS must be at least 30MB/second with access times much faster than those of the -11/780. Further, a VENUS system configured with the new ICCS I/O bus must be capable of handling the equivalent of 4 UNIBUSes plus 8 MASSBUSes in addition to the maximum allowable number of intersystem connections. This assumes that the ICCS bus also supports (via MERCURY) line printer, card reader, asynchronous and synchronous communications lines, customer real-time devices, and slow-speed mass storage units. For availability reasons, it must be possible to configure a single VENUS system with at least two ICCS busses.

Delays in development --

Should the time to market goal not be met, it is required that all performance specifications given above will increase by 30% per year.

Availability of performance data --

An extensive set of performance measurements tasks must be done to provide data that is relevant to customers in the respective market segments. These performance analyses must be completed with results available by the time of VENUS product announcement. Such performance measurement projects must continue throughout the lifetime of VENUS in response to its changing environment (e.g., in terms of new DIGITAL products and competitors' new offerings).

PRIORITY #2: New I/O architecture based on ICCS, HSC50, and MERCURY

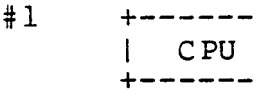
VENUS will have the new I/O architecture based on the ICCS bus, the HSC50 mass storage controller for disk and tape, and the MERCURY intelligent communications subsystem for asynchronous and synchronous lines and for unit record equipment.

PRIORITY #3: SBI capability for -11/780 migration

To facilitate migration of the current VAX-11/780 customers to VENUS, ports for UNIBUS and MASSBUS devices must be provided via the SBI interface.

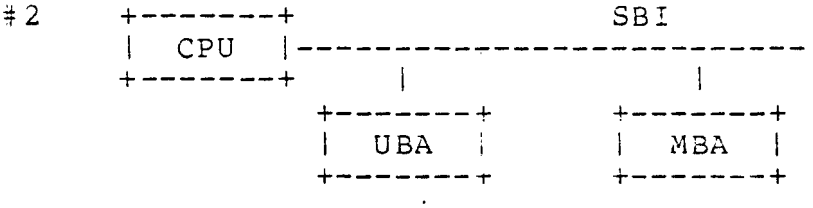
Under the above two priorities (ICCS and SBI capabilities), the various i/o configurations for VENUS systems and the priorities for their development are:

#1



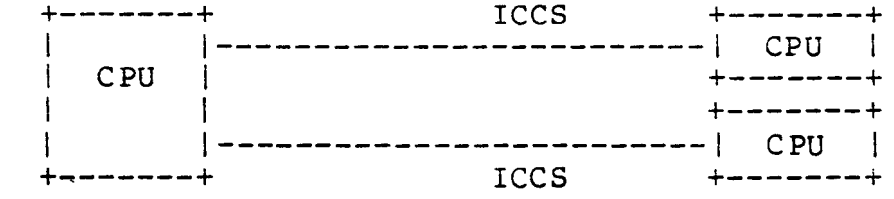
For the design center system and the entry level system. This must be available at FCS.

#2

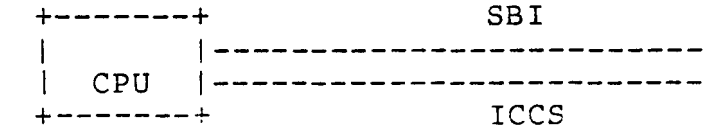


For -11/780 customers wanting an upgrade to a system with a more powerful central processor but with support for existing peripheral devices (via the UBA and/or MBA). This must be available at FCS.

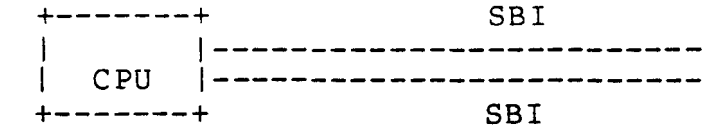
#3



For customers with a distributed processing operation comprised of high-performance connections among intelligent subsystems. These high availability, high reliability systems will be suited for transaction processing, real-time computation, and general purpose timesharing off-loaded from a mainframe. This I/O configuration is part of the VENUS/HYDRA configuration and must be available at FCS.



For customers (especially OEM) needing a system with a link to the past (via the SBI) and to newly-developed products (via the ICCS).



Again for -11/780 customers wanting a more powerful central processor but with increased i/o performance using existing peripheral devices.

PRIORITY #4: FCS in Q1FY82; volume in Q2FY82

The VAX-11/780 product was announced in October, 1977 with FCS in December, 1977 (Q2FY78). Assuming the need for a replacement product every 3-4 years, the desired announcement date for the VENUS product is Q3 or Q4FY81. First customer shipments will follow in Q1FY82. Volume shipments will be reached by Q2FY82.

To summarize the key dates for VENUS:

Announcement of design center product	Q3/Q4FY81
First customer shipment (FCS)	Q1FY82
Volume availability	Q2FY82
Availability of	
Entry level system	FCS + 3 months
Vector processor	FCS + 6 months
Large system	FCS + 9 months

PRIORITY #5: Entry level system at $99K MLP

The entry level system is aimed at cost-sensitive applications. At a $99K MLP, this system permits a marketing campaign based on pure system cost. With an MLP of $99K, the system transfer cost is $22K (based on MU = 4.5).

The entry level system is bounded. Expansion is allowed but only at a price which does not disturb the product's design center business.

The basic components of this dock merge system are:

CPU with CIS and warm floating point (including G and H)
1MB ECC MOS memory
2 x 40/50 MB ea. disk drives (removable media)
Console, including terminal and dual load device for software patches, software distribution
8 asynchronous lines
ICCS I/O bus
Remote diagnostics with console port
Cabinetry, power supplies
On-line diagnostics. UETP
VAX/VMS operating system with license for one language

Expansion space in this single cabinet for
- 1MB ECC MOS memory (additional)
- 8 asynchronous lines (additional)
- 1 line printer
- 1 card reader
- 6 x 40/50MB ea. disk drives (removable media)
- 2 synchronous lines
- 1 accelerator (FORTRAN or COBOL)

Note that the $99K MLP covers the pre-wiring for these expansion components only and not the components themselves.

PRIORITY #6: Significant RAMP improvements

VENUS development must expand on the RAMP designed and implemented for the VAX-11/780. The BMC of the resulting product must not exceed 1.5% of its transfer cost. RAMP plans should include software warranty and installation cost goals. RAMP must be considered according to the customer's perception of a total system, e.g., Are spares available when needed? Can troubleshooting be done without taking the entire system down? Is field software support responsive? Did the system product undergo enough quality assurance testing?

Improved RAMP is necessary for at least two reasons:

1. Customers are demanding highly reliable systems. They are becoming increasingly intolerant of computer system interruptions which wreak havoc throughout their organization.

2. With the growth of distributed data processing systems, customers will see growing maintenance costs for the many system processing units that are spread across a wide geographical area. Customers will not pay these high service costs. Furthermore, system vendors will be unable to provide a sufficient (and large) number of capable service personnel for such maintenance.

For VENUS, undetected failures must be minimized, and the total unrecovered system crash rate must be reduced. The latter includes all crashes attributed to environmental causes, operational procedures, software, hardware, and unexplained failures.

The MTBF must be increased significantly from that of the -11/780. Further, the total unproductive time per year must be reduced drastically. This includes, but is not limited to,

1. hardware preventive maintenance
2. software updates and maintenance
3. disk backup operations
4. emergency system maintenance
5. time spent waiting for parts
6. the ambiguous time during repetitive and undetected failures (especially for service calls which do not isolate the problem cause)

It is expected that reducing non-productive time will require significant changes in operational and service philosophies.

As part of the VENUS RAMP, consideration should be given to those features available from the HYDRA learning experience. Examples are redundant power supplies and fans plus devices with redundant access paths.

PRIORITY #7: System options

To meet the requirements of the marketplace, the VENUS system product will have several options as listed below. These options must be included in VENUS from the start of product design and development. Availability to the marketplace is noted here as "FCS+n" in months.

Hardware --

(FCS+9) 32MB max. ECC MOS memory (system total)
(FCS+12) I/O busses: max. of 4 ICCS
(FCS+12) max. of 2 SBI each with up to 2 UBAs plus 4 MBAs
(FCS) disks: 600 MB, fixed media
(FCS+3) 40-80 MB, removable media
optional dual channel access for both
(FCS+9) tapes: 6250 bpi, 200 ips, auto load, radial bus, dual channel access (optional)
(FCS) 1600/6250 bpi, 125 ips
(FCS) unit record equipment: line printer (IBM quality, VFU, DMA)
(FCS) card reader (DMA)
(FCS) processor options: FORTRAN (accelerator)
(FCS) COBOL (accelerator)
(FCS+6) Vector processor
(FCS) VCS, VMC (or equivalent)
(FCS) MA780 (including COMET shared memory systems)
(FCS) DR780
(FCS) terminals: multi-drop terminals for TP
(FCS) VT100-based terminals
(FCS) PDT terminals
(FCS) GIGI terminal
(FCS) Typeset terminals
(FCS) terminal clusters

Communications --

(FCS) DECnet
(FCS) X25
(FCS) Interconnect to IBM, CDC, UNIVAC
(FCS) MERCURY communications controller
(FCS) DMA/buffered asynchronous and synchronous lines

Software (native mode) --

(FCS)	SORT/MERGE	(FCS)	Interactive BASIC-PLUS
(FCS)	APL with file system	(FCS)	BASIC-PLUS-2
(FCS)	PL/1	(?)	ADA
(FCS)	PASCAL	(FCS)	CORAL-66
(FCS)	BLISS-32	(?)	PEARL
(FCS)	RPG II	(FCS)	MUMPS
(?)	ALGOL	(?)	LISP

Note for all new languages: compliance with existing ANSI-standard language specifications; validated compilers.

(FCS) Symbolic debuggers for all languages
(FCS) Language support for vector processor
(FCS) DBMS-32
(FCS) DATATRIEVE-32 (inquiry language, report writer)
(FCS) TRAX-32
(FCS) Forms language compiler, debugger
(FCS) Message control with transaction roll forward/backward, journalling, shadow recording
(FCS) Multi-volume disk files
(FCS) ANSI-standard mag tape handling routines
(FCS) IBM mag tape handling
(FCS) Routines for graphics displays and plotters
(FCS) Math library
(FCS) Routines for performance measurement, network tuning, applications program tuning
(FCS) System resource accounting
(FCS) Resource allocation, quotas, scheduling (especially by JOB class); all in BATCH also
(FCS) Support routines for office automation (interface to remote word processors, backup storage for large documents, document interchange utility, electronic mail)
(FCS) Routines for RSTS migration (emulators, conversion utilities)
(FCS) Cross-system development for RSX-11M, -11S, RT-11, RT2.

General --

(FCS) Node in a HYDRA configuration
(FCS) 512 simultaneous educational users (BASIC-PLUS on a single-processor system)
(FCS) System-/network-wide data dictionary, data directory
(FCS) H/W, S/W support of all devices on -11/780, COMET, NEBULA, HYDRA, FONZ, SCS, PDT.
(FCS) Additional operator consoles

PRIORITY #8: Large system

The components of this system are:

CPU with CIS and warm floating point (including G and H)
16MB ECC MOS memory
4 x 600 MB disk mass storage; fixed or fixed/removable media; dual channel access
2 x 6250 bpi, 200 ips magnetic tape; auto load; dual channel access
Console, including terminal and dual load device for software patches, software distribution
128 asynchronous lines (MERCURY)
2 x ICCS I/O bus with two (2) ICCS ports connected to VENUS, COMET, or NEBULA processors
2 synchronous lines to IBM or CDC (MERCURY)
1 line printer (MERCURY)
1 card reader (MERCURY)
Vector processor
Remote diagnostics with console port
Cabinetry, power supplies
On-line diagnostics, UETP
VAX/VMS operating system with FORTRAN, COBOL, PL/1, BASIC, DBMS, PASCAL

6.0 PRODUCT ASSESSMENT

6.1 Market Fitness and Competitive Goodness

The potential for success of the VENUS system product in its marketplace is summarized below:

MARKET SEGMENT	ASSESSMENT
Scientific Computation	Excellent
Real-time Computation	Excellent
Transaction Processing	Very Good
General Purpose Commercial EDP	Very Good
General Purpose Timesharing	Excellent

Against the competitive products either currently offered in the market or else known to be under development for release during VENUS' product life, VENUS should be a very strong performer for DIGITAL, especially in the traditional market segments (scientific, real-time, general purpose timesharing). The challenge will be to achieve the same level of excellence for transaction processing and for general purpose commercial EDP. By and large, success here depends upon our ability

1. to develop a considerable number of software products in time for the VENUS announcement and shipment;

2. to gain TP and commercial experience and to build a reputation as a viable vendor of high-end commercial-oriented products;

3. to understand how to win a sizable share of these two markets which are now dominated by extremely strong, well-entrenched competition.

Particular attention must be given here to the competitive challenge of IBM in all our market segments. With the coming H-series to complement their current 4300-series, the 8100, and the System/38, IBM will appear to have a comprehensive product offering aimed specifically at the distributed processing marketplace. The best resources of DIGITAL will be required to beat back this challenge with a rich array of products that can win wide customer acceptance.

6.2 VAX Family Product Positioning

As stated earlier, VENUS is at the top of a pyramid of VAX family distributed processing products. A brief description of each product is given below.

VENUS

- High end of the VAX family.
- VAX-11/780 replacement.
- Top of distributed computers hierarchy.
- General purpose capabilities for Scientific and Real-time Computation, Commercial Data Processing, Time Sharing, BATCH.
- Basic system optimized for $180K sale price.
- Configurable in high availability topologies (HYDRA).
- Attack product for new customers and PDP-11 customers in the $150K-$300K average systems range.
- Migrate top end of 11/74-MP business to VENUS.

Competition for VENUS is

- IBM 303X, 370, 4300
- CDC Cyber
- Burroughs
- NCR
- Honeywell
- SEL

COMET/HYDRA

Mid-range VAX family product at center of corporate business ($50K-$80K) for single processor applications.
Main product for distributed data processing host machine at the department/group level.
Tailored by application of VMS software options to Scientific Computation, Commercial Data Processing, Real-time Computation, Time-sharing, or Transaction Processing environment.
Attack product for new customers and PDP-11 customers in the $50K-$150K range.
Attack product for non-stop systems.
Migrate low end of 11/780 business, some of 11/74-MP business, and top end of 11/44.

Competition for COMET/HYDRA is:

IBM Low end 370, 4331
HP 3000
DG M600
S2?0
DG 32-bit
PRIME
Tandem
Interdata
SEL

NEBULA

Low end VAX family product.
Optimized for $25K systems range.
Tailored to specific application by packaging VMS options.
Attack product for new customers and PDP-11 customers in the $20K-$50K range.
Migrate bulk of 11/34, 11/44 business.

Competition for NEBULA is:

IBM Series 1, 8100
HP 3000, HP 1000
DG S250 + New Series
PRIME
Microdata

LSI/VAX

Very low end VAX family product.
Optimized for personal, lab, and office use.
$8-$10K system.
Bottom of the distributed data processing pyramid.
Tailored to a specific function by packaging of VMS (sysgen out functions).
Migration for 11/04, LSI-11 business; co-exist with PDP-11 bounded systems.

Competition for LSI/VAX is:

DG Micro Nova
HP desk top
Microdata
Intel
Wang

6.3 Compatibility with DIGITAL Systems

The key compatibility issues relative to VENUS product development are:

1. VAX family architecture is maintained.

2. The strategy of one operating system, VMS/VMS, is maintained.

3. VENUS supports SBI ports to allow connection (with no changes) to the MA780 and DR780.

4. UNIBUS and MASSBUS ports are provided to facilitate migration of the current -11/780 customer base.

5. The new ICCS I/O bus structure is implemented in a uniform fashion across all family members.

6. The PDP-11 compatibility mode is maintained as in the -11/780.

7. It is possible to use the library of VAX diagnostics unchanged for all existing devices.

8. RSX-11M compatibility is maintained in emulator mode.

9. There is a continued convergence on a single on-disk structure (ODS II), file access method (RMS), and command language (DCL).

10. Tools will be developed to support VENUS-based RT-11 development of software for PDT clusters and the FONZ and also SCS-11 host development.

6.4 Product Development Assumptions

The VENUS product development assumes that VAX/VMS is the one operating system maintained for the entire family. Further, the VAX family architecture is maintained, and all implementations are consistent throughout the family.

In this product development, emphasis is placed on languages, data management, communications, ease of use, and system availability. VENUS is suitable as a node in a HYDRA configuration. DECnet is an integral and critical part of the VENUS product. X25 and interconnects to IBM, CDC, UNIVAC are integrated into continued VAX/VMS development.

Tailoring of hardware products to the VENUS marketplace is achieved by adding layered software products and/or boot-time selecting VAX/VMS as appropriate.

6.5 Product Development Risks

A major risk involves timely development of several software products required especially by the commercial-oriented market segments. The significant challenge of this product development is being met today. Major software development projects are currently underway. Others are in the planning stages and require corporate funding and commitment of resources.

The design center and entry level systems depend on the availability of mass storage subsystems (disk and tape) that are significantly more cost-/performance-effective than our current product offerings.

VENUS is the first DIGITAL product scheduled to use the Mosaic ECL Array technology. Originally VENUS development plans had included the opportunity to learn from the DOLPHIN experience with the MCAs. Their extensive diagnostic capability will contribute heavily to achievement of VENUS' higher RAMP goals.

To achieve the higher RAMP goals, significant changes must be made in the philosophies governing hardware/software design and implementation and system support in the field.

APPENDIX A: VENUS Systems

Representative configurations of VENUS systems are:

1. 1MB memory
   ICCS bus
   2 x RL04 via UDA
   8 asynchronous lines
   VMS + one language
   $99K MLP (entry level system)

2. 2MB memory
   SBI
   RM/RA80 via MBA
   TU77
   8 lines
   VMS

3. 4MB memory
   ICCS bus
   RP08 via HSC
   TU78
   8 lines
   VMS (no languages)
   $180K MLP (design center)

4. 16MB Memory
   2 x ICCS bus
   4 x RP08 via HSC
   4 x TU78
   128 lines (MERCURY)
   VMS

5. 2 x System #2
   1 MA780 with 1MB memory (esp. for OEM)

6. 2 x System #4 with 256 lines total (this is VENUS/HYDRA)

APPENDIX B: Preliminary Product Forecasts and Assumptions

In late December 1978 Al Avery and Peter Conklin prepared a forecast of VENUS units covering FY82-FY85. The forecast amounts for all high-end mid-range systems (-11/70, -11/74, -11/780, and VENUS) were derived and appear here with the assumptions of the forecasting exercise.

FORECAST

	78	79	80	81	82	83	84	85
Corp. NOR ($G)	1.44	1.87	2.43	3.16	4.11	5.34	6.94	9.02
15% of NOR ($M)	216	280	365	475	615	800	1040	1350
Units:								
11/70 + 11/74	1400	1800	1800	1600	1000	500	100	0
11/780	35	550	1000	1500	1800	1500	1000	800
VENUS	0	0	0	0	400	1800	3600	5900
TOTAL UNITS ---	1435	2350	2800	3100	3200	3800	4600	6300
Discount:								
11/70 + /74 (%)	15	----------						
11/780 + VENUS (%)	11	11	11	13	15	12	12	14
NOR:								
11/70 + /74 ($M)	180	260	260	230	145	70	15	0
/780 + VENUS ($M)	6	117	220	325	470	730	1025	1350
TOTAL NOR ($M)	186	377	480	555	615	800	1040	1350

APPENDIX B: (continued)

ASSUMPTIONS

1. High-end Packaged Systems = 15% of corporate NOR.
2. Corporate NOR grows at 30%/year compounded.
3. By the beginning of FY81, 11/780 has sufficient commercial functionality to pick up the high-end 11/70 commercial business.
4. DEC will have learned by the beginning of FY81 how to penetrate high-functionality commercial business.
5. No 16-bit products beyond 11/74 will be developed at the high end ($).
6. VENUS FCS = Q2/82; has all functionality: 3X performance @ +15% MLP. (NOTE: Requirements now are FCS = Q1FY82, Vol. = Q2FY82, 3.5 X -11/780 performance.)
7. VENUS is new market attack product; 11/780's continue to be built in whatever volumes are required to satisfy customer demands, namely, no great forced migration from /780's to VENUS.
8. OEM's and TELCO continue to buy high-end midi systems.
9. Discounts: 11/70 and 11/74 = 15% flat
10. Add-on business = 15% of average of previous two years NOR.
11. This forecast addresses the Mid-range Systems contribution to corporate NOR. Large Systems Group will be responsible for an updated forecast based on VENUS' replacement of current LSG products.
12. No analysis has been made here by the Product Lines to determine how the volume given above can be achieved.

APPENDIX C: Comparison Prices and Costs for VAX-11/780

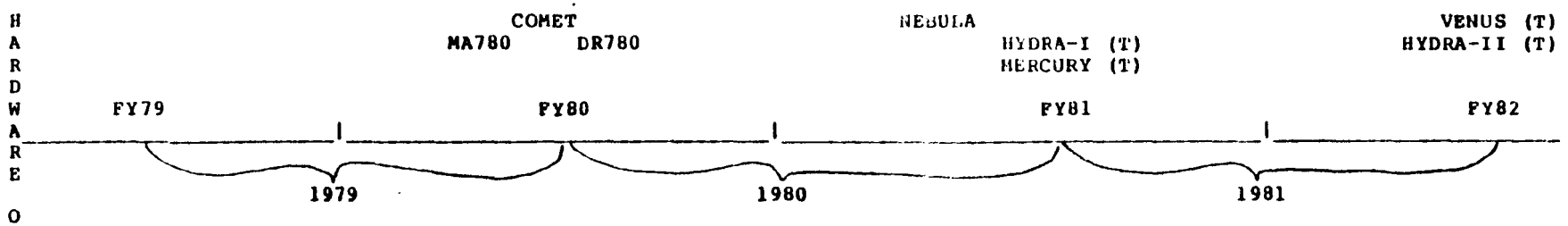
Given here are the configuration, price, cost and BMC of VAX-11/780 packaged systems.

DESCRIPTION	FY80 MLP (M.U.)	FY80 Est. Transfer Cost	FY80 BMC (% XFER)
VAX-11/780 System, 512KB memory, 2 x RK07 (28 MB ea.), 8 asynch. comm. lines, and virtual memory operating system software.	$134,600 (x5.1)	$ 26,600	$748 (2.8%)
VAX-11/780 System, 512KB memory, RM03 disk pack (67MB), TE16 magnetic tape (1600/800 bpi, 45 ips), 8 asynch. comm. lines, and virtual memory operating system software	167,000 (x4.6)	36,100	783 (2.2%)
VAX-11/780 System, 512KB memory, RM03 disk pack (67MB), TU77 magnetic tape (1600/800 bpi, 125 ips), 8 asynch. comm. lines, and virtual memory operating system software.	177,000 (x4.3)	40,700	843 (2.1%)
VAX-11/780 System, 1MB memory, RP06 disk pack (176 MB), TE16 magnetic tape (1600/800 bpi, 45 ips), 8 asynch. comm. lines, and virtual memory operating system software.	207,000 (x4.8)	42,800	937 (2.2%)
VAX-11/780 System, 1MB memory, RP06 disk pack (176MB), TU77 magnetic tape (1600/800 bpi, 125 ips), 8 asynch. comm. lines, and virtual memory operating system software.	217,000 (x4.7)	46,500	997 (2.1%)
"UNIBUS" VAX-11/780 System, 256KB memory, 2 x RK07 (28 MB ea.), 8 asynch. comm. lines, and virtual memory operating system software. (system subject to corporate approval)	99,800 (x3.9)	25,300	698 (2.7%)

APPENDIX D: VAX Software Plans

This calendar was prepared by Peter F. Conklin

V A X C A L E N D A R



OPER. SYSTEM
- Multi-User
- Real Time
- Interfacing
- Tunable
- Batch
- Operator
- Fast I/O
- Tapes
- Mount-Disks
- RMS
- English Messages

M-V Disks
Alloc. Ctrl.
ISAM

Resource Alloc.
SMP Breadboard
Disk Quotas
MA780

	R1.0	R1.5	R2.0	R3.0(T)	R4.0 (T)
LAYERED COMPONENTS	COMPUTATION	COBOL-74	PASCAL	FORMS (T)	ALLSP (T)
	DECnet	SORT	BASIC +	3271 (T)	STEP (T)
	FORTRAN	BLISS	BASIC +2	FAST BACKUP (T)	
	DATATRIVE	RPG-II	RSTS MIGR.I	NATIVE DATATR.(T)	
	COMMON LANG.	GRAPHICS	DECnetIII	SNA (T)	
	DEBUG	DX	MUX200	RSTS REPLACEMENT (T)	
	DCL		APL	COBOL 79	
	MCR		DECmail	PL/I (T)	
	EDITORS		CORAL-66	TYPESET	
	DIFFERENCES		KMC TOOLS	CATS (T)	
	RUNOFF	2780/3780	MULTI-DROP	MERGE (T)	DIST.DATA MGMT (T)
	RSX MIGR.	WCS TOOLS		MGMPS (?)	
				PEARL (?)	
				X.25 (?)	
				LISP (?)	

(T) = Target, not yet phase 1 commit
(?) = Possible; plans not firm

APPENDIX E: Related Documentation

Existing documents that are useful for anyone working in VENUS product development and marketing are listed below:

1. VENUS Project Proposal (27 Dec. 1978)
   Contact Steve Jenkins, TW/C04, DTN #247-2395.

2. VENUS Product Description (20 Jan. 1979)
   Contact Steve Jenkins

3. VENUS Impact Statement (13 Feb. 1979)
   Contact Don Ames, TW/A02, DTN #247-2517.

4. VENUS Software Plans (10 Apr. 1979)
   Contact Peter Conklin, TW/A08, DTN #247-2119.

5. VAX/VMS R2.0 Requirements Document (Sept. 1978)
   Contact Kathryn Norris, TW/A08, DTN #247-2580.

6. System plan for VAX/VMS (10 Jan. 1979)
   Contact Joe Carchidi, TW/D08, DTN #247-2251.

7. VAX/VMS RELEASE TWO Project Plan (9 Feb. 1979)
   Contact Trevor Porter, TW/D08, DTN #247-2262

8. Commercial Market Product Requirements (March 1979)
   Contact Roger Cady, MK1-1/E25, DTN #264-5045.

9. NEBULA Product Requirements (Feb. 1979)
   Contact Lou Philippon, TW/A08, DTN #247-2860.

VENUS PRODUCT REQUIREMENTS - DISTRIBUTION LIST

Gordon Bell	ML12-1/A51	Robert Lane	MK1-2/B11
Ron Bingham	MR1-2/E85	John Leng	MR1-1/A65
Brian Croxon	TW/C04	Bill Long	ML10-2/A57
Art Campbell	PK3-1/M12	Si Lyle	MR1-1/M42
Roger Cady	MK1-2/E25	Ward MacKenzie	PK3-1/A60
Patrick Courtin	MK1-2/D29	Julius Marcus	MK1-2/C37
Dick Clayton	ML12-2/E71	Jim Marshall	TW/A03
Bill Demmer	TW/D19	Bob Nealon	MR2-4/F19
Raff Ellis	MR2-4/M79	Ken Olsen	ML10-2/A50
Ulf Fagerquist	MR1-2/E78	Stan Olsen	MK1-2/C36
Barbara Farquhar	MR2-4/M79	Stan Pearson	ML12-2/E71
Ed Fauvre	MK1-2/E06	George Plowman	ML5-5/E97
Jack Gilmore	MK1-1/J14	Larry Portner	ML12-3/A62
Rose-Ann Giordano	MR1-2/A65	Franco Previd	MR1-2/E18
Mike Gutman	ML3-6/E94	Dick Rislove	MK1-2/L35
Bill Heffner	TW/C10	Joel Schwartz	MR2-4/M51
Win Hindle	ML10-2/A53	John Shebell	MR1-1/S35
Per Hjerppe	MR1-2/E78	Jack Shields	PK3-2/A58
George Hoff	MR1-2/E47	Leo Shpiz	MK1-2/H32
John Holman	PK3-1/P84	Pete Smith	MR1-1
Irwin Jacobs	MK1-2/H32	Dick Snyder	MR1-2/E37
Bob Joseph	MR1-1/M42	Charlie Spector	ML5-2/A33
Paul Kelley	MR1-2/E18	Harvey Weiss	MR1-1/M85
John Kevill	ML3-6/E84	Jim Willis	MK1-2/H32
Bill Kiesewetter	MK1-1/M49	Jerry Witmore	PK3-1/M40
Bob Klein	MR1-1/M85		

MSD PRODUCT MANAGEMENT

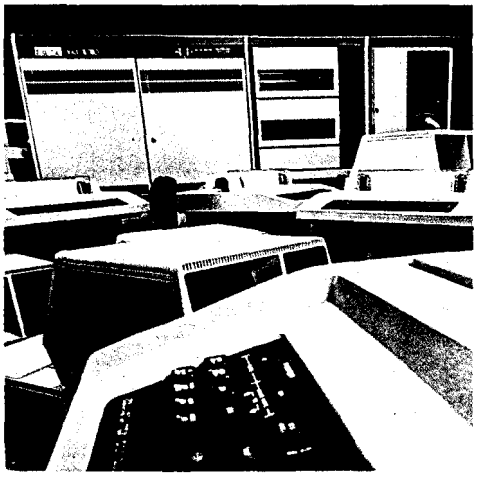
Al Avery	TW/A08	Lou Philippon	TW/A08
Dave Best	TW/A08	Mike Powell	TW/A08
Walt Colby	TW/A08	Carol Reid	TW/A08
Peter Conklin	TW/A08	Marilyn Ressler	TW/A08
Marion Dancy	TW/A08	Tom Sherman	TW/A08
Bernie Lacroute	TW/A08	Ed Slaughter	TW/A08
Don McInnis	TW/A08	Mike Torla	TW/A08
Kathie Norris	TW/A08	Ed Wargo	TW/A08

MSD ENGINEERING

Don Ames	TW/A02	Ed McHugh	TW/E07
Joe Carchidi	TW/D08	Dave Potter	TW/C04
Dave Cutler	TW/D08	Wayne Rosing	TW/C03
Sas Durvasula	TW/C04	Steve Rothman	TW/D06
Tom Eggers	TW/C04	Dave Rodgers	TW/C04
John Gilbert	TW/E07	Bob Stewart	TW/C04
Al Helenius	TW/C04	Bill Strecker	TW/A08
Dick Hustvedt	TW/D08	Jim O'Loughlin	TW/E07
Steve Jenkins	TW/C04	Peter van Roekens	TW/E07
Jud Leonard	TW/C04	Linda Wright	TW/E07

[redacted]

VAX11 780



SYSTEMS AND OPTIONS SUMMARY

APRIL 1978

digital

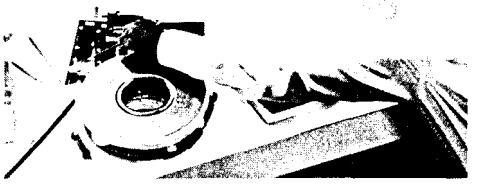
Contents

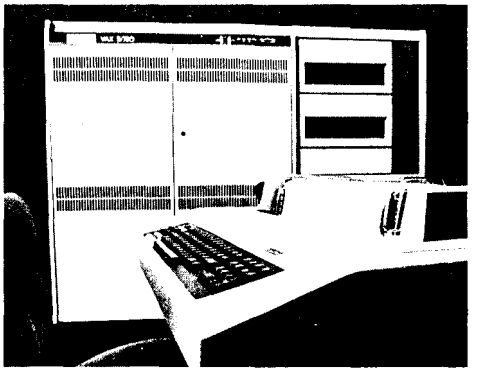
This summary was designed, produced and typeset
by DIGITAL's Sales Support Literature Group
using an In-House text-processing system
operating on a DECSYSTEM-20.

Introduction to VAX-11/780 Systems

VAX-11/780 is DIGITAL's new 32-bit computer system designed for interactive environments and high throughput applications.

The three basic systems described in this product summary combine the powerful VAX/VMS virtual memory operating system with a selection of system disks and backup/load devices. Also included are descriptions and configuring information for VAX-11/780 add-on options.





LEGEND

System Code/Option Code
The first entry is the order number for the system/option, with 115 Vac, 60 cycle power. The second entry, shown immediately below in *italics* is used for 230 Vac, 50 cycle power.

The basic features and specifications of each system/option are included in the description. More complete hardware and software descriptions can be found in handbooks and Software Product Descriptions.

SU	System Unit. Unit of space in chassis for mounting pre-wired backplanes(s) which can accept Hex- or Quad-sized modules.
SU 1-2	Defined here as the first two system units in a BA11-K UNIBUS expander box.
SU 3-5	Defined here as the last three system units in a BA11-K UNIBUS expander box.
Hex slot	Space in pre-wired backplane which will accept a 15.604 inch (39.634cm) high module.
Quad slot	Space in pre-wired backplane which will accept a 10.437 inch (26.510cm) high module.
MBA	MASSBUS adapter

+5V Power Available/Drawn
The +5V current available in or drawn from the system.

System UNIBUS Loads Available/Drawn
The number of UNIBUS loads remaining on or drawn from the UNIBUS. There can be a total of 20 loads or 50 feet (15.2 meters) of UNIBUS cable before a UNIBUS repeater (DB11) is needed.

Mounting Code
The mounting code indicates how the option mounts into the system.

CAB	Cabinet mounted.
Dedicated CAB	Option is packaged in its own cabinet.
FS	Free standing unit.
TT	Table top unit.
PAN	Panel mounted. Front panel height is 10.5 inches (26.7cm).
SM PAN	Small panel. Front panel height is 5.25 inches (13.3cm).

Support Category
The software product includes in the license fee, at a minimum, the service specified for the category indicated:

A On-site installation, one year Software Performance Reporting (SPR) Service, remedial service within the first 90 days.

B One year SPR service.

Dual RK07 disk-based VAX-11/780 System

System Code
SV-AXHHA-LA
SV-AXHHA-LD

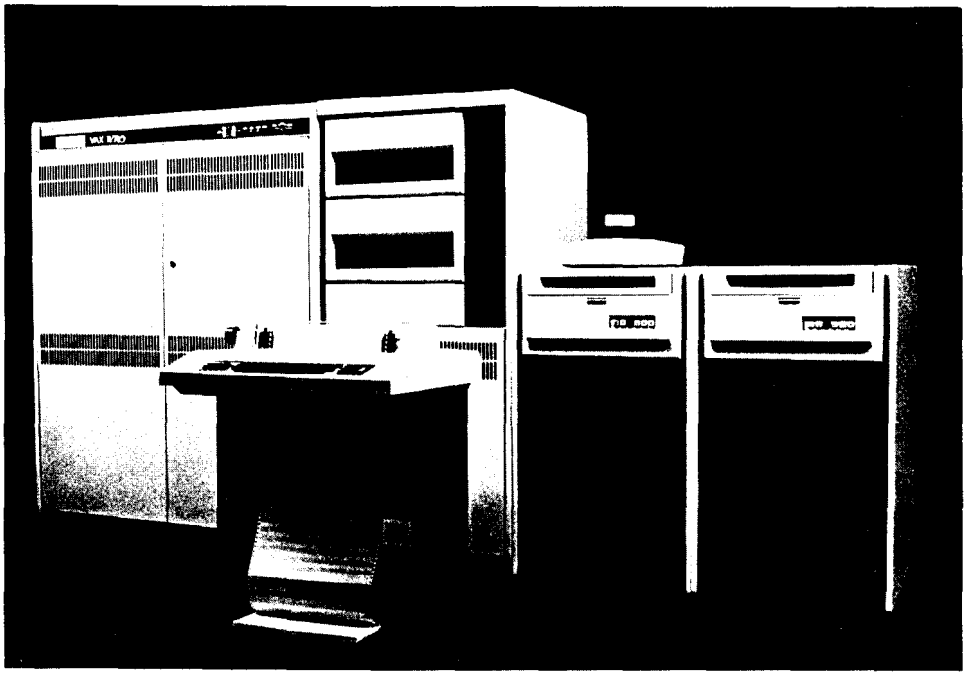
This VAX-11/780 interactive computer system, featuring dual RK07 disk-based storage devices, is designed to provide a high-performance, general-purpose, multiprogramming environment by combining DIGITAL's powerful VAX-11/780 central processor and the fully supported VAX/VMS operating system.

The system is configured with 256K bytes of ECC MOS memory, an RK711 UNIBUS controller with two top loading RK07 cartridge disk drives for a total of 56 megabytes of on-line storage, one DZ11-A asynchronous multiplexer providing eight EIA terminal lines, and an LA36 DECwriter II console terminal.

Basic equipment for this system includes the VAX-11/780 central processing unit with virtual memory management, bootstrap loader, standard instructions for packed decimal, floating and fixed point arithmetic, and character and string manipulations, 8K byte parity bipolar cache memory, high precision programmable real-time clock, time-of-year clock (with battery backup), and 12K bytes of writable diagnostic control store.

Also included as standard equipment is an integral diagnostic console subsystem, for use in both local and remote operations, which consists of an intelligent microcomputer (LSI-11 with 16K bytes read/write memory and 8K bytes read only memory) to which an RX01 floppy disk and the LA36 DECwriter II are connected.

This VAX-11/780 configuration is arranged in a 60"(H) x 48"(W) x 30"(D) (152.4cm x 122cm x 76.2cm) double-width high-boy cabinet with power supplies, two free-standing dedicated disk cabinets, and one single-width high-boy UNIBUS expansion cabinet which includes a BA11-K extension mounting box, one DD11-DK backpanel mounting unit, and a DZ11-A distribution panel.



EXPANSION CAPABILITY

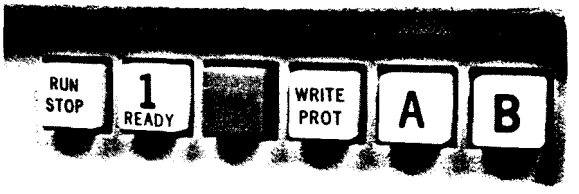
Expansion space for this system is available in three separate areas:

- Pre-designated space in the VAX-11/780 CPU Cabinet
- UNIBUS Expansion Cabinet
- VAX-11/780 Expansion Cabinet (not included)

CPU Cabinet

The CPU cabinet itself has pre-designated space available to accept the following options:

- FP780-AA(AB) high-performance floating-point accelerator with power supply.
- 12K bytes writable control store (KU780).
- An additional 768K bytes of ECC MOS memory.
- Memory battery backup (H7112) for up to one million bytes of memory.
- Serial line unit for remote diagnosis. (Remote diagnosis is only available to those customers under the terms and conditions of a current DIGITAL Field Service contract.)
- Plus, space for two MASSBUS adapters. (Adapter logic is bundled in with the REM03, REP05, REP06, and TEE16.)

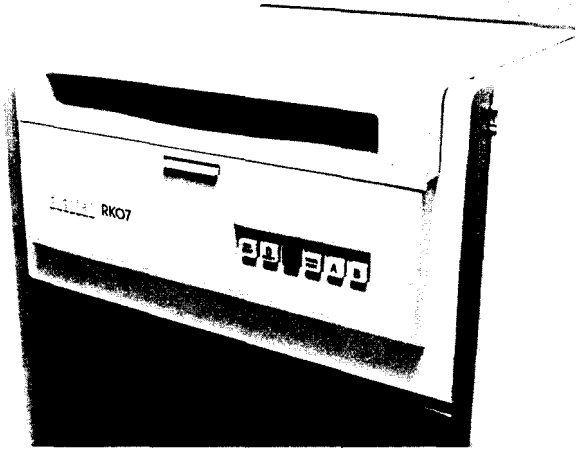


UNIBUS Expansion Cabinet

UNIBUS expansion specified for the VAX-11/780 systems can be added in the system's single-width, high-boy UNIBUS expansion cabinet in the same way that conventional UNIBUS options are added to PDP-11 computers. This cabinet can accept a total of two BA11-K extension mounting boxes and three DZ11 distribution panels.

For UNIBUS expansion beyond the limits of this cabinet, an H9602-DF(DH) "add-on" UNIBUS Options Cabinet may be ordered.

Please note, however, that only those UNIBUS options described in this summary are supported on VAX-11/780 systems.



Box	Expansion Space	+5V Power Available	UNIBUS Loads Available
BA11-K			18
SU 1 - 2	6 Hex slots, 2 Quad slots	22.8	
SU 3 - 5	2 Hex slots, 1 Quad slot, 1 SU	10.0	

VAX-11/780 Expansion Cabinet

A single-width, high-boy VAX-11/780 expansion cabinet, H9602-HA(HB), (not included with this system) is available and provides mounting space for an additional one million bytes of memory with control, space for two MASSBUS adapters, and one memory battery backup option (H7112-A/B).

RM03 disk/TE16 magnetic tape-based VAX-11/780 System

System Code
SV-AXTVA-LA
SV-AXTVA-LD

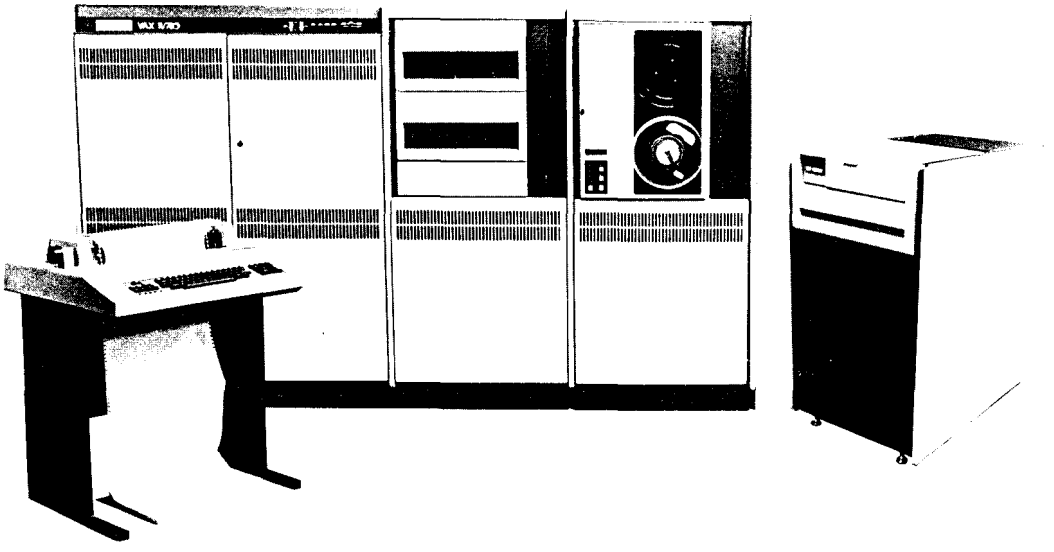
This VAX-11/780 interactive computer system, featuring RM03 disk and TE16 magnetic tape-based storage devices, is designed to provide a high-performance, general-purpose, multiprogramming environment by combining DIGITAL's powerful VAX-11/780 central processor and the fully supported VAX/VMS operating system.

The system is configured with 256K bytes of ECC MOS memory, an REM03 single access 67 million byte disk drive with MASSBUS adapter, a TEE16 magnetic tape transport unit (45 inches/second) with MASSBUS adapter, one DZ11-A asynchronous multiplexer providing eight EIA terminal lines, and an LA36 DECwriter II console terminal.

Basic equipment for this system includes the VAX-11/780 central processing unit with virtual memory management, bootstrap loader, standard instructions for packed decimal, floating and fixed point arithmetic, and character and string manipulations, 8K byte parity bipolar cache memory, high precision programmable real-time clock, time-of-year clock (with battery backup), and 12K bytes of writable diagnostic control store.

Also included as standard equipment is an integral diagnostic console subsystem, for use in both local and remote operations, which consists of an intelligent microcomputer (LSI-11 with 16K bytes read/write memory and 8K bytes read only memory) to which an RX01 floppy disk and the LA36 DECwriter II are connected.

This VAX-11/780 configuration is arranged in a 60"(H) x 48"(W) x 30"(D) (152.4cm x 122cm x 76.2cm) double-width high-boy cabinet with power supplies, one free-standing dedicated disk cabinet, one dedicated single-width high-boy tape transport cabinet, and one single-width high-boy UNIBUS expansion cabinet which includes a BA11-K extension mounting box, one DD11-DK backpanel mounting unit, and a DZ11-A distribution panel.



EXPANSION CAPABILITY

Expansion space for this system is available in three separate areas:

- Pre-designated space in the VAX-11/780 CPU Cabinet
- UNIBUS Expansion Cabinet
- VAX-11/780 Expansion Cabinet (not included)

CPU Cabinet

The CPU cabinet itself has pre-designated space available to accept the following options:

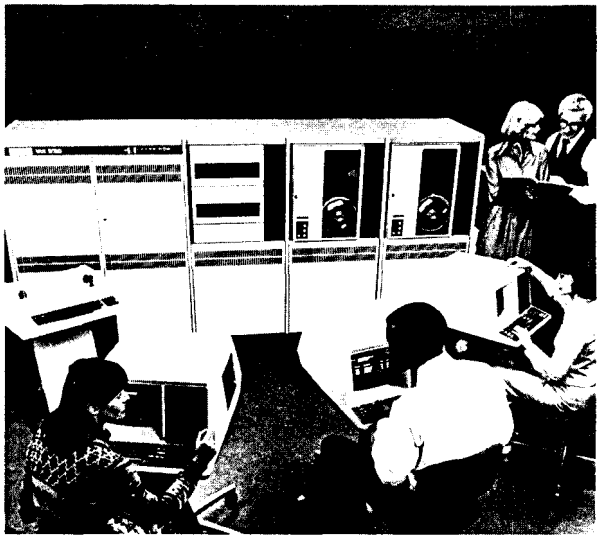
- FP780-AA(AB) high-performance floating-point accelerator with power supply.
- 12K bytes writable control store (KU780).
- An additional 768K bytes of ECC MOS memory.
- Memory battery backup (H7112) for up to one million bytes of memory.
- Serial line unit for remote diagnosis. (Remote diagnosis is only available to those customers under the terms and conditions of a current DIGITAL Field Service contract.)

UNIBUS Expansion Cabinet

UNIBUS expansion specified for the VAX-11/780 systems can be added in the system's single-width, high-boy UNIBUS expansion cabinet in the same way that conventional UNIBUS options are added to PDP-11 computers. This cabinet can accept a total of two BA11-K extension mounting boxes and three DZ11 distribution panels.

For UNIBUS expansion beyond the limits of this cabinet, an H9602-DF(DH) "add-on" UNIBUS Options Cabinet may be ordered.

Please note, however, that only those UNIBUS options described in this summary are supported on VAX-11/780 systems.



Box	Expansion Space	+5V Power Available	UNIBUS Loads Available
BA11-K			19
SU 1 - 2	6 Hex slots, 2 Quad slots	22.8	
SU 3 - 5	3 SUs	25.0	

VAX-11/780 Expansion Cabinet

A single-width, high-boy VAX-11/780 expansion cabinet, H9602-HA(HB), (not included with this system) is available and provides mounting space for an additional one million bytes of memory with control, space for two MASSBUS adapters, and one memory battery backup option (H7112-A/B).

RP06 disk/TE16 magnetic tape-based VAX-11/780 System

System Code
SV-AXCVA-LA
SV-AXCVA-LD

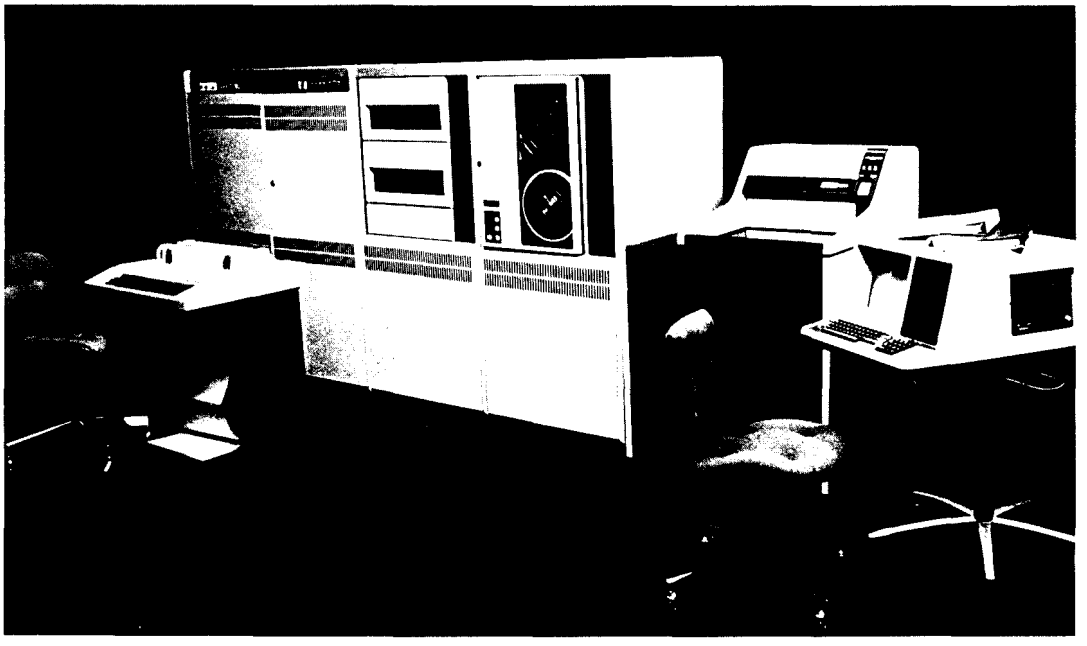
This VAX-11/780 interactive computer system, featuring RP06 disk and TE16 magnetic tape-based storage devices, is designed to provide a high-performance, general-purpose, multiprogramming environment by combining DIGITAL's powerful VAX-11/780 central processor and the fully supported VAX/VMS operating system.

The system is configured with 512K bytes of ECC MOS memory, an REP06 single access 176 million byte disk drive with MASSBUS adapter, a TEE16 magnetic tape transport unit (45 inches/second) with MASSBUS adapter, one DZ11-A asynchronous multiplexer providing eight EIA terminal lines, and an LA36 DECwriter II console terminal.

Basic equipment for this system includes the VAX-11/780 central processing unit with virtual memory management, bootstrap loader, standard instructions for packed decimal, floating and fixed point arithmetic, and character and string manipulations, 8K byte parity bipolar cache memory, high precision programmable real-time clock, time-of-year clock (with battery backup), and 12K bytes of writable diagnostic control store.

Also included as standard equipment is an integral diagnostic console subsystem, for use in both local and remote operations, which consists of an intelligent microcomputer (LSI-11 with 16K bytes read/write memory and 8K bytes read only memory) to which an RX01 floppy disk and the LA36 DECwriter II are connected.

This VAX-11/780 configuration is arranged in a 60"(H) x 48"(W) x 30"(D) (152.4cm x 122cm x 76.2cm) double-width high-boy cabinet with power supplies, one free-standing dedicated disk cabinet, one dedicated single-width high-boy tape transport cabinet, and one single-width high-boy UNIBUS expansion cabinet which includes a BA11-K extension mounting box, one DD11-DK backpanel mounting unit, and a DZ11-A distribution panel.



EXPANSION CAPABILITY

Expansion space for this system is available in three separate areas:

- Pre-designated space in the VAX-11/780 CPU Cabinet
- UNIBUS Expansion Cabinet
- VAX-11/780 Expansion Cabinet (not included)

CPU Cabinet

The CPU cabinet itself has pre-designated space available to accept the following options:

- FP780-AA(AB) high-performance floating-point accelerator with power supply.
- 12K bytes writable control store (KU780).
- An additional 512K bytes of ECC MOS memory.
- Memory battery backup (H7112) for up to one million bytes of memory.
- Serial line unit for remote diagnosis. (Remote diagnosis is only available to those customers under the terms and conditions of a current DIGITAL Field Service contract.)

UNIBUS Expansion Cabinet

UNIBUS expansion specified for the VAX-11/780 systems can be added in the system's single-width, high-boy UNIBUS expansion cabinet in the same way that conventional UNIBUS options are added to PDP-11 computers. This cabinet can accept a total of two BA11-K extension mounting boxes and three DZ11 distribution panels.

For UNIBUS expansion beyond the limits of this cabinet, an H9602-DF(DH) "add-on" UNIBUS Options Cabinet may be ordered.

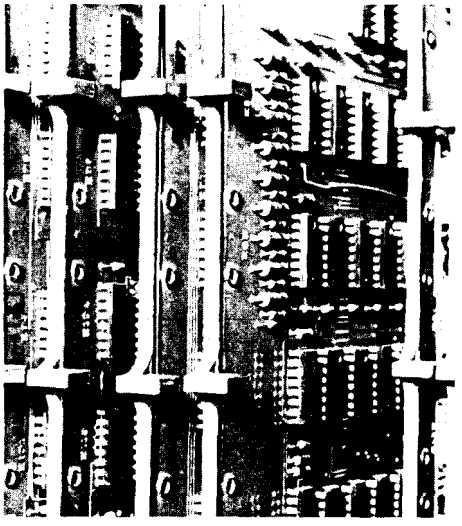
Please note, however, that only those UNIBUS options described in this summary are supported on VAX-11/780 systems.

Box	Expansion Space	+5V Power Available	UNIBUS Loads Available
BA11-K			19
SU 1 - 2	6 Hex slots, 2 Quad slots	22.8	
SU 3 - 5	3 SUs	25.0	

VAX-11/780 Expansion Cabinet

A single-width, high-boy VAX-11/780 expansion cabinet, H9602-HA(HB), (not included with this system) is available and provides mounting space for an additional one million bytes of memory with control, space for two MASSBUS adapters, and one memory battery backup option (H7112-A/B).

VAX-11/780 Processor & Memory Options



PROCESSOR OPTIONS

Please note that there is adequate power and prewired mounting space in the CPU cabinet to add the following options to VAX-11/780 systems.

FP780-AA
FP780-AB

High-performance floating-point accelerator for single- and double-precision floating-point instructions plus POLY, EMOD and MULL. Power supply is also included.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
Prewired CPU slots	N/A	N/A

KU780

12K byte writable control store. Please note that this option is not supported by VAX-11 system software.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
Prewired CPU slots	N/A	N/A

EXPANSION MEMORY

MS780-AA
MS780-AB

128K byte ECC MOS memory with controller. Expandable to a total of one million bytes with the addition of seven MS780-BAs or other combinations of expansion memories listed below. This option must be ordered for expansion beyond one million bytes of CPU cabinet-mounted memory. (Please note that one MS780-AA(AB) is included with each VAX-11/780 system to accommodate the first million bytes.)

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
H9602-HA(HB)	N/A	N/A

MS780-BA

128K byte ECC MOS expansion memory.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
MS780-AA(AB)	N/A	N/A

MS780-BB

256K byte ECC MOS expansion memory.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
MS780-AA(AB)	N/A	N/A

MS780-BC

512K byte ECC MOS expansion memory.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
MS780-AA(AB)	N/A	N/A

H7112-A
H7112-B

MOS memory battery backup. Powers up to one million bytes of memory for at least 10 minutes, or less memory for longer than 10 minutes.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
VAX-11/780 CPU CAB or H9602-HA(HB)	N/A	N/A

VAX-11/780 MASSBUS Options

VAX-11/780 EXPANSION CABINET

H9602-HA
H9602-HB

Single-width, high-boy expansion cabinet. 60"(H) x 28"(W) x 30"(D) (152.4cm x 71.2cm x 76.2cm). Designed to provide mounting space for up to one million bytes of memory with control, one memory battery backup option (H7112-A/B), and space for two MASSBUS adapters (which are bundled in with the REM03, REP05, REP06, and TEE16).

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
FS Cabinet	N/A	N/A

VAX-11/780 MASSBUS PERIPHERALS

Note 1: Each VAX-11/780 system has adequate power available to add the following MASSBUS peripherals. For this reason, the power drawn by these options is stated as not applicable.

Note 2: Average access time is defined as the sum of the average seek time plus the average latency. All average access times given below are stated as worst case. All capacities are formatted.

The nominal positioning time for the following disk drives is 28 msec. However, the positioning time for any disk drive varies slightly in a statistical distribution around the nominal value. Therefore, the worst case average positioning time is stated at 30 msec.

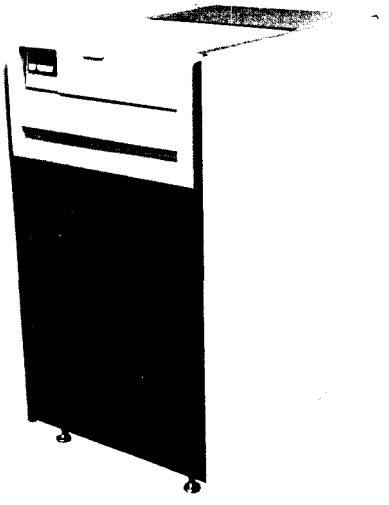
Disk Pack Subsystems

The dual access capability of disk subsystems is not supported by DIGITAL operating system software nor diagnostics.

REM03-AA
REM03-AD

Single-access 67 megabyte removable disk pack drive and VAX-11/780 MASSBUS adapter. Expandable to a total of 8 single access RM03 drives. One RM03-P disk pack is included. 1.2 megabytes/second peak transfer rate, 38.3 msec average access time.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
FS Drive Predesignated MBA space in CPU CAB or H9602-HA(HB)	N/A	N/A



REM03-BA
REM03-BD

Dual-access 67 megabyte removable disk pack drive and two VAX-11/780 MASSBUS adapters. Expandable to a total of 8 dual access RM03 drives. One RM03-P disk pack is included. 1.2 megabytes/second peak transfer rate, 38.3 msec average access time.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
FS Drive 2 predesignated MBA spaces in CPU CAB or H9602-HA(HB)	N/A	N/A

REP05-AA
REP05-AB

Single-access 88 megabyte removable disk pack drive and VAX-11/780 MASSBUS adapter. Expandable to a total of 8 single access RP drives (RP05, RP06). One RP04-P disk pack is included. 806K bytes/second peak transfer rate, 38.3 msec average access time. Field upgradable to the RP06.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
FS Drive Predesignated MBA space in CPU CAB or H9602-HA(HB)	N/A	N/A

REP05-BA
REP05-BB

Dual-access 88 megabyte removable disk pack drive and two VAX-11/780 MASSBUS adapters. Expandable to a total of 8 dual access RP drives (RP05, RP06). One RP04-P disk pack is included. 806K bytes/second peak transfer rate, 38.3 msec average access time. Field upgradable to the RP06.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
FS Drive 2 predesignated MBA spaces in CPU CAB or H9602-HA(HB)	N/A	N/A

REP06-AA
REP06-AB

Single-access 176 megabyte removable disk pack drive and VAX-11/780 MASSBUS adapter. Expandable to a total of 8 single access RP drives (RP05, RP06). One RP06-P disk pack is included. 806K bytes/second peak transfer rate, 38.3 msec average access time.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
FS Drive Pre-designated MBA space in CPU CAB or H9602-HA(HB)	N/A	N/A

REP06-BA
REP06-BB

Dual-access 176 megabyte removable disk pack drive and two VAX-11/780 MASSBUS adapters. Expandable to a total of 8 dual access RP drives (RP05, RP06). One RP06-P disk pack is included. 806K bytes/second peak transfer rate, 38.3 msec average access time.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
FS Drive 2 predesignated MBA spaces in CPU CAB or H9602-HA(HB)	N/A	N/A

Add-On Disk Pack Drives

RM03-AA
RM03-AD

Single-access 67 megabyte removable disk pack drive. One RM03-P disk pack is included. 1.2 megabytes/second peak transfer rate, 38.3 msec average access time.
Prerequisite: REM03-AA(AD) disk subsystem.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
FS Drive	N/A	N/A

RM03-BA
RM03-BD

Dual-access 67 megabyte removable disk pack drive. One RM03-P disk pack is included. 1.2 megabytes/second peak transfer rate, 38.3 msec average access time.
Prerequisite: REM03-BA(BD) disk subsystem.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
FS Drive	N/A	N/A

RP05-AA
RP05-AB

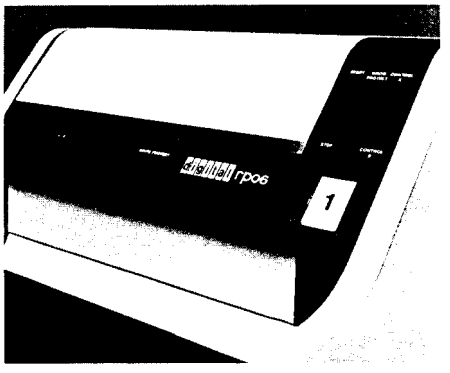
Single-access 88 megabyte removable disk pack drive. One RP04-P disk pack is included. 806K bytes/second peak transfer rate, 38.3 msec average access time. Field upgradable to the RP06.
Prerequisite: REP05-AA(AB) disk subsystem.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
FS Drive	N/A	N/A

RP05-BA
RP05-BB

Dual-access 88 megabyte removable disk pack drive. One RP04-P disk pack is included. 806K bytes/second peak transfer rate, 38.3 msec average access time. Field upgradable to the RP06.
Prerequisite: REP05-BA(BB) disk subsystem.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
FS Drive	N/A	N/A



RP06-AA
RP06-AB

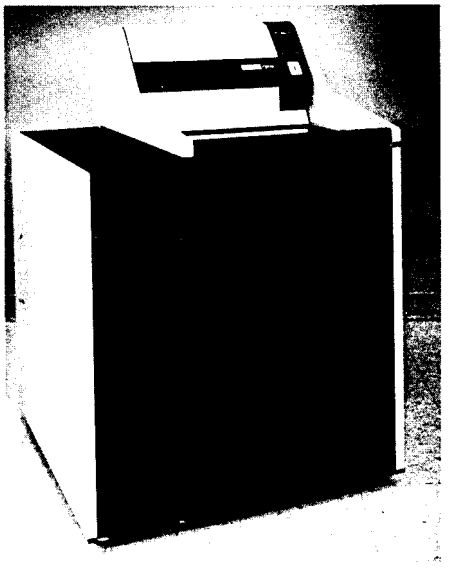
Single-access 176 megabyte removable disk pack drive. One RP06-P disk pack is included. 806K bytes/second peak transfer rate, 38.3 msec average access time.
Prerequisite: REP06-AA(AB) disk subsystem.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
FS Drive	N/A	N/A

RP06-BA
RP06-BB

Dual-access 176 megabyte removable disk pack drive. One RP06-P disk pack is included. 806K bytes/second peak transfer rate, 38.3 msec average access time.
Prerequisite: REP06-BA(BB) disk subsystem.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
FS Drive	N/A	N/A



Dual Access & Upgrade Options

The dual access capability of disk subsystems is not supported by DIGITAL operating system software nor diagnostics.

REM03-DA
REM03-DB

RM03 dual access conversion kit. Contains RM03-C, VAX-11/780 MASSBUS adpater and power supply to convert REM03-A to REM03-B.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
REM03-A	N/A	N/A

RM03-C

RM03 dual access kit containing drive logic and cables to convert RM03-A to RM03-B.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
RM03-A	N/A	N/A

REP05-DA
REP05-DB

RP05 dual access conversion kit. Contains RP05-C, VAX-11/780 MASSBUS adpater and power supply to convert REP05-A to REP05-B.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
REP05-A	N/A	N/A

RP05-C

RP05 dual access kit containing drive logic and cables to convert RP05-A to RP05-B.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
RP05-A	N/A	N/A

REP06-DA
REP06-DB

RP06 dual access conversion kit. Contains RP06-C, VAX-11/780 MASSBUS adpater and power supply to convert REP06-A to REP06-B.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
REP06-A	N/A	N/A

RP06-C

RP06 dual access kit containing drive logic and cables to convert RP06-A to RP06-B.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
RP06-A	N/A	N/A

RP06-U

RP05 to RP06 upgrade kit. Includes drive upgrade parts and RP06-P disk pack.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
RP05	N/A	N/A

Disk Packs

RM03-P

67 megabyte removable disk pack for RM03.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
RM03	N/A	N/A

RP04-P

88 megabyte removable disk pack for RP05.

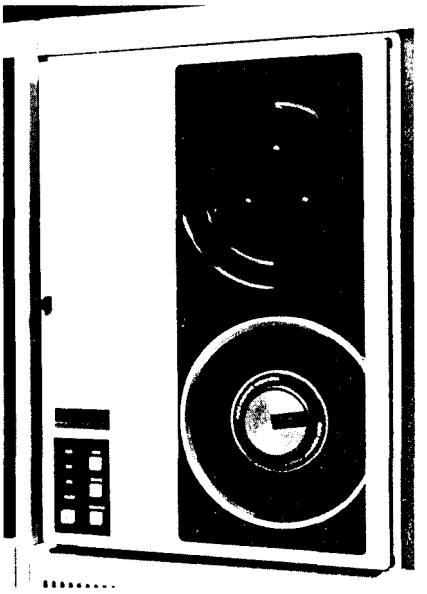
Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
RP05	N/A	N/A

RP06-P

176 megabyte removable disk pack for RP06.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
RP06	N/A	N/A

Magnetic Tape Subsystem



TEE16-AE
TEE16-AJ

Program selectable 800 or 1600 bpi, 9-track, 45 inches/second, magnetic tape transport and VAX-11/780 MASSBUS adapter. Industry compatible. Expandable to total of eight TE16 transports.

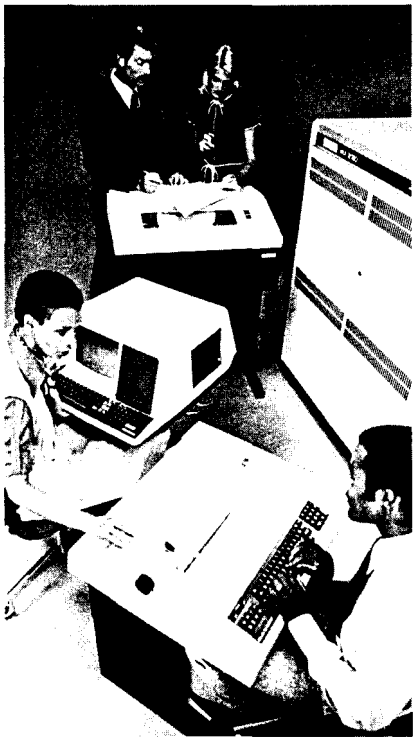
Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
Dedicated CAB plus pre-designated MBA space in CPU CAB or H9602-HA(HB)	N/A	N/A

Add-On Magnetic Tape Drive

TE16-AE
TE16-AJ

Program selectable 800 or 1600 bpi, 9-track, 45 inches/second, magnetic tape transport unit. Industry compatible.
Prerequisite: TEE16 magtape subsystem.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
Dedicated CAB	N/A	N/A



VAX-11/780 UNIBUS OPTIONS CABINET

H9602-DF
H9602-DH

Single-width, high-boy "add-on" UNIBUS expansion cabinet with single-phase power control. Provides space for an additional two BA11-K boxes and three DZ11 distribution panels. Backpanel mounting units must be ordered separately. 60"(H) x 28"(W) x 30"(D) (152.4cm x 71.2cm x 76.2cm).

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
FS Cabinet	0.0	0

VAX-11/780 EXTENSION MOUNTING BOX

BA11-KE
BA11-KF

Rack-mountable extension mounting box. Provides mounting space for five system units. SUs 1-2 together, and SUs 3-5 together, each have 25.0 amps of power available @+5V. One BA11-K is already included with each VAX-11/780 system.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
PAN space	50.0 total available	0

SYSTEM UNIT EXPANSION

DD11-CK

Backpanel mounting unit. Provides space for 2 Hex and 2 Quad slot modules.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
1 SU in BA11-K	N/A	N/A

DD11-DK

Backpanel mounting unit. Provides space for 7 Hex and 2 Quad slot modules. One DD11-DK is included with each VAX-11/780 system.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
2 SUs in BA11-K	N/A	N/A

BB11

Blank mounting panel for custom interface design and mounting system units.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
PAN space 1 SU in BA11-K	N/A	N/A

VAX-11/780 UNIBUS PERIPHERALS

Please note that average access time is defined as the sum of the average seek time plus the average latency.

Cartridge Disk Subsystems

The dual access capability of disk subsystems is not supported by DIGITAL operating system software nor diagnostics.

RK711-EA
RK711-EB

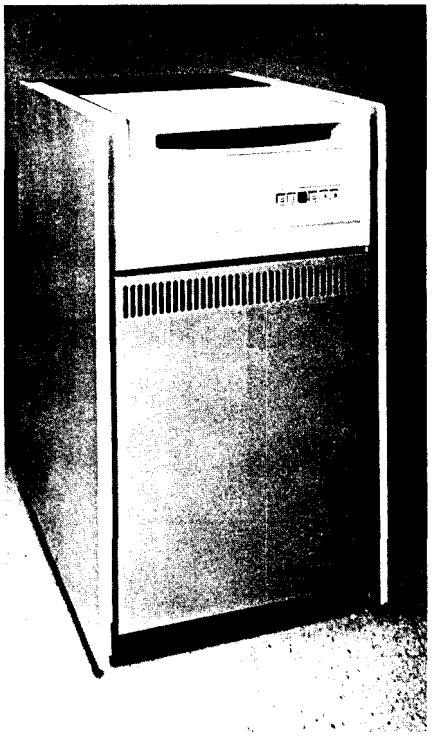
Single-access 28 megabyte disk drive and control unit. Expandable to a total of eight single access RK06 or RK07 drives. One RK07K-DC data cartridge is included. Average access time of 49.0 msec, peak transfer rate of 538K bytes per second.
NOTE: The RK711 controller requires 2 SUs of mounting space in a BA11-K and has 2 Hex slots and 1 Quad slot of additional UNIBUS expansion space.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
2 SUs FS Drive	15.0	1

RK711-FA
RK711-FB

Dual-access 28 megabyte disk drive and two control units. Expandable to a total of eight dual access RK06 or RK07 drives. One RK07K-DC data cartridge is included. Average access time of 49.0 msec, peak transfer rate of 538K bytes per second.
NOTE: Each RK711 controller requires 2 SUs of mounting space in a BA11-K and has 2 Hex slots and 1 Quad slot of additional UNIBUS expansion space.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
2 SUs per controller and FS Drive	15.0 per controller	1 per controller



RK611-EA
RK611-ED

Single access 14 megabyte disk drive and control unit. Expandable to a total of eight single access RK06 drives. One RK06K-DC data cartridge is included. Average access time of 50.5 msec, peak transfer rate of 538K bytes per second.
Note: The RK611 controller requires 2 SUs of mounting space in a BA11-K and has 2 Hex slots and 1 Quad slot of additional UNIBUS expansion space.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
2 SUs FS Drive	15.0	1

RK611-FA
RK611-FD

Dual access 14 megabyte disk drive and two control units. Expandable to a total of eight dual access RK06 drives. One RK06K-DC data cartridge is included. Average access time of 50.5 msec, peak transfer rate of 538K bytes per second.
Note: Each RK611 controller requires 2 SUs of mounting space in a BA11-K and has 2 Hex slots and 1 Quad slot of additional UNIBUS expansion space.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
2 SUs per controller and FS Drive	15.0 per controller	1 per controller

Add-On Cartridge Disk Drives

RK07-EA
RK07-EB

Single-access 28 megabyte disk drive. One RK07K-DC data cartridge is included. Average access time of 49.0 msec, peak transfer rate of 538K bytes per second.
Prerequisite: RK711-E or RK611-E subsystem.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
FS Drive	0.0	0

RK07-FA
RK07-FB

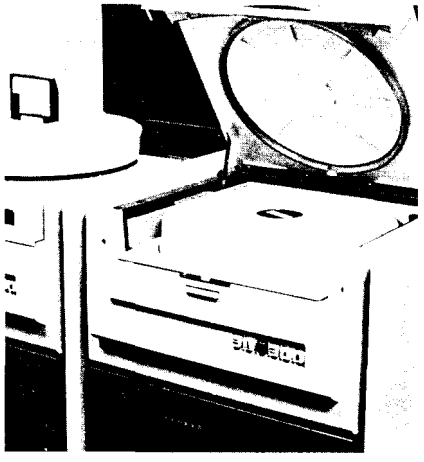
Dual-access 28 megabyte disk drive. One RK07K-DC data cartridge is included. Average access time of 49.0 msec, peak transfer rate of 538K bytes per second.
Prerequisite: RK711-F or RK611-F subsystem.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
FS Drive	0.0	0

RK06-EA
RK06-ED

Single access 14 megabyte disk drive. One RK06K-DC data cartridge is included. Average access time of 50.5 msec, peak transfer rate of 538K bytes per second.
Prerequisite: RK611-E controller.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
FS Drive	0.0	0



RK06-FA
RK06-FD

Dual access 14 megabyte disk drive. One RK06K-DC data cartridge is included. Average access time of 50.5 msec, peak transfer rate of 538K bytes per second.
Prerequisite: RK611-F controller.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
FS Drive	0.0	0

Dual Access Options

The dual access capability of disk subsystems is not supported by DIGITAL operating system software nor diagnostics.

RK711-C

Dual-access kit containing drive logic and hardware, one controller and cables to convert an RK711-E to an RK711-F.
Prerequisite: RK711-E.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
2 SUs for controller	15.0 for controller	1 for controller

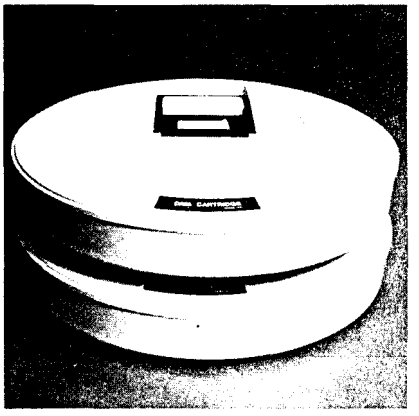
RK07-C

Dual-access kit containing drive logic, hardware and cables to convert an RK07-E to an RK07-F.
Prerequisite: RK07-E.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
RK07-E	0.0	0

RK611-C Dual access kit containing drive logic and hardware, one controller and cables to convert an RK611-E to an RK611-F.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
2 SUs for controller	15.0 for controller	1 for controller



RK06-C Dual access kit containing drive logic, hardware and cables to convert an RK06-E to an RK06-F.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
RK06-E	0.0	0

Cartridge Disks and Accessories

RK07K-EF Error free 28 megabyte data cartridge for RK07 subsystems.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
RK07	0.0	0

RK07K-AC 28 megabyte alignment cartridge for RK07.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
RK07	0.0	0

RK07K-DC 28 megabyte data cartridge for RK07 subsystems.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
RK07	0.0	0

RK06K-EF Error free 14 megabyte data cartridge for RK06 subsystems.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
RK06	0.0	0

RK06K-AC 14 megabyte alignment cartridge for RK06.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
RK06	0.0	0

RK06K-DC 14 megabyte data cartridge for RK06 subsystems.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
RK06	0.0	0

ASYNCHRONOUS MULTIPLEXERS (PROGRAMMED I/O)

DZ11-A

Asynchronous 8-line multiplexer for EIA/CCITT terminals or lines. Features programmable speeds (up to 9600 Baud) and formats on a per-line basis. Expandable to 16 lines. Includes data set control for use with Bell 103 or 113 modems or equivalent. BC05D cables are needed for modems. For local connect of EIA/CCITT terminals use BC03M-XX series of cables.
One DZ11-A is included with each VAX-11/780 system.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
Hex slot and SM PAN space	2.2	1

DZ11-B

Eight-line EIA/CCITT expansion multiplexer.
Prerequisite: DZ11-A.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
Hex slot	2.2	1

DZ11-C

Asynchronous 8-line multiplexer for 20mA current loop terminals. Features programmable speeds (up to 9600 Baud) and formats on a per-line basis. Expandable to 16 lines. Use BC04R-12 cables for Digital 20mA terminals.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
Hex slot and SM PAN space	3.0	1

DZ11-D

Eight-line 20mA current loop expansion multiplexer.
Prerequisite: DZ11-C.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
Hex slot	3.0	1

DZ11-E

Asynchronous 16-line multiplexer for EIA/CCITT terminals or lines. Features programmable speeds (up to 9600 Baud) and formats on a per-line basis. Includes data set control for use with Bell 103 and 113 modems or equivalent. BC05D cables are needed for modems. For local connect of EIA/CCITT terminals use BC03M-XX series of cables.

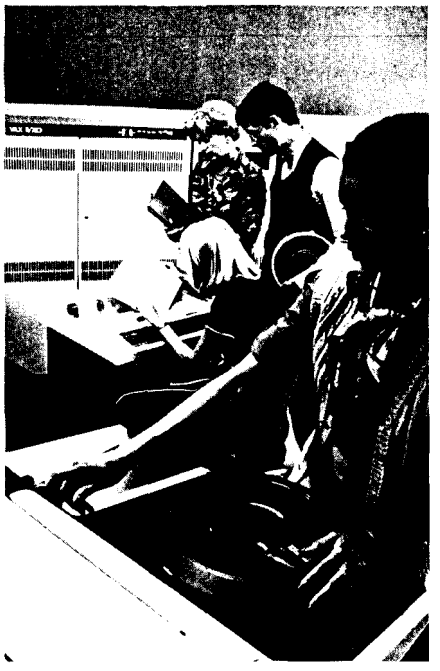
Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
2 Hex slots and SM PAN space	4.4	2

DZ11-F

Asynchronous 16 line multiplexer for 20mA current loop terminals. Features programmable speeds (up to 9600 Baud) and formats on a per-line basis. Use BC04R-12 cables for Digital 20mA terminals.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
2 Hex slots and SM PAN space	6.0	2

SINGLE LINE SYNCHRONOUS INTERFACES



DMC11-AR

Network link DDCMP microprocessor module (remote). DDCMP protocol implemented in firmware for remote operation. Operates full or half duplex. NPR input and output transfers.
Prerequisite: DMC11-DA on DMC11-FA line units.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
Hex slot	5.0	1

DMC11-DA

Network link remote line unit module. Interfaces to EIA/CCITT synchronous modems (Bell series 200 compatible) at speeds up to 19,200 bits/second. Operates full or half duplex. Includes data set control for switched network operations. Can be used to communicate over common carrier facilities to another DMC11 or to a synchronous interface with software implementation of DDCMP version 3.2. Includes 25 ft. (7.6m) modem cable.
Prerequisite: DMC11-AR.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
Hex slot (next to DMC11-AR)	3.0	0

DMC11-FA

Network link remote line unit module. Interfaces to CCITT V.35/DDS synchronous modems (Bell 500A L1/5 or equivalent) at speeds up to 250,000 bits/second. Includes data set control for full or half duplex, private wire operation. Can be used to communicate over common carrier facilities to another DMC11 or to a synchronous interface with software implementation of DDCMP version 3.2. Includes 25 ft. (7.6m) modem cable.
Prerequisite: DMC11-AR.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
Hex slot (next to DMC11-AR)	3.0	0

DMC11-AL

Network link DDCMP microprocessor module (local). DDCMP protocol is implemented in firmware for high speed NPR input and output transfers. One DMC11-AL operates at 1,000,000 bits/second in full duplex mode. Two DMC11-ALs operate at 1,000,000 bits/second in half duplex mode.
Prerequisite: DMC11-MA or DMC11-MD line units.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
Hex slot	5.0	1

DMC11-MA

Network Link local line unit module 1,000,000 bits/second. Provides high speed connection to another local DMC11 using coaxial cable up to 6,000 ft. (1,829m) long. (Includes built-in modem). Operates full duplex with two cables and half duplex with a single cable. Cables not included.
Prerequisite: DMC11-AL.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
Hex slot (next to DMC11-AL)	3.0	0

DMC11-MD

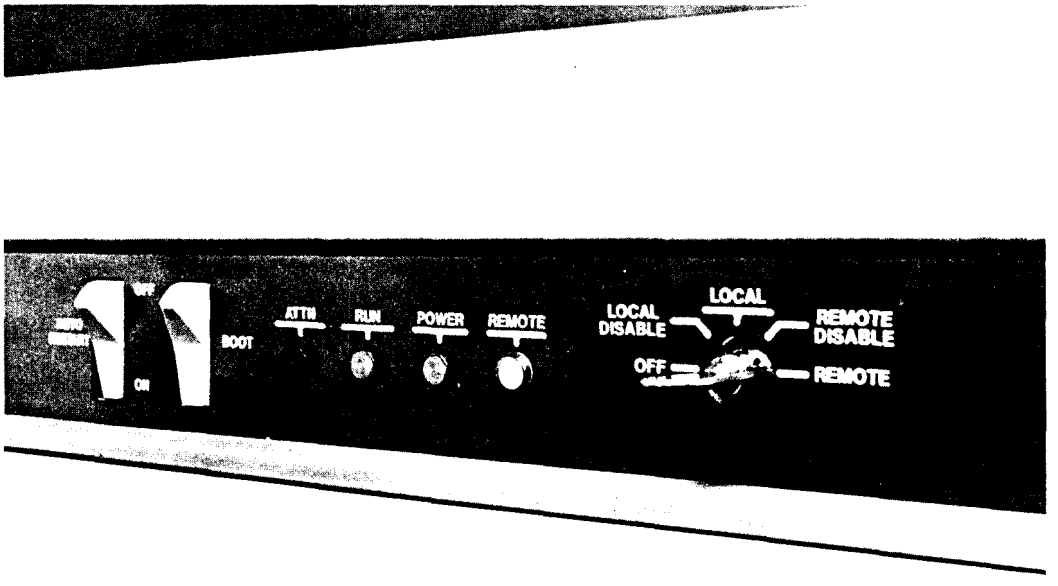
Network Link local line unit module 56,000 bits/second. Provides high speed connection to another local DMC11 using coaxial cable up to 18,000 ft. (5,487m) long. (Includes built-in modem). Operates full duplex with two cables and half duplex with a single cable. Cables not included.
Prerequisite: DMC11-AL.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
Hex slot (next to DMC11-AL)	3.0	0

BC03N-A0

100 ft. (30.5m) cable for DMC11 line units. (Use Belden cable type 8232 or equivalent for lengths greater than 100 ft. (30.5m). Refer to manual EK-DMCLU-MM-001 for cable connector details.)
Prerequisite: DMC11-MA or DMC11-MD.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
N/A	N/A	N/A



VAX-11/780 Input/Output Devices

LINE PRINTERS

LA11-PA
LA11-PD

132 column, 96 character matrix printer (DIGITAL's LA180 Line Printer) and control unit. 180 characters/second.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
1 Quad slot and FS	1.5	1

LP11-CA
LP11-CD

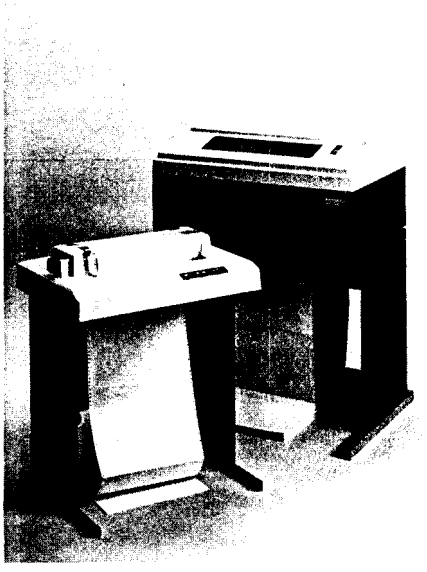
132 column, 64 character high speed printer and control unit. 900 lines/minute.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
1 Quad slot and FS	1.5	1

LP11-DA
LP11-DD

132 column, 96 character high speed printer and control unit. 660 lines/minute.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
1 Quad slot and FS	1.5	1



LP11-RA
LP11-RB

Heavy duty line printer and control unit. 132 columns, 64 characters, 1250 lines/minute.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
1 Quad slot and FS	1.5	1

LP11-SA
LP11-SB

Heavy duty line printer and control unit. 132 columns, 96 characters, 925 lines/minute.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
1 Quad slot and FS	1.5	1

LP11-VA
LP11-VD

132 column, 64 character printer and control unit. 300 lines/minute.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
1 Quad slot and FS	1.5	1

LP11-WA
LP11-WD

132 column, 96 character printer and control unit. 240 lines/minute.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
1 Quad slot and FS	1.5	1

LP11-YA
LP11-YD

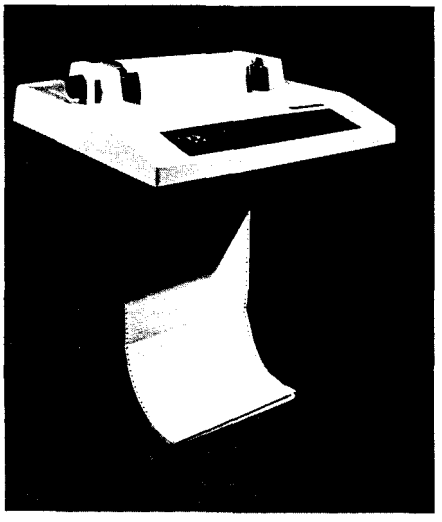
132 column, 64 character printer and control unit. 600 lines/minute.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
1 Quad slot and FS	1.5	1

LP11-ZA
LP11-ZD

132 column, 96 character printer and control unit. 436 lines /minute.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
1 Quad slot and FS	1.5	1



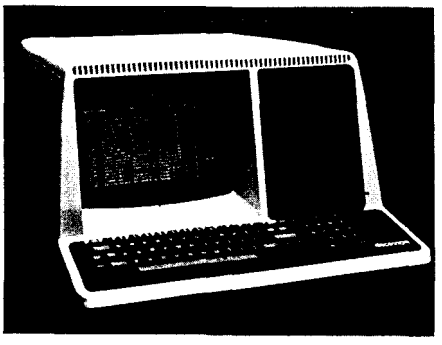
CARD READER

CR11
CR11-A

300 cards/minute reader and control unit. Reads 80-column punched cards.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
1 Quad slot and TT	1.5	1

TERMINALS



LA36-CE
LA36-CJ

DECwriter II hardcopy terminal with numeric keypad. 30 characters/second, 96 characters, with 20mA current loop interface.
Prerequisite: DZ11-C or DZ11-F.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
FS	N/A	N/A

LAXX-KG

EIA/CCITT adapter. Allows an LA36 to connect to an EIA/CCITT interface.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
N/A	N/A	N/A

VT52-AA
VT52-AB

Alphanumeric CRT terminal. Switch-selectable parity, 96-character keyboard, 80-column by 24-line display with cursor control. 20mA current loop interface.
Prerequisite: DZ11-C or DZ11-F.

Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
TT	N/A	N/A

VT52-AE
VT52-AF

Alphanumeric CRT terminal. Switch-selectable parity, 96-character keyboard, 80-column by 24-line display with cursor control. EIA/CCITT interface.
Prerequisite: DZ11-A or DZ11-E.

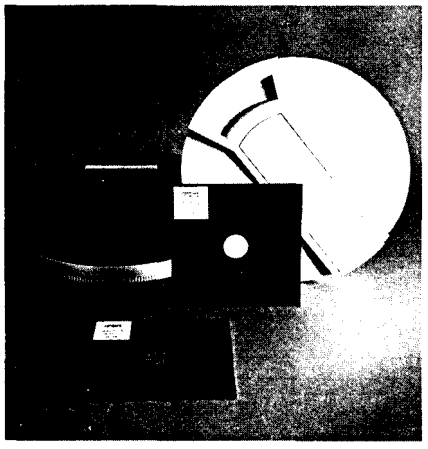
Mounting Code	+5V Power Drawn	UNIBUS Loads Drawn
TT	N/A	N/A

VAX-11 FORTRAN IV-PLUS

DESCRIPTION: VAX-11 FORTRAN IV-PLUS is an optimizing FORTRAN compiler designed to achieve high execution speed. It is based on the ANS FORTRAN X3.9-1966 standard. It's generated code takes advantage of the floating point and character instruction set and the VAX/VMS virtual memory system.

SOFTWARE COMPONENTS: VAX-11 FORTRAN IV-PLUS compiler.

MINIMUM SYSTEM REQUIREMENTS: Any valid VAX/VMS system.



Option Number	Distribution Medium	Support Category
QE100-AY	Floppy Disk	A

PDP-11 COBOL-74/VAX

DESCRIPTION: PDP-11 COBOL-74/VAX is a language processor for business data processing. It is based on the ANS X3.23-1974 standard. The COBOL-74/VAX compiler generates code for compatibility mode.

SOFTWARE COMPONENTS: COBOL compiler and run-time system, report generator and reformat utility programs.

MINIMUM SYSTEM REQUIREMENTS: Any valid VAX/VMS system that includes an LP11 series line printer.

Option Number	Distribution Medium	Support Category
QE101-AY	Floppy Disk	A

PDP-11 BASIC-PLUS-2/VAX

DESCRIPTION: PDP-11 BASIC-PLUS-2/VAX is a superset of the RSTS/E BASIC-PLUS, BASIC-11 IAS-RSX, and Dartmouth BASIC languages. It includes CALL statements, COM or COMMON statements, record I/O, and interactive debugging. The BASIC-PLUS-2/VAX compiler generates code for compatibiltity mode.

SOFTWARE COMPONENTS: BASIC-PLUS-2 compiler and run-time system.

MINIMUM SYSTEM REQUIREMENTS: Any valid VAX/VMS system.

Option Number	Distribution Medium	Support Category
QE102-AY	Floppy Disk	A

VAX/RSX-11 Development Package

DESCRIPTION: A set of software tools for the development and limited execution of RSX-11M/S task images.

SOFTWARE COMPONENTS: RSX-11M/S SYSGEN, FORTRAN IV/IAS-RSX compiler and run-time system.

MINIMUM SYSTEM REQUIREMENTS: Any valid VAX/VMS system.

Option Number	Distribution Medium	Support Category
QE103-AY	Floppy Disk	A

DECnet-VAX

DESCRIPTION: DECnet-VAX allows a suitably configured VAX/VMS system to participate as a Phase II DECnet node in point-to-point computer networks. DECnet-VAX offers task-to-task communications, network file transfer, and network resource sharing capabilities using the DIGITAL Network Architecture (DNA) protocols. DECnet-VAX communicates with adjacent nodes over synchronous communication lines.

SOFTWARE COMPONENTS: DECnet-VAX protocol manager.

MINIMUM SYSTEM REQUIREMENTS: Any valid VAX/VMS system with one of the following communication devices: DMC11-AR, -DA; DMC11-AL, -MD; or DMC11-AL -MA.

Option Number	Distribution Medium	Support Category
QED01-AY	Floppy Disk	A

Index

VAX-11/780 Systems & Options Summary
Insert for Prices & Index

VAX11 780

April 1978

PRICE

Purchase prices are stated in U.S. dollars, FOB DIGITAL plant, and apply only in the continental United States. Federal, state, and local taxes are not included. All prices and specifications are subject to change without notice. The fully supported systems described in the VAX-11/780 Systems & Options Summary, April 1978, include a Category A license for the operating system.

LICENSE FEE

The license fee for the software product on a single computer system, in U.S. dollars. Included are the binaries (and/or sources) on the specified medium, services, and documentation as specified in the DIGITAL Software Product Description.

FIELD SERVICE MONTHLY MAINTENANCE

This column lists the Field Service 8 hour/5 day monthly maintenance charge for a Basic Service Agreement. It includes all parts and labor required for system maintenance, plus scheduled preventive maintenance based on the specific needs of the equipment, and installation of engineering changes.

FIELD SERVICE INSTALLATION

This column lists the Field Service fixed price charge for performing field add-on installation of additional equipment to already installed systems.

PAGE

Refers to the corresponding page in the VAX-11/780 Systems & Options Summary, April 1978.

NOTE: (*) means to contact DIGITAL for prices.

digital

VAX11 780

System Code	Price($)	FS Monthly Maintenance	Page
VAX-11/780 SYSTEMS			
SV-AXCVA-LA(LD)	185,000	832	6
SV-AXHHA-LA(LD)	128,600	692	2
SV-AXTVA-LA(LD)	153,000	722	4

Option Number	Price($)	Field Maint	Serv Instl	Page
ADD-ON OPTIONS FOR VAX-11/780 SYSTEMS				
BA11-KE(KF)	2,420	16	120	14
BB11	187	N/C	100	14
BC03N-A0	121	N/C	N/A	20
CR11-(A)	6,170	53	280	22
DD11-CK	330	N/C	100	14
DD11-DK	660	N/C	100	14
DMC11-AL	1,520	13	175	19
DMC11-AR	1,520	13	175	19
DMC11-DA	850	6	145	19
DMC11-FA	850	6	145	19
DMC11-MA	850	6	145	19
DMC11-MD	850	6	145	20
DZ11-A	2,310	25	195	18
DZ11-B	1,710	21	155	18
DZ11-C	2,310	25	195	18
DZ11-D	1,710	21	155	18
DZ11-E	3,740	46	275	18
DZ11-F	3,740	46	275	18
FP780-AA(AB)	9,900	45	335	8
H7112-A(B)	1,145	10	343	8
H9602-DF(DH)	2,300	N/C	N/A	14
H9602-HA(HB)	3,900	N/C	N/A	9
KU780	10,000	50	260	8
LA36-CE(CJ)	2,100	19	125	22
LAXX-KG	65	N/C	110	22
LA11-PA(PD)	3,770	55	125	21
LP11-CA(CD)	24,000	185	260	21
LP11-DA(DD)	25,700	185	260	21
LP11-RA(RB)	38,470	185	290	21
LP11-SA(SB)	42,900	185	290	21
LP11-VA(VD)	11,800	95	260	21
LP11-WA(WD)	14,050	95	260	22
LP11-YA(YD)	18,900	108	260	22
LP11-ZA(ZD)	20,500	108	260	22
MS780-AA(AB)	22,500	70	338	8
MS780-BA	8,000	30	281	8
MS780-BB	13,000	60	325	8
MS780-BC	22,000	120	377	8
REM03-AA(AD)	25,000	170	865	9
REM03-BA(BD)	33,000	215	1,110	9
REM03-DA(DB)	8,000	45	700	11
REP05-AA(AB)	40,950	220	1,145	10
REP05-BA(BB)	53,550	270	1,165	10
REP05-DA(DB)	14,700	50	1,165	12

Option Number	Price($)	Field Maint	Serv Instl	Page
REP06-AA(AB)	44,000	220	1,145	10
REP06-BA(BB)	56,600	270	1,165	10
REP06-DA(DB)	14,700	50	1,165	12
RK611-EA(ED)	11,500	108	380	15
RK611-FA(FD)	19,000	148	405	15
RK611-C	10,450	40	285	17
RK06-EA(ED)	7,500	78	335	16
RK06-FA(FD)	11,000	78	335	16
RK06-C	3,850	10	175	17
RK06K-AC	995	N/A	N/A	17
RK06K-DC	249	N/A	N/A	17
RK06K-EF	349	N/A	N/A	17
RK711-EA(EB)	14,500	145	425	15
RK711-FA(FB)	22,000	190	495	15
RK711-C	10,450	45	295	16
RK07-EA(EB)	10,500	115	380	16
RK07-FA(FB)	14,000	130	405	16
RK07-C	3,850	15	250	16
RK07K-AC	1,295	N/A	N/A	17
RK07K-DC	325	N/A	N/A	17
RK07K-EF	425	N/A	N/A	17
RM03-AA(AD)	19,000	140	495	10
RM03-BA(BD)	21,000	155	680	11
RM03-C	2,000	15	435	11
RM03-P	595	N/A	N/A	12
RP04-P	600	N/A	N/A	12
RP05-AA(AB)	31,400	190	655	11
RP05-BA(BB)	36,540	210	900	11
RP05-C	5,150	20	900	12
RP06-AA(AB)	34,000	190	655	11
RP06-BA(BB)	39,140	210	900	11
RP06-C	5,150	20	900	12
RP06-P	750	N/A	N/A	12
RP06-U	10,500	N/A	N/A	12
TEE16-AE(AJ)	18,850	120	660	13
TE16-AE(AJ)	11,290	60	315	13
VT52-AA(AB)	1,900	20	125	23
VT52-AE(AF)	1,900	20	125	23

Option Number	License Fee($)	Page
LANGUAGES & UTILITIES		
QE100-AY	3,300	24
QE101-AY	7,700	24
QE102-AY	4,400	24
QE103-AY	1,500	25
QED01-AY	2,700	25

digital

DIGITAL EQUIPMENT CORPORATION, Corporate Headquarters: Maynard, Massachusetts 01754, Telephone: (617)897-5111—SALES AND SERVICE OFFICES: UNITED STATES—ALABAMA, Huntsville • ARIZONA, Phoenix and Tucson • CALIFORNIA, El Segundo, Los Angeles, Oakland, Ridgecrest, San Diego, San Francisco (Mountain View), Santa Ana, Santa Clara, Stanford, Sunnyvale and Woodland Hills • COLORADO, Englewood • CONNECTICUT, Fairfield and Meriden • DISTRICT OF COLUMBIA, Washington (Lanham, MD) • FLORIDA, Ft. Lauderdale and Orlando • GEORGIA, Atlanta • HAWAII, Honolulu • ILLINOIS, Chicago (Rolling Meadows) • INDIANA, Indianapolis • IOWA, Bettendorf • KENTUCKY, Louisville • LOUISIANA, New Orleans (Metairie) • MARYLAND, Odenton • MASSACHUSETTS, Marlborough, Waltham and Westfield • MICHIGAN, Detroit (Farmington Hills) • MINNESOTA, Minneapolis • MISSOURI, Kansas City (Independence) and St. Louis • NEW HAMPSHIRE, Manchester • NEW JERSEY, Cherry Hill, Fairfield, Metuchen and Princeton • NEW MEXICO, Albuquerque • NEW YORK, Albany, Buffalo (Cheektowaga), Long Island (Huntington Station), Manhattan, Rochester and Syracuse • NORTH CAROLINA, Durham/Chapel Hill • OHIO, Cleveland (Euclid), Columbus and Dayton • OKLAHOMA, Tulsa • OREGON, Eugene and Portland • PENNSYLVANIA, Allentown, Philadelphia (Bluebell) and Pittsburgh • SOUTH CAROLINA, Columbia • TENNESSEE, Knoxville and Nashville • TEXAS, Austin, Dallas and Houston • UTAH, Salt Lake City • VIRGINIA, Richmond • WASHINGTON, Bellevue • WISCONSIN, Milwaukee (Brookfield) • INTERNATIONAL—ARGENTINA, Buenos Aires • AUSTRALIA, Adelaide, Brisbane, Canberra, Melbourne, Perth and Sydney • AUSTRIA, Vienna • BELGIUM, Brussels • BOLIVIA, La Paz • BRAZIL, Rio de Janeiro and Sao Paulo • CANADA, Calgary, Edmonton, Halifax, London, Montreal, Ottawa, Toronto, Vancouver and Winnipeg • CHILE, Santiago • DENMARK, Copenhagen • FINLAND, Helsinki • FRANCE, Lyon, Grenoble and Paris • GERMAN FEDERAL REPUBLIC, Cologne, Frankfurt, Hamburg, Hannover, Munich, Nuremburg, Stuttgart and West Berlin • HONG KONG • INDIA, Bombay • INDONESIA, Djakarta • IRELAND, Dublin • ITALY, Milan, Rome and Turin • IRAN, Tehran • JAPAN, Osaka and Tokyo • MALAYSIA, Kuala Lumpur • MEXICO, Mexico City • NETHERLANDS, Utrecht • NEW ZEALAND, Auckland and Christchurch • NORWAY, Oslo • PUERTO RICO, Santurce • SINGAPORE • SPAIN, Madrid • SWEDEN, Gothenburg and Stockholm • SWITZERLAND, Geneva and Zurich • UNITED KINGDOM, Birmingham, Bristol, Epsom, Edinburgh, Leeds, Leicester, London, Manchester and Reading • VENEZUELA, Caracas •

COMPANY CONFIDENTIAL

FEB 29 1980

VENUS MACHINE STRUCTURE REVIEW: AGENDA

(FEB. 29, 1980 - CORPORATE AUDITORIUM, PK3-1)

1.	OVERVIEW	SAS DURVASULA	8:30 - 8:45
2.	RAMP/PERFORMANCE SYSTEM BALANCE/SIMULATIONS	JUD LEONARD	8:45 - 9:30
	COFFEE BREAK		9:30 - 9:45
3.	CPU PIPELINE STRUCTURE AND TRADEOFFS	AL HELENIUS	9:45 - 10:30
4.	I BOX	TOM KNIGHT	10:30 - 11:15
5.	E BOX	PAUL GUGLIELMI	11:15 - 12:15
	BUFFET LUNCH		12:15 - 1:15
6.	F BOX/CONSOLE	MIKE BROWN ED ANTON	1:15 - 2:00
7.	M BOX & MEMORY ARRAY	BILL BRUCKERT	2:00 - 2:45
	COFFEE BREAK		2:45 - 3:00
8.	ADAPTER BUS	JIM LACEY	3:00 - 3:30
9.	SBI ADAPTER	BARRY FLAHIVE	3:30 - 4:00
10.	FEED BACK & WRAP UP		4:00 - 4:30

SAS
2/28/80

OVERVIEW

SAS DURVASULA

WHAT ARE WE TRYING TO GET OUT OF THE REVIEW

1. REVIEW THE VENUS CPU/IO ADAPTER STRUCTURE WITH KEY TECHNICAL PEOPLE IN THE CORPORATION AND GET THEIR CRITIQUE TO CORRECT/IMPROVE WHAT WE ARE DESIGNING.

2. PRESENT THE RAMP FEATURES AND GET A FEEDBACK.

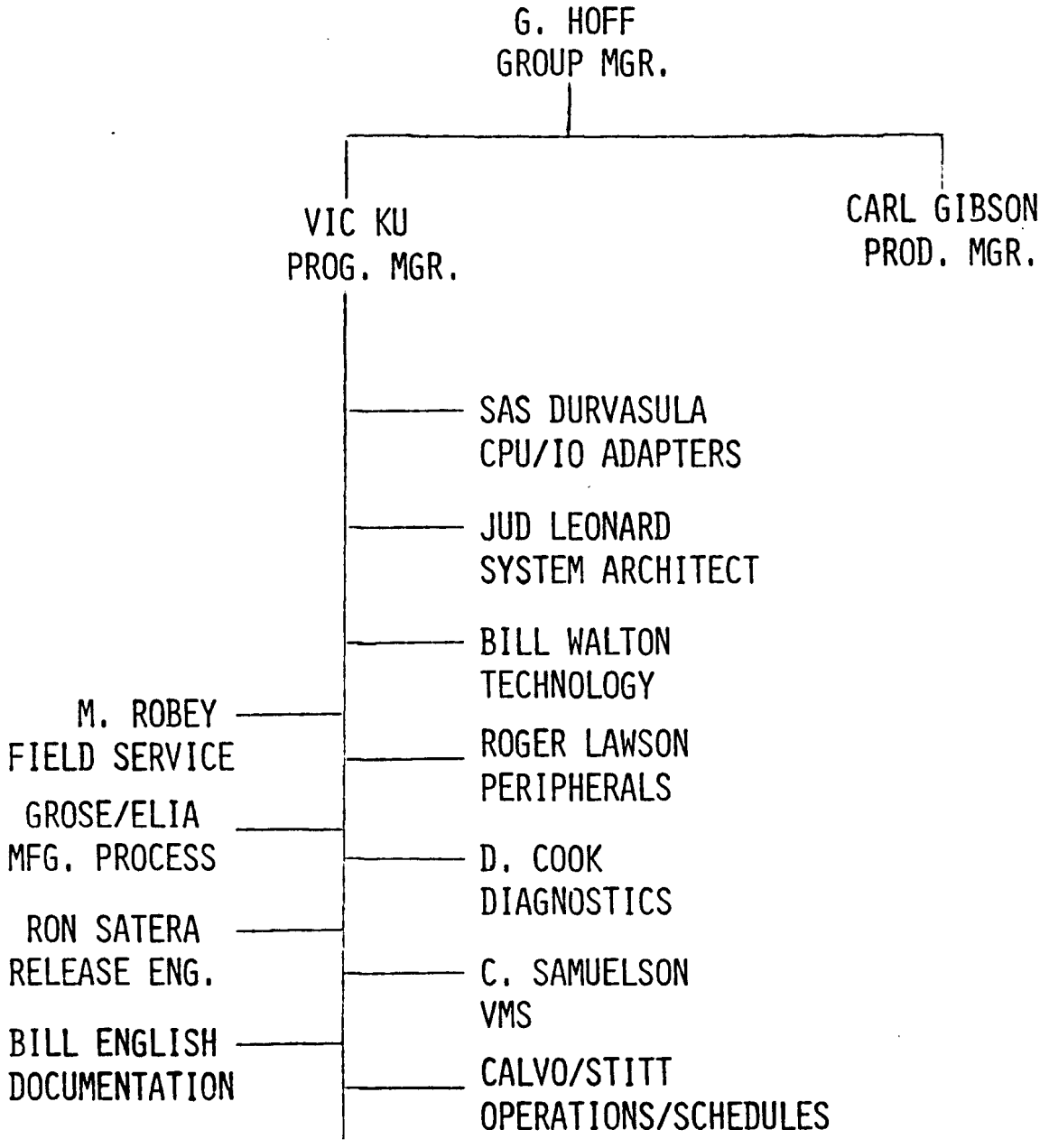
3. GIVE THE TEAM THE ADDED CONFIDENCE TO PROCEED WITH THE IMPLEMENTATION PHASE.

A NOTE OF CAUTION:

LET US NOT GET INTO GATE LEVEL
DISCUSSIONS OR TECHNOLOGY ISSUES
IN THIS MEETING.

SAS
2/28/80

VENUS PROGRAM ORGANIZATION



SAS
2/28/80

VENUS GOALS

1. FCS - (AUG. 82)

2. RAMP - BETTER THAN 11/780

3. XFER COST - SAME AS 11/780

4. PERFORMANCE - 3.5 X 11/780

SAS
2/28/80

MODULE COUNT & PARTS ESTIMATE: (8-15-79)

		MCA TYPES	TOTAL NO. OF MCAs	RAMS 1K & 4K	10K & OTHER	NO. OF MODULES
1.	I/E BOX AND CONTROL STORE	26	65	38 & 207	120	5
2.	M BOX & MEM. CONTROL	17	32	17 & 31	30	2
3.	CONSOLE	-	-	-	150	1
4.	FPA	13	53	-	-	2
5.	SBI A	-	-		(TTL/10K) (MIX)	2
6.	MEMORY ARRAY (1MB)					1

ASSUMPTIONS:

1. 8 LAYER P.C.B. WITH CONTROLLED IMPEDENCE (FOR MCA MODULES)
2. 30 MCA EQUIVALENTS/MODULE

SAS
2/28/80

VENUS SCHEDULE

SPECS PUBLISHED		NOV. 79
FIRST MCA INTO LAYOUT		NOV. 79
MACHINE PARTITIONING COMPLETED & MODULE COUNT FULLY UNDER-STOOD (90% CONFIDENCE)		APRIL 80
FIRST MODULE INTO LAYOUT		MAY 80
LAST MODULE INTO LAYOUT		NOV. 80
CONSOLE & SBI POWER ON		DEC. 80
M BOX POWER ON		JAN. 80
I/E BOXES (FULL B.B.) POWFR ON		FEB. 80
PROTO POWER ON		SEPT. 81
PILOTS		FEB. 82
FCS	(50%)	AUG. 82
	(90%)	MAY 83

SAS
2/28/80

RAMP/PERFORMANCE
SYSTEM BALANCE/SIMULATIONS

JUD LEONARD

	100% cache hit			90% cache hit		
Operation	11/780	VENUS	ratio	11/780	VENUS	ratio
ADDL2 R,R	400	67	6.0	520	132	3.9
ADDL2 ^B(R),R	800	133	6.0	1040	263	4.0
ADDL2 R,^B(R)	1200	267	4.5	1440	397	3.6
ADDL2 ^B(R),^B(R)	1400	267	5.2	1760	462	3.8
ADDF2 R,R	800	133	6.0	920	198	4.6
ADDF2 ^B(R),R	1200	200	6.0	1440	330	4.4
MOVL R,^B(R)	800	133	6.0	920	263	3.5
MOVL ^B(R),^B(R)	1000	200	5.0	1240	395	3.1
INCL R	400	67	6.0	520	132	3.9
INCL ^B(R)	1000	200	5.0	1240	330	3.8
ADDL3 R,R,R	600	133	4.5	720	198	3.6
ADDL3 ^B(R),^B(R),R	1200	200	6.0	1560	460	3.4
ADDL3 ^B(R),^B(R),^B(R)	1600	267	6.0	2080	592	3.5
BR successful	600	133	4.5	720	198	3.6
BR unsuccessful	400	67	6.0	520	132	3.9
MOVC3 per byte	300	33	9.0	650	108	6.0
MOVC5 clear per byte	300	33	9.0	424	71	6.0

VENUS RAMP

RELIABILITY -

INHERENT RELIABILITY ACHIEVED THROUGH LOW COMPONENT COUNT AND WORST-CASE DESIGN.

SELF-CHECKING LOGIC WITH REDUNDANT ENCODING (PARITY, ECC, OR DUPLICATION) IN ALL RAMS AND THROUGH MANY DATA PATHS.

AVAILABILITY -

RIDE-THROUGH STRATEGIES FOR MOST DETECTABLE ERRORS.

INSTRUCTION RETRY.

RAM RE-WRITE TO CORRECT BIT FAILURES.

HYDRA CONFIGURATIONS FOR EXTREME NEEDS.

MAINTAINABILITY -

DIAGNOSTIC CONSOLE PROCESSOR WITH RD CAPABILITY.

CONSOLE VISIBILITY TO ALL MCA'S.

ERROR LOGGING.

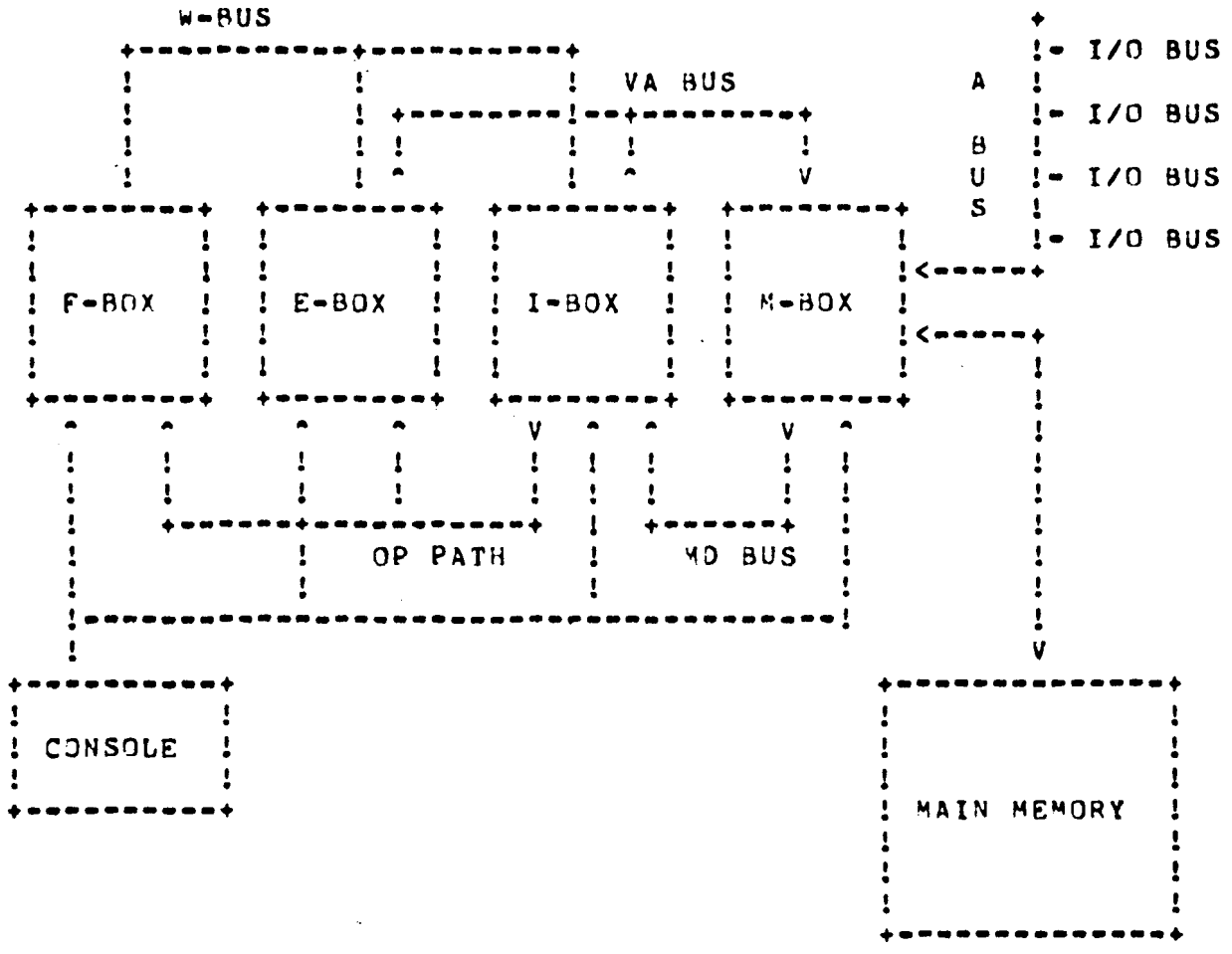
FIRST-FAILURE FAULT LOCALIZATION.

FAST, ACCURATE DIAGNOSTICS.

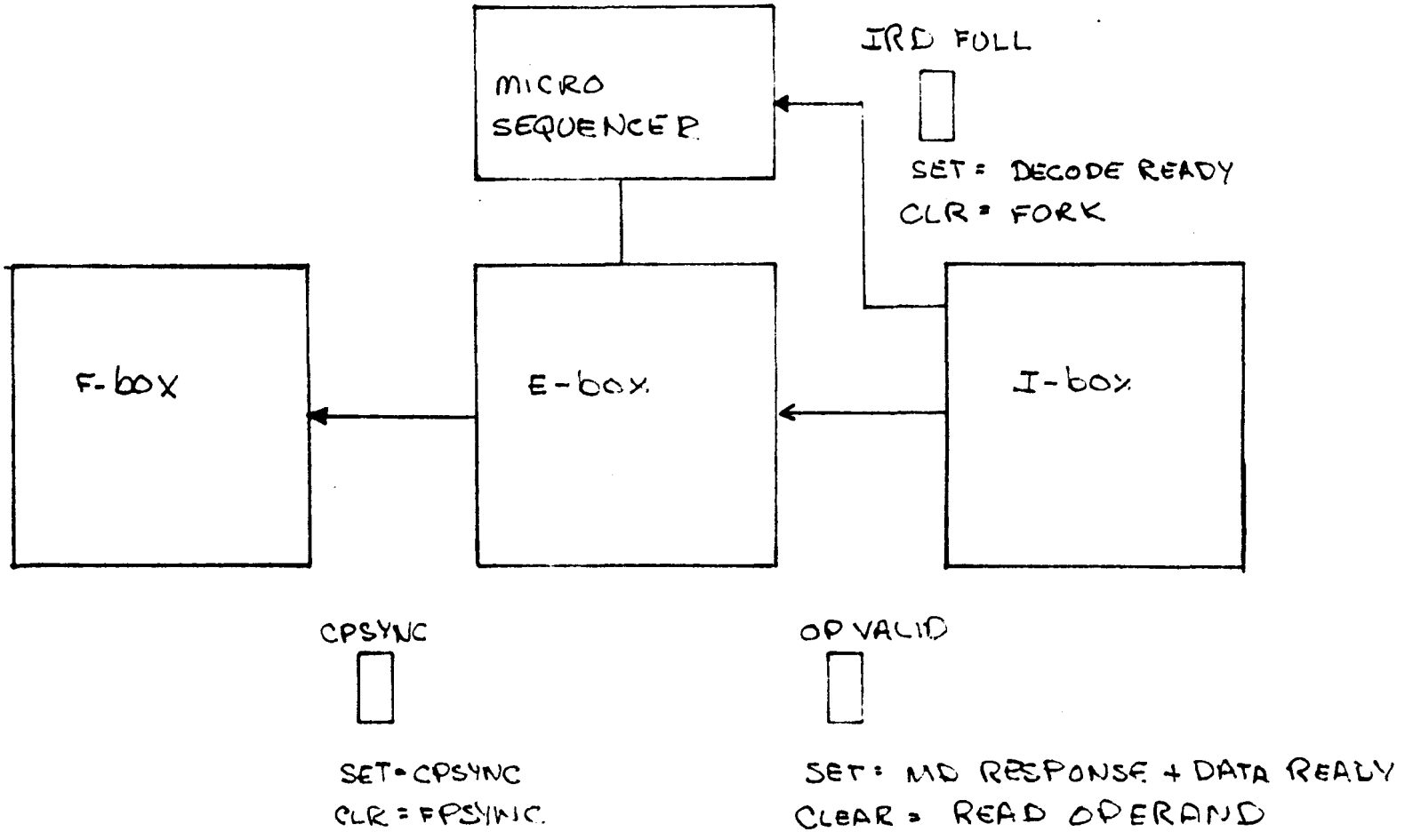
J. LEONARD
9/26/79

CPU PIPELINE STRUCTURE AND TRADEOFFS

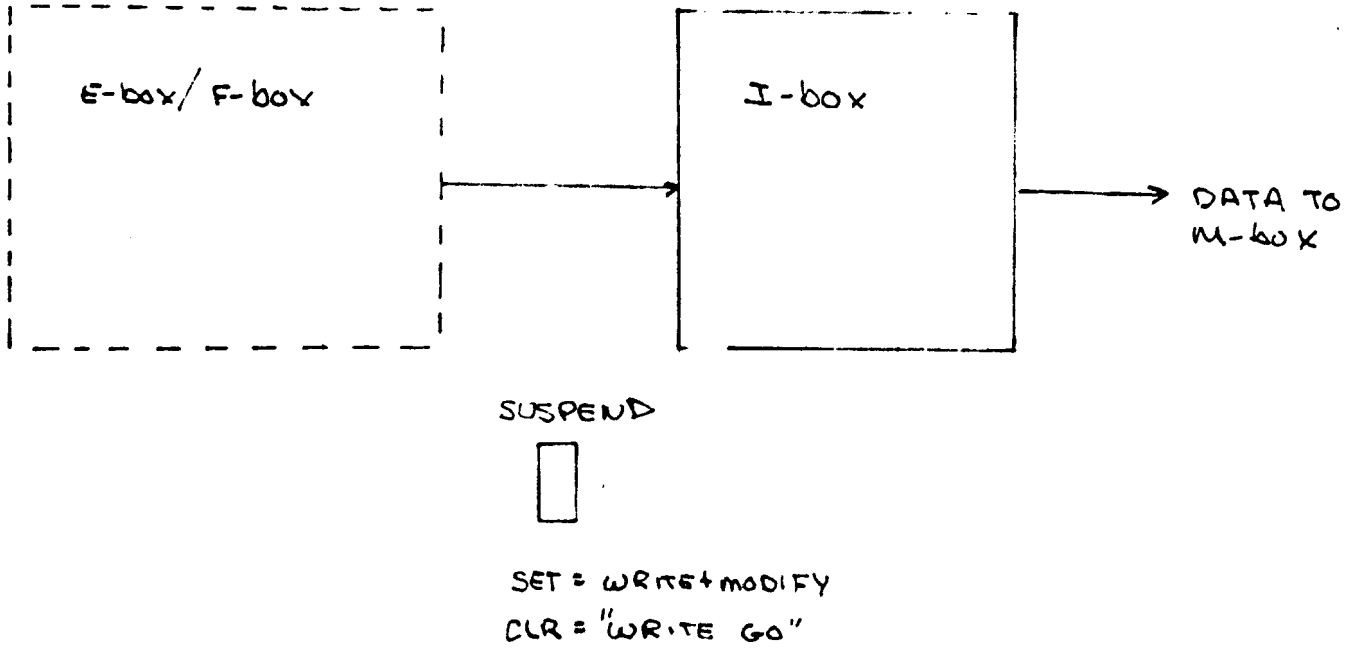
AL HELENIUS



VENUS BLOCK DIAGRAM



PIPE CONTROL FLOW



STORE FLOW

Pipeline tradeoffs

There are two basic approaches to pipline architectures:

- provide enough buffering at each stage to be able to assume data is ready
- keep the buffering down to a minimum (one)

The advantage to alternative one is that most of the time the pipe depth will hide events such as cache misses etc.

The disadvantage is that it is costly and complex to handle conflicts.

Restart time

Non useful work

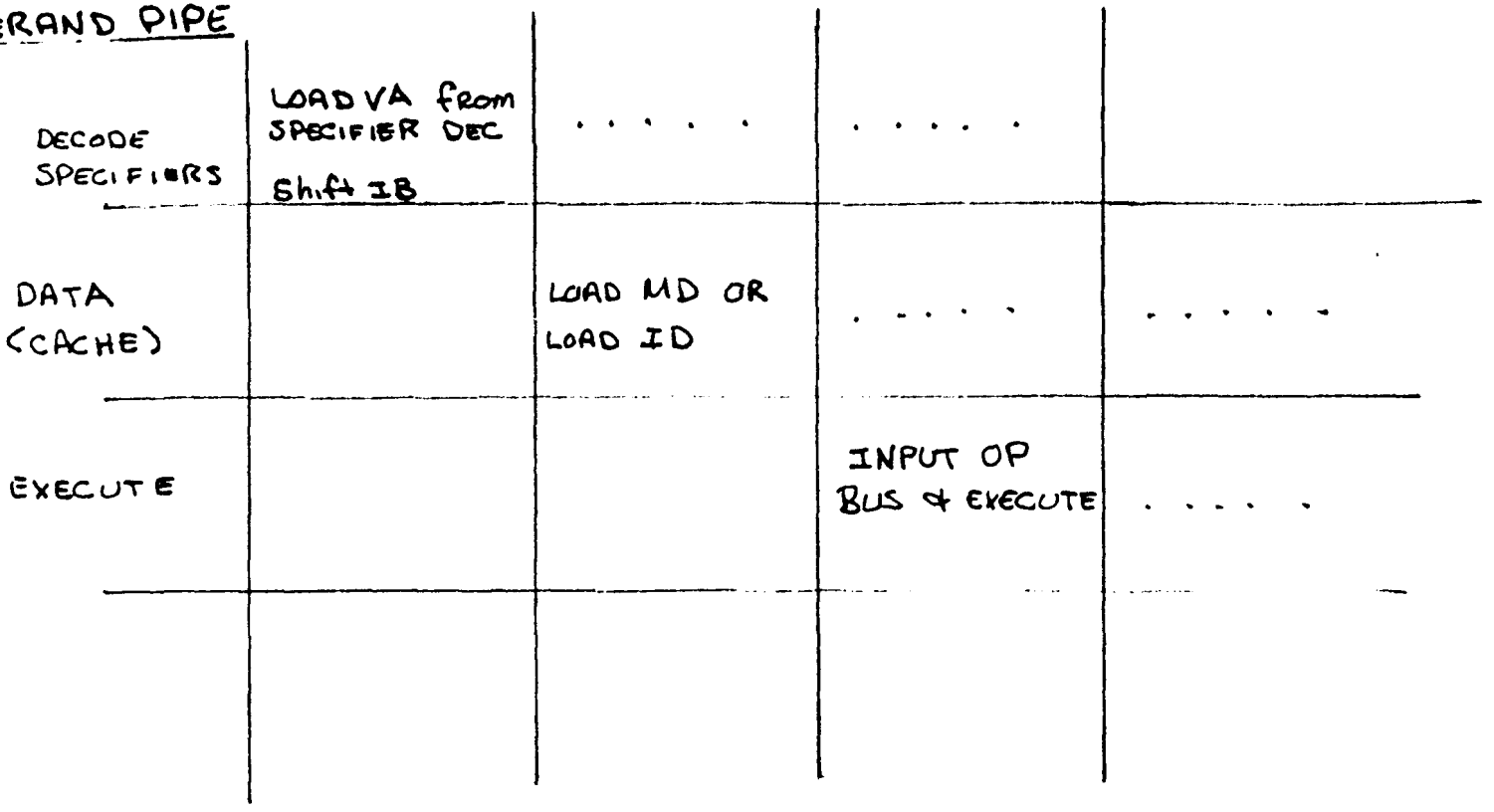
The advantage to alternative two is that it is shallow enough that it is reasonable to wait on a conflict and continue when it is resolved.

memory reads

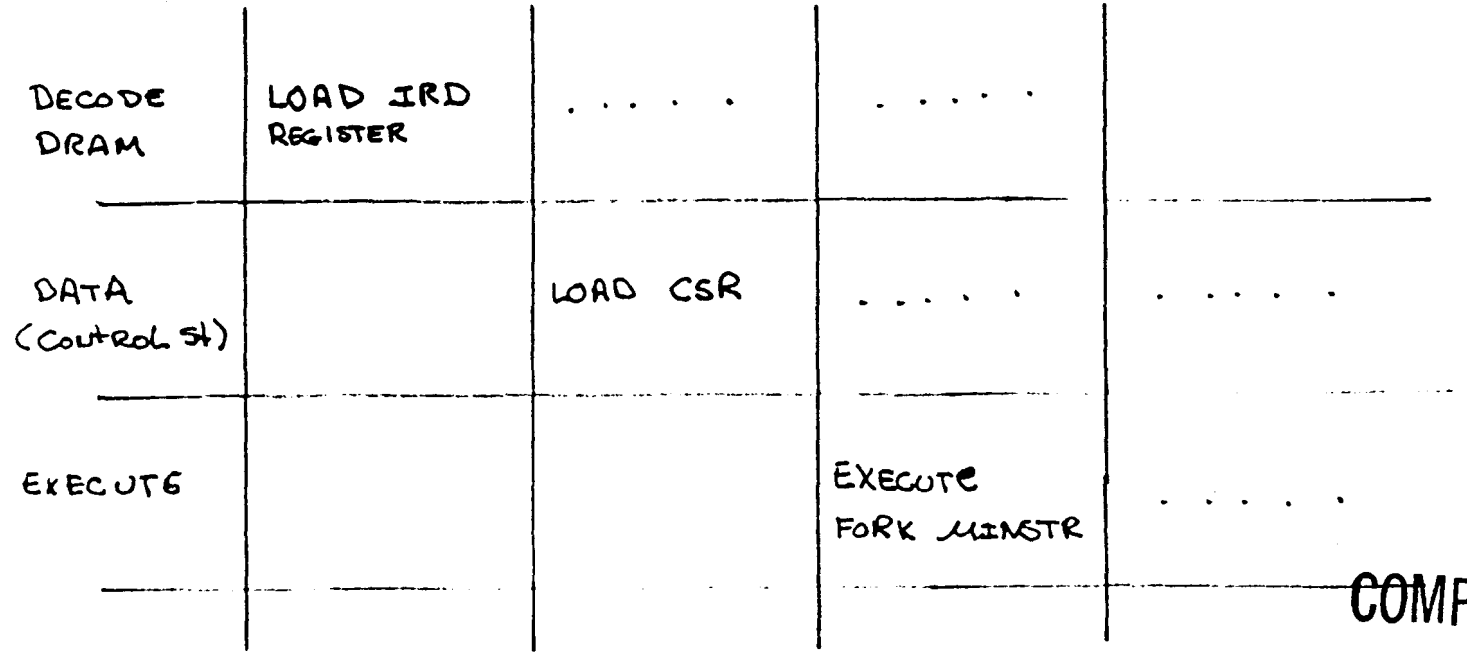
GPP reads

The disadvantage is performance

OPERAND PIPE



CONTROL STORE PIPE



Storage conflicts

Memory - none

Register - The I-box will save the register number and context of a register to be written by the execution unit. If during the next address calculation the same register number is used, the I-box will stall. This includes the case of R+1 and quad data type.

For H data type, string instructions and other implied register destinations, the I-box will suspend operation until E-box is done.

Control flags

Instruction decode:

IPD FULL - This bit indicates that the register which holds the output of the instruction decode is full. This means the I-box must wait for the e-box to remove this address. Otherwise, the i-box may run.

Operand passing:

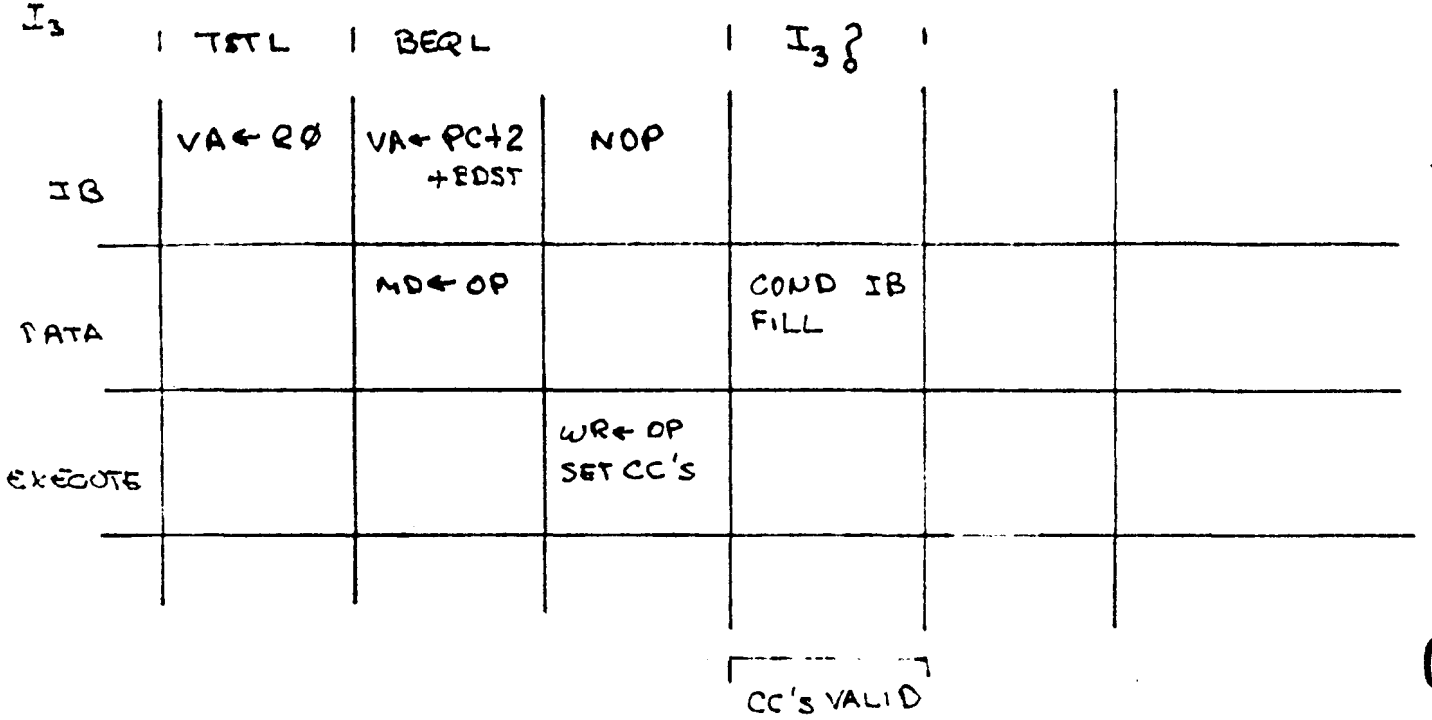
OP VALID - This flag is set when an operand becomes available to the execution units. This will cause the I-box to stop fetching operands until the data is removed by the execution unit.

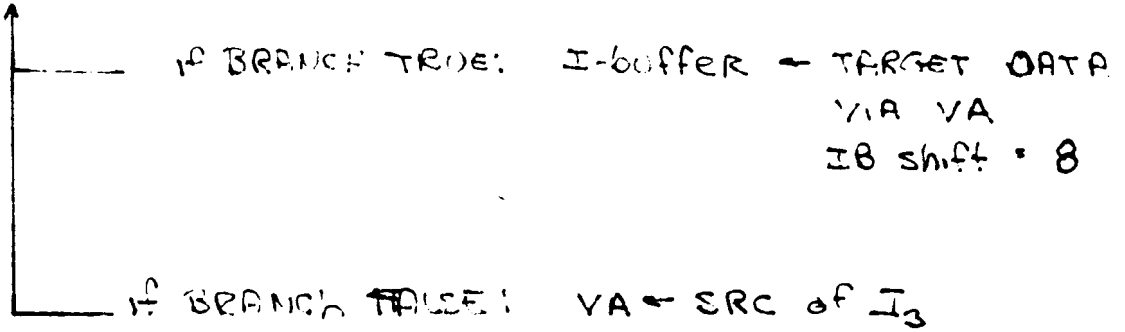
This same flag is used by the execution unit to stall if it wants data and the flag is not set.

Floating point control:

CPSYNC - This flag is set to indicate that the e-box has reached a control point and is waiting for the f-box. The e-box will stall until the f-box clears this flag by asserting FSYNC.

TSTL (R0)
BEQL TAG
I_3

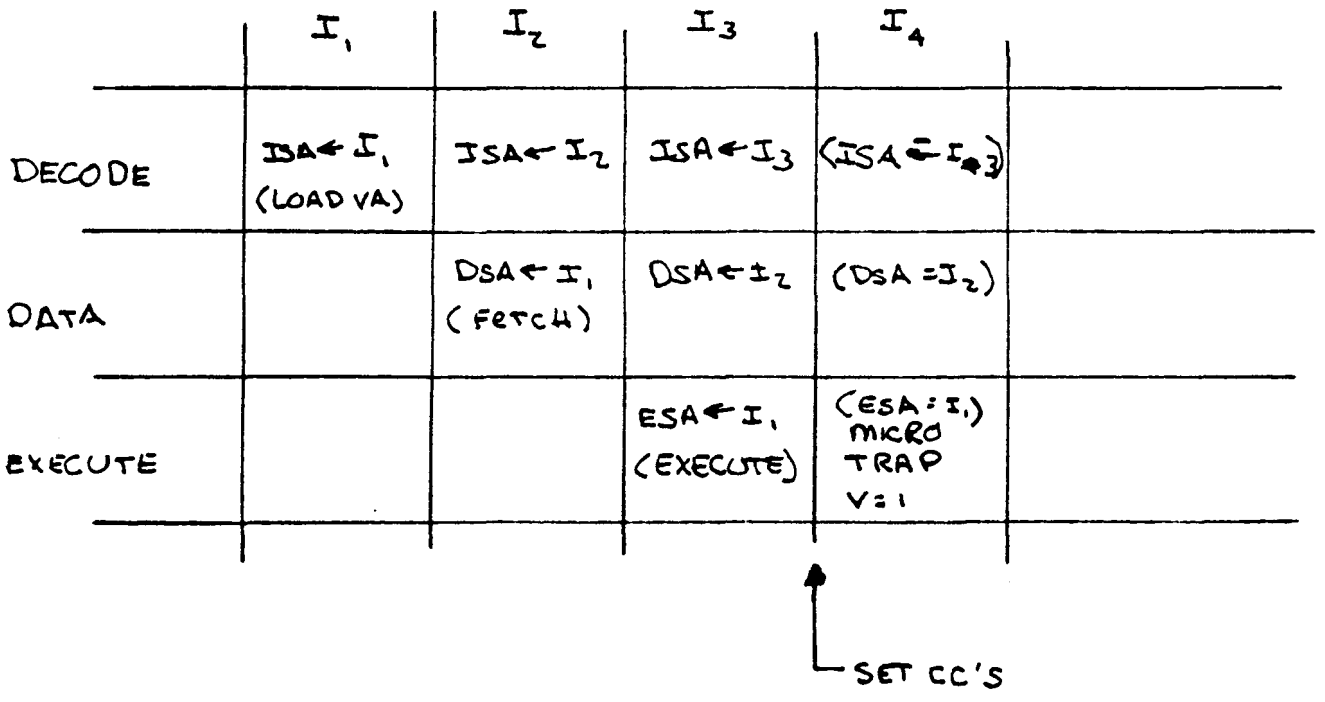




BR FALSE = 2 CYCLES

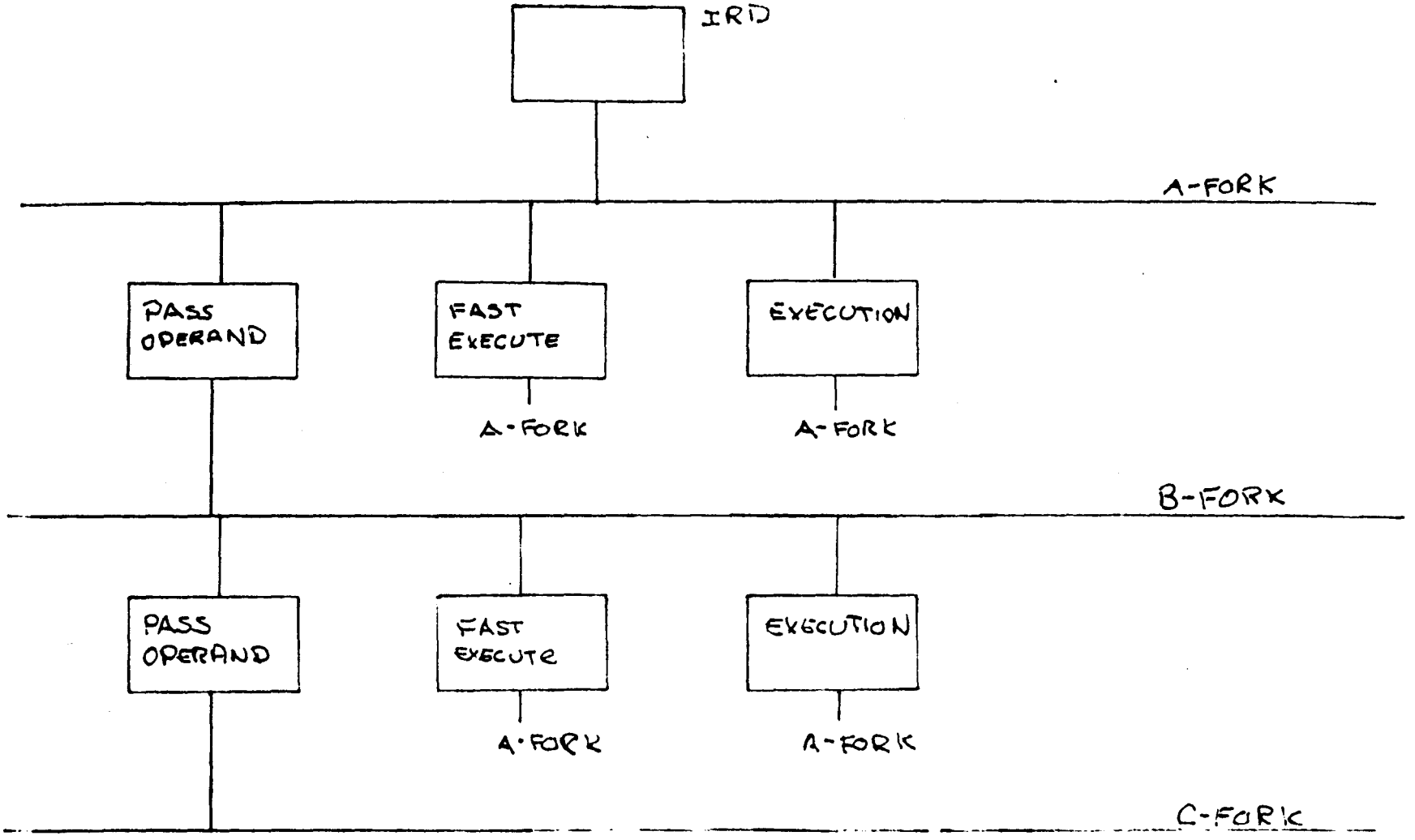
BR TRUE = 3 CYCLES *

* If next DECODE completes in 1 I-STREAM fetch



ESA = SA for ARith FAULT
DSA = SA for ARith TRAP

FAULTS



E-Box INSTRUTION FLOW

I BOX

TOM KNIGHT

* = Items to be covered in Ibox presentation

FUNCTIONAL OVERVIEW

*1.1 FUNCTIONS PERFORMED

I-stream Decode
Ebox Microdispatch
Branch/Jump Instructions
String Fetching

*1.2 COMPONENTS OF THE IBOX

Micromachine
Instruction Buffer
Dispatch RAMS
Address Data Path

INSTRUCTION BUFFER AND ADDRESS DATA PATH

*2.1 INSTRUCTION BUFFER

Byte Usage
- Opcode
- Specifiers
- Branch Displacement
- Literal Data

2.1.1 Ibuffer Loading

Byte Valid Bits
Byte Rotation

2.1.2 Ibuffer Shifting

Shift Count - based on specifiers and context
Shifting of Bytes 1 & 0

2.2 IBOX DATA PATH EXAMPLES

2.2.1 Index Mode - Autoincrement Deferred (Longword Indexed)

*2.2.2 Literal Mode - Quadword Long Literal

*2.2.3 Branch Displacement - BEQL Instruction

2.2.4 H_floating Literal Asource

IBOX/EBOX INTERFACE

*3.1 OPERAND BUS

3.1.1 Operand Bus Data Unpacking

3.1.2 Operand Bus Operation

OP VALID interlock

3.2 IBOX TESTABILITY FROM THE EBOX

3.2.1 Reading Ibox Registers

3.2.2 Loading Microcode

3.2.3 Writing GPRs

*3.3 EBOX MICROCODE DISPATCH

Microdispatch Address Path
DRAM control & optimizations

3.4 EXTENDED OPCODES

ESCC opcodes have no operands and dispatch to a common ESCC microcode flow. The rest is up to the customer.

3.5 FIRST PART DONE

EP CTR <== 7

*3.6 BRANCHES

Simple Branches - Ibox tests Condition Codes
Computed Branches - Ebox restarts Ibox

3.7 JUMPS

3.8 STRING MODE OPERATION

IBOX/MBOX INTERFACE

*4.1 GENERAL DESCRIPTION

Mbox Response
Mbox Response Codes

*4.2 VA/VIBA ARBITRATION

Select VA reg only if doing Memory cycle

4.3 MBOX READ OPERATIONS FROM THE IBOX

4.3.1 Read Request Precedence

Ebox
Ibox D-stream
Ibox I-stream

4.3.2 Successful I-stream Read References

4.3.3 Unsuccessful I-stream References

Dispatched by Ebox Microdispatch Adr.

4.3.4 Successful D-stream Read References (from VA Register)

4.3.5 Unsuccessful D-stream Read References

Ebox microtrap direct from Mbox

4.4 MBOX WRITE OPERATIONS

4.4.1 Address In Ibox (VA Register)

4.4.2 Address In Ebox (VMQ Register)

4.4.3 Unaligned Writes

4.4.4 Page Boundary Crossing

IBOX CONTROL AND MICROMACHINE

*5.1 GENERAL DESCRIPTION

Ibox and Pipeline Control Point
Ibox Hardwr permitted to function by Micromachine

*5.2 IBOX MICROMACHINE

5.2.1 Micromachine Functions

Aligned Multiprecision read references
Unaligned read references
Indirect read references
Indirect Unaligned read references
Indexed Asource
Indexed operand (not Asource)
Register Mode Multiprecision reads
Zero fill for Short Literal Multiprecision
floating operands
Immediate Mode Asource
Immediate Mode operand (not Asource)
R-Log Unwind after Trap
R-Log Unwind after Fault
Adjustment of operand address after read
portion of a modify operand
Idle State (Ibox is hardware driven)
Branch Stall for Condition Code testing
Unaligned write references (address in VA register)

5.2.2 Micromachine Description

BEN MUX
ADR REG
RAM FILE
CS REG

*5.3 IBOX SCOREBOARD LOGIC

5.3.1 Byte, Word Writes To A Scoreboarded GPR

*5.4 IBOX STALLS

STALL ON	CLEARED BY
Waiting for cache	Mbox Response
Ibox Requesting W bus	Ibox granted W bus
OP VALID flag	E WAIT I from Ebox
IRD register full	CALL2 from Ebox microsequencer

IBOX DISPATCH/IR DECODE RAM

*6.1 GENERAL DESCRIPTION

Control of I-stream Processing
Supplies Ebox Microdispatch Addresses
DRAM configuration

6.2 DRAM CODE BIT FIELDS

Context	3 Bits
Type	3 Bits
Reference	2 Bits
Control (E/D)	2 Bits
Suspend	1 Bit
BDEST next	1 Bit
Last	1 Bit
Parity	1 Bit
Execute Address	6 Bits

MBOX/IBOX ERROR HANDLING

7.1 GENERAL DESCRIPTION

7.2 MBOX ERRORS - COVERED IN D-STREAM AND I-STREAM REFERENCES

7.3 IBOX ERRORS

7.3.1 Reserved Opcodes

DRAM has special disptach address

7.3.2 Not Enough Bytes In Ibuffer

Dispatch Address modified by Mbox IBUF Response Code

7.3.3 Reserved Addressing Modes

Hardware detected - special dispatch address

*7.3.4 Ibox Parity Errors

Detected by Ibox for all bytes used by the Ibox to create operand addresses. Opcodes and specifiers and GPR file are also checked

7.4 ERROR RECOVERY

7.4.1 Ibox Microinstruction Retry

ECC correction via Console Serial Channel

7.4.2 Macro Instruction Retry

R-LOG unwind after a Trap or Fault
In general, retry is predicated on ther being no modification of any GPR or memory data

COMPATIBILITY MODE - still being worked

E BOX

PAUL GUGLIELMI

EBOX GOALS

A. EXECUTE SIMPLE INSTRUCTIONS IN ONE MACHINE CYCLE.

B. BASE MACHINE:

1. MAKE INTEGER, LOGICAL, BCD, AND STRING INSTRUCTIONS FAST.

2. ADD THE MINIMUM AMOUNT OF HARDWARE NEEDED TO ACHIEVE REASONABLE "F" FORMAT FLOATING POINT PERFORMANCE.

3. ACCEPT WHATEVER PERFORMANCE THE DATA PATH ALLOWS ON "D", "G", AND "H" FORMAT FLOATING POINT.

C. RELY ON THE FBOX FOR HIGH PERFORMANCE FLOATING POINT.

D. MINIMIZE HARDWARE OPERATIONS THAT COULD LEAD TO BUG PRONE MICROCODE.

E. PARITY CHECK THE EBOX DATA PATH.

F. PARITY CHECK ALL EBOX RAMS AND WHERE POSSIBLE CORRECT THE ERROR AND CONTINUE.

G. SUPPORT INSTRUCTION RETRY.

LARGE SCRATCHPADS

A. HELP TO MAKE SIMPLE DATA PATH EFFICIENT

B. PROVIDE STORAGE FOR:

1. GENERAL PURPOSE REGISTERS
2. TEMPORARIES
3. CONSTANTS
4. ARCHITECTURALLY DEFINED REGISTERS

BINARY/BCD ALU

A. LOGICAL OPERATIONS

1. A, B, AND, AND.NOT, OR, XOR

B. BINARY OPERATIONS

1. A+B+[Ø, ALU<C>, PSL<C>]
2. A-B-[Ø, ALU<C>, NOT.ALU<C>, PSL<C>]
3. B-A-[Ø, ALU<C>, NOT.ALU<C>, PSL<C>]
4. A+4, A-4

C. BCD OPERATIONS

1. A+B+[Ø, ALU<C>, PSL<C>]
2. A-B-[Ø, NOT.ALU<C>]

D. DIVIDE FUNCTION

SHIFTER FUNCTIONS

A. SHIFT LEFT Ø TO 32 PLACES

B. SHIFT LEFT 1 FOR DIVIDE AND
SHIFT RIGHT 2 FOR MULTIPLY.

C. DATA CONVERSIONS:

1. BYTE SWAP

2. NIBBLE SWAP (NUMERIC TO PACKED)

3. PACKED TO NUMERIC

4. F FORMAT FLOATING POINT PACK

W REGISTER AND VMQ REGISTER

A. W REGISTER SCRATCHPAD READS AND WRITES

1. W REGISTER NOT ADDRESSABLE BY MICROCODE
2. WRITES ARE PIPELINED OPERATIONS

B. VMQ REGISTER

1. VIRTUAL MEMORY ADDRESS REGISTER
2. MULTIPLIER REGISTER
3. QUOTIENT REGISTER

SCRATCHPAD WRITES

	Tφ T1 T2 T3	Tφ T1 T2 T3	Tφ T1 T2 T3
MICRO WORD	ALU ← SP⟨ω⟩ SP⟨X⟩ ← ALU	SP⟨Y⟩ ← SP⟨Y⟩ + SP⟨X⟩	
	A LATCH ← SP⟨ω⟩ | HOLD A LATCH HOLD B LATCH	A LATCH ← SP⟨Y⟩ | HOLD A LATCH HOLD B LATCH SP⟨X⟩ ← WREG	SP⟨Y⟩ ← WREG
HARDWARE ACTIONS	ALU ← A LATCH ALU FCN: A WREG ← ALU REQ WBUS CYCLE FOR SP⟨ω⟩ WRITE	ALU A INPUT ← A LATCH ALU B INPUT ← ωU REG ALU FCN: ADD WREG ← ALU REQ WBUS CYCLE FOR SP⟨X⟩ WRITE	
		WBUS ← WREG[SP⟨X⟩]	WBUS ← WREG[SP⟨Y⟩]

VA BUS

A. OWNERSHIP CONTROLLED BY THE EBOX

B. VIRTUAL ADDRESSES SENT TO MBOX OVER THIS BUS

WBUS

A. MEMORY WRITES

B. BACK TO BACK WRITES

MEMORY WRITE

	TΦ | T1 | T2 | T3	TΦ | T1 | T2 | T3
MICRO CODE	SP<WR.SAV.LOC> ← ALU VMQ ← MEM ADR	WRITE MEMORY
		← WREG SP<WR.SAV.LOC> HOLD I WR LATCH
		LOAD I WR LATCH
HARDWARE ACTIONS	WREG ← ALU VMQ ← MEM ADR REQ WBUS CYCLE FOR SP<WR.SAV.LOC> WRITE	WBUS ← WREG IBOX WR LATCH ← WBUS VA BUS ← VMQ CACHE ← I WR LATCH CACHE ← MEM WR RQ CACHE ← VA BUS

CONDITION CODE SETTING

A. NATIVE MODE

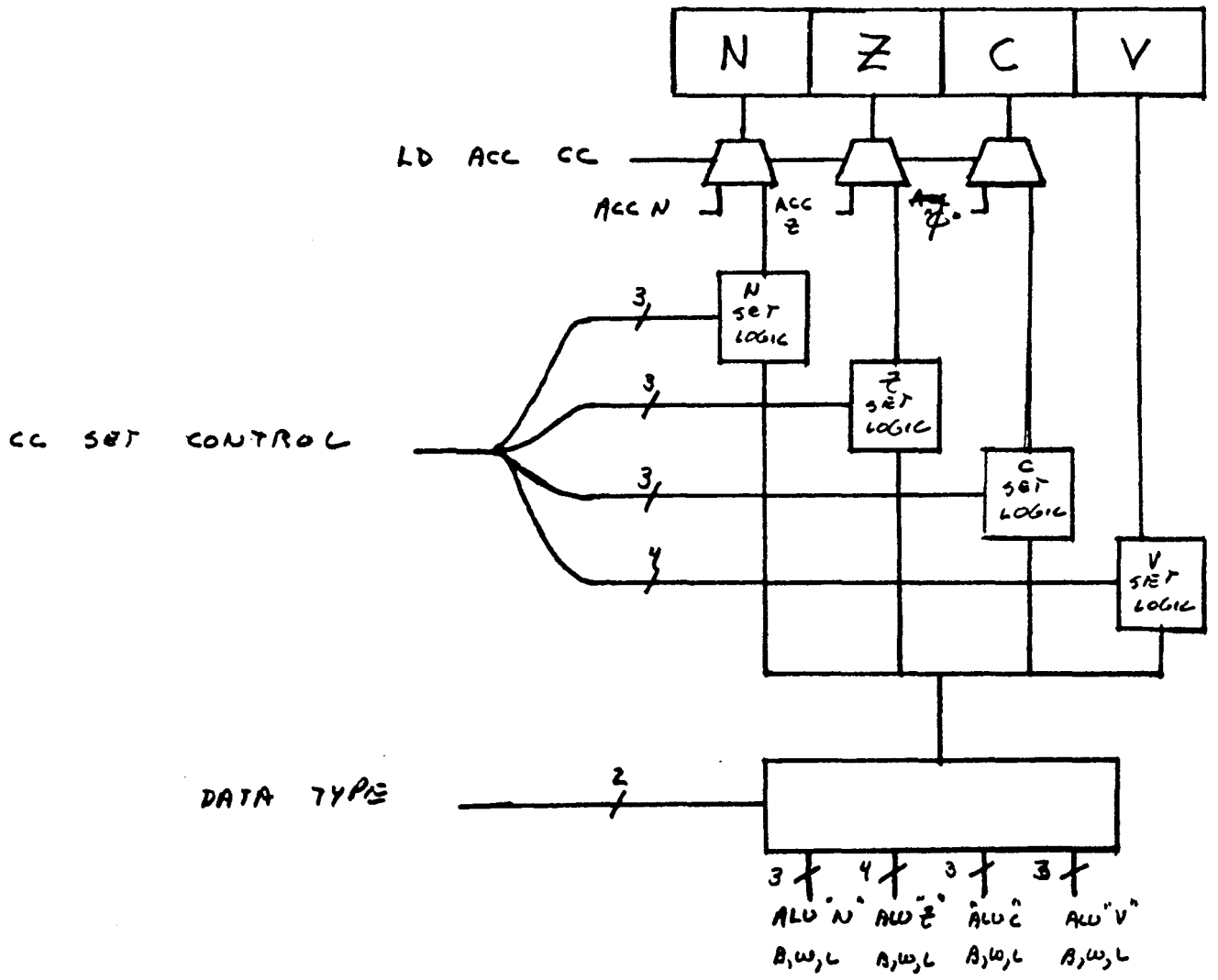
1. ONE INSTRUCTION DEPENDENT SETTING PER OPCODE FOR 1024 OPCODES

2. 64 ADDITIONAL WAYS TO SET THE CONDITION CODES ARE SPECIFYABLE BY THE MICROCODE.

B. COMPATIBILITY MODE

1. 64 COMBINATIONS OF SETTINGS ARE SPECIFYABLE BY THE MICROCODE.

CONDITION CODE SETTING



MICROTRAPS AND FAULT HANDLING

A. TYPES OF FAULTS

1. MEMORY MANAGEMENT

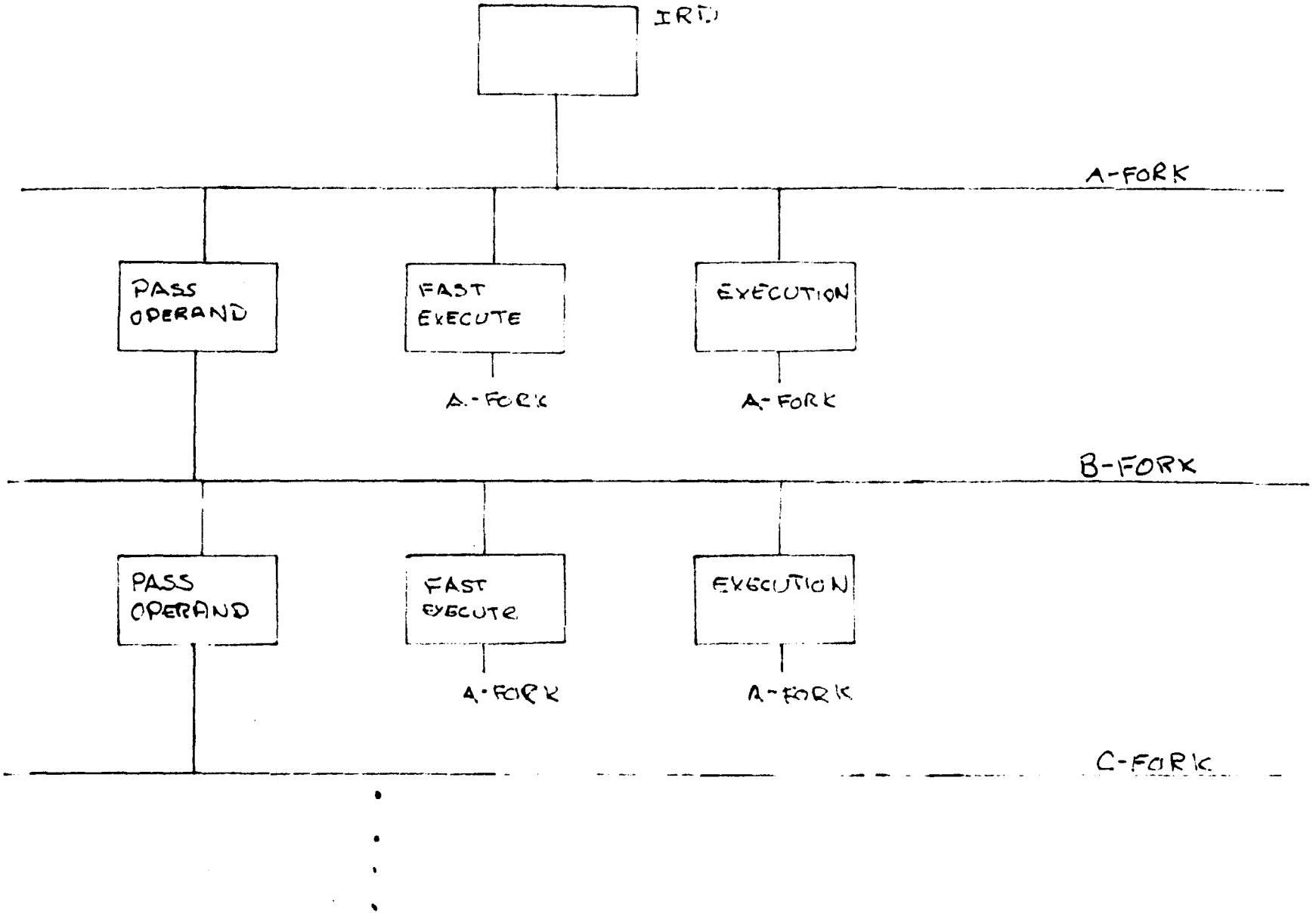
I. DATA TO EBOX

II. IBUF FETCHES

2. TRAPS AND INTERRUPTS

3. HARDWARE ERRORS

B. MICROTRAP OUT OF CURRENT MICROINSTRUCTION



E-box INSTRUTION FLOW

F BOX

MIKE BROWN

CONSOLE

ED ANTON

I-BOX	DECODE SPECIFIERS ACCESS OP 1 GPR CALC OP 1 VA	INSTR OP BFR PREFETCH		DECODE SPECIFIERS NEXT INSTR		
M-BOX		FETCH OPERAND 1	I PREFETCH			
E-BOX		(LAST CYCLE PREVIOUS INSTR)	(A-FORK) ACCEPT OP. 1 ACCESS OP. 2	(B-FORK)	CPSYNC	(AFORK
F-BOX		(LAST CYCLE PREVIOUS INSTR .OR. WAIT)	ACCEPT OP. 1 ACCESS OP. 2 EXT. DIF	ALIGN, ADD., CALC NORM CT	NORM ROUND EXP. ADJ FPSYNC	WAIT- .OR. NEXT I

Venus F-Box Summary

1.0 GOALS

OPTIONAL Modules removable for low-cost entry level system

HIGH-PERFORMANCE (3-4 x FP780)

FCS within 3 mos. of base machine

RAMP- lower on list due to warm backup in base machine.

MODULE COUNT- 2 MCA-intensive modules (now at 3 Modules due to increase in 10K : MCA ratio)

COST-Perceived to be relatively cost-insensitive

2.0 FUNCTIONALITY

64-bit data path in Align, Add, and Normalize logic, 32 x 8 multiply path in 1/2 cycle to provide:

1. Full support of:

   ADD, SUB, MUL, and DIV in F, D, and G_Floating formats.

2. Subroutines invokable by EBOX to support:

   EMOD and POLY in F, D, and G_Floating formats.

   ADD, SUB, MUL, DIV, EMOD, and POLY in H_Floating format.

3.0 INTERFACES TO OTHER SYSTEM COMPONENTS

IBOX
    Operand bus-operand delivery

EBOX
    W bus-operand/result delivery

Control and synchronization

MBOX-no direct interface

CONSOLE
Diagnostics bus

Control store load

4.0 TRADEOFFS

1. Pipelined vs. Iterative (Speed vs. Cost)

   Improvement in speed not proportional to cost due to cache bandwidth limitations, therefore the iterative approach chosen.

2. Support of all instructions/data types vs. limited support

   Chose to spend to optimize performance of high-frequency instructions and data types. Provide microcode hooks to aid base machine for the less-frequent operations/data types.

3. Microcode for flexibility vs. Hard control for speed.

   Microcode is fast enough. Cache bandwidth is well-matched by microcode implementation. Much lower ECO risk in microcode.

4. Divide algorithm- Complex, multiple bit per iteration vs. simple non- restoring. Resolved in favor of simplicity Performance gain in the fancy algorithm was not substantial enough to risk slips in FCS if it fails. Divide is a low- frequency operation so lower performance less painful.

5.0 ALGORITHMS

ADD/SUBTRACT

Alignment is accomplished by a 0-63 place shifter. Parity is computed for the bits lost in the alignment and is used to predict the parity of the sum.

The sum is applied to a priority-encode network to determine the normalization count. It is also applied to a network which detects all cases of a "1" followed by a "0". The output of this network is applied to another priority-encode network. The output of this network defines the location of the leftmost 1-0 transition. If this transition is at or to the right of the rounding bit, a roundoff carry will occur. This is detected and presented to the exponent ALU and formatting logic to permit one-cycle normalize and round operations.

MULTIPLY

The multiplier is treated as a set of 8-bit quantities. The 8-bit elements are recoded using Booth's method and 4 32-bit partial products are produced. These partial products are reduced by a column-reduction technique to two 40-bit operands. These are summed by a 3-input adder with the accumulated product to create a new accumulated product. The accumulated product is delivered to the adder for normalization and delivery to the W-bus.

DIVIDE

Divide is implemented by a non-restoring, 1-bit algorithm in the adder. The divide iteration is performed twice per microcycle, or 33 ns per quotient bit. The quotient bits are accumulated by the multiplier product register. At the completion of the quotient iterations, the quotient is delivered to the adder for normalization and delivery to the W-bus.

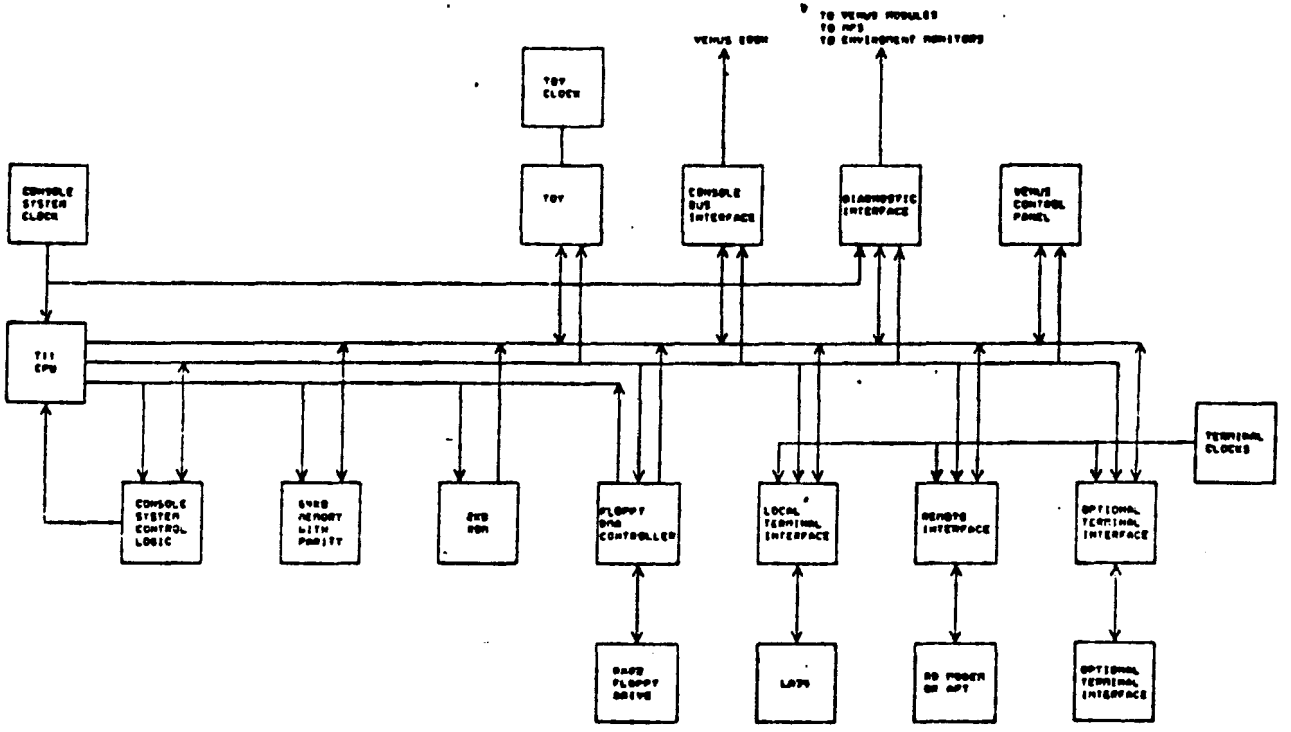
6.0 RAMP

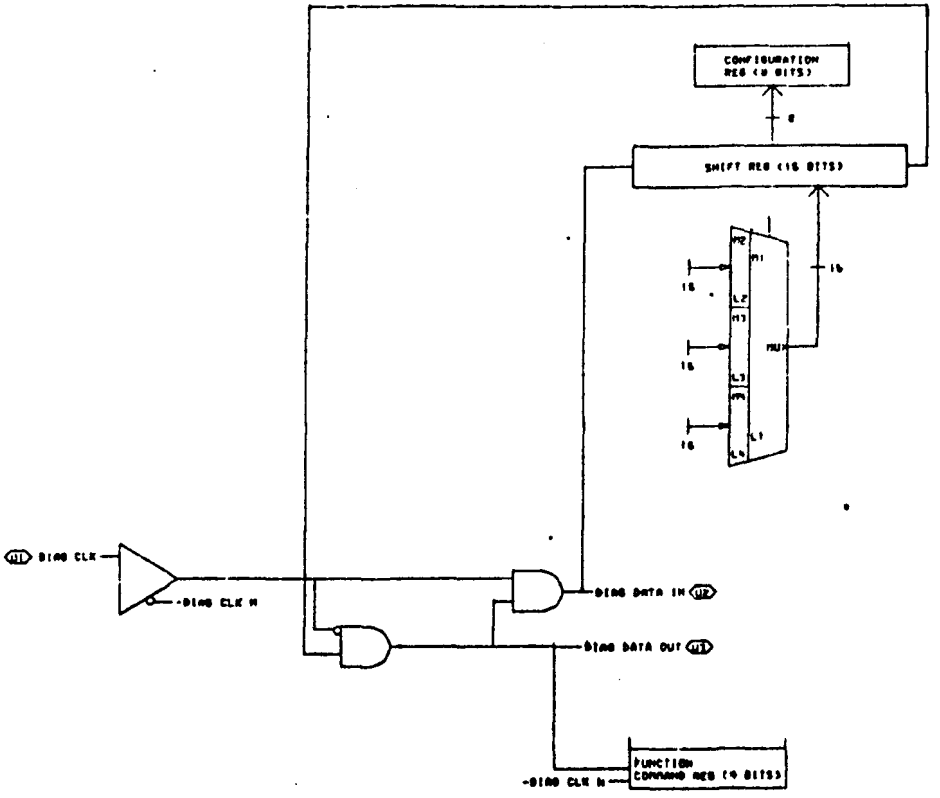
Parity-checked control store, with alternate copy for retry. Alternative is console-driven ECC a la Ebox.

Parity-checked adder.

Warm FP backup under software control via enable bit in mode register.

Multiplier- Checking strategy still not firm. Studying the propagation of errors and the sensitivity to error bursts. Parity prediction and modulo N checking under consideration.





REVISIONS

SYM	CHANGE NO.	REV

digital

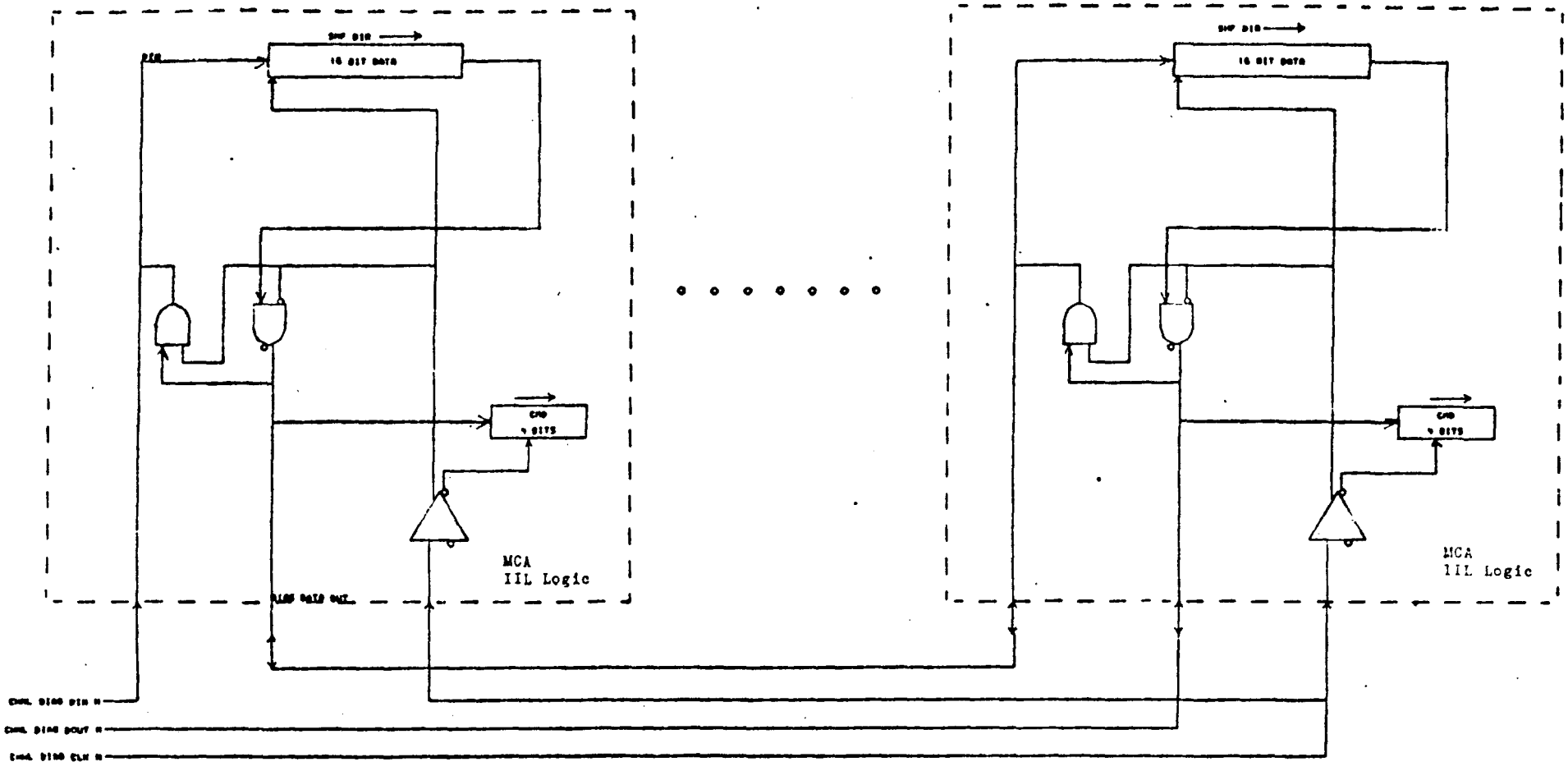
DES. K.V. Hastings DATE 27-FEB-80
CHK'D. DATE
VENUS\CUIS\SLX211L.DRW 27-FEB-80 16:35
FIRST USED ON (OPTION/MODEL)

ENG. DATE
BOARD LOCATION:
SHEET OF
NEXT HIGHER ASSEMBLY:

TITLE:

SIZE D | CODE CS | NUMBER | REV.

0 7 6 5 4 3 2 1



MCA Diagnostics
Channel

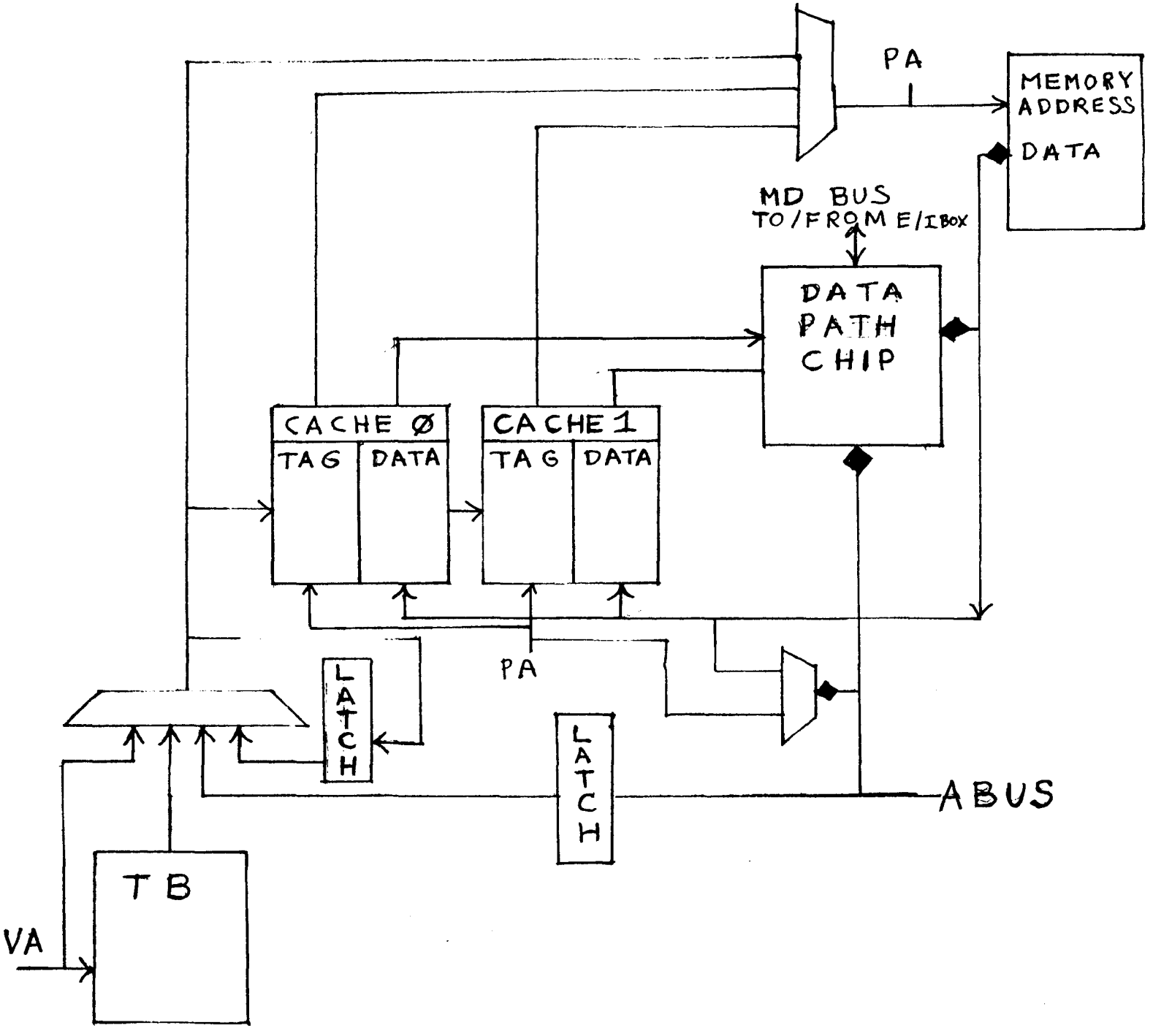
M BOX & MEMORY ARRAY

BILL BRUCKERT

GOALS

1. READ IN ONE CYCLE
2. WRITES IN ONE CYCLE
3. MASKED WRITES IN TWO CYCLES
4. SUPPORT SBI, BI
5. TIME TO MARKET = BB FEB 81 POWER ON
6. RAMP
    1. ISOLATION OF FAULTS TO MBOX
    2. DETECTION OF A SINGLE BIT ERRORS IN DATA PATH
7. COST - 2 MODULES

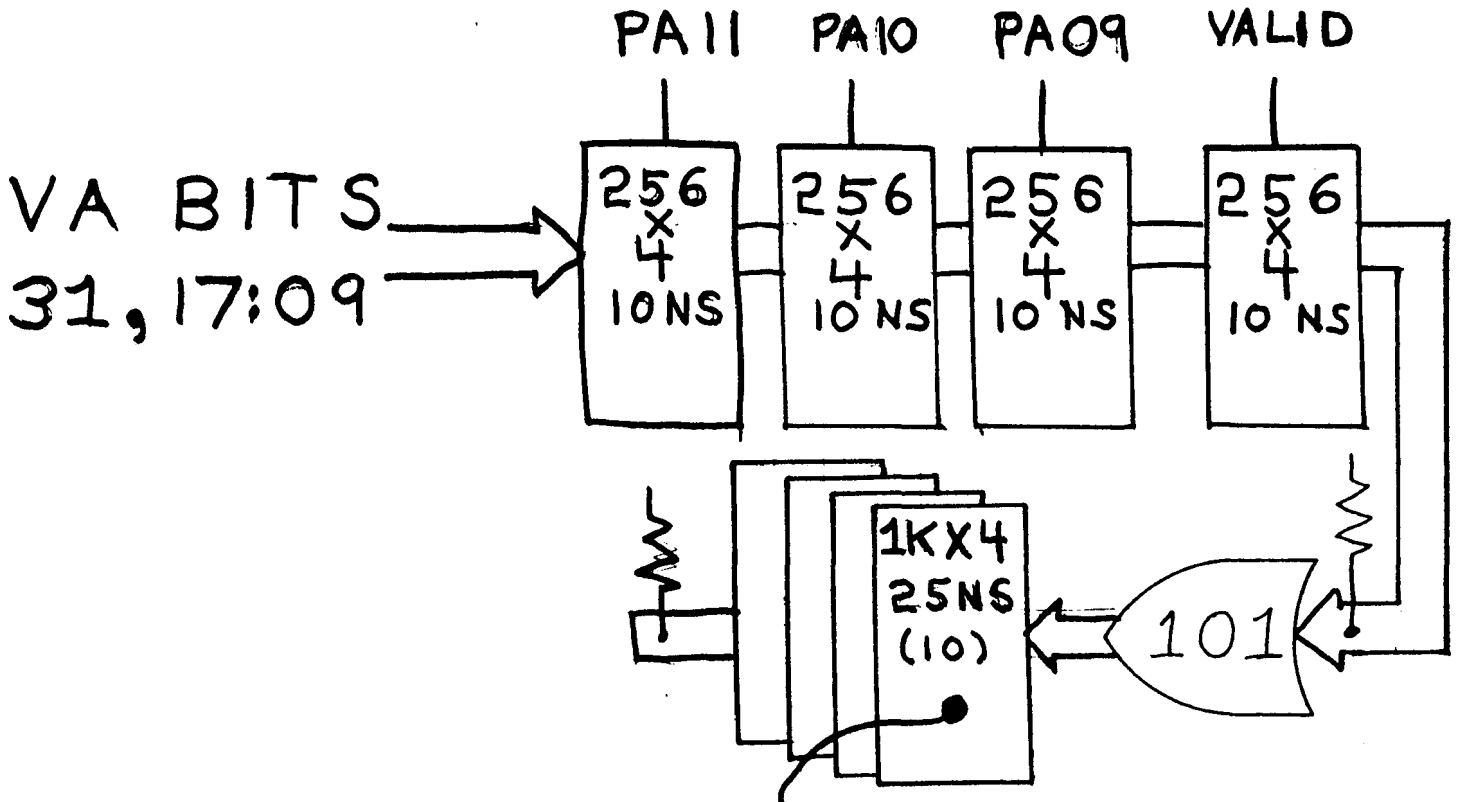
VENUS MBOX



TRANSLATION BUFFER

1. 1-WAY ASSOCIATIVE - CHOSEN TO SUPPORT PHYSICALLY ADDRESSED CACHE
2. 1K ENTRIES TOTAL
    1. 512 SYSTEM SPACE
    2. 512 PROCESS SPACE
3. PARITY CHECKED
4. FAST TB CLEAR - 128 MICRO CYCLES TO CLEAR EITHER SYSTEM OR PROCESS SPACE
5. ERROR RECOVERY - INVALIDATE AND REFILL

TRANSLATION BUFFER



4 PROTECTION
1 MODIFY
1 PFN VALID
18 ADDRESS BITS 29:12
1 PARITY BIT
13 TAG BITS (30...18)
1 PARITY BIT FOR TAG

TOTAL 39

WHY ONE WAY ASSOCIATIVE?

TB 8 TIMES LARGER THAN VAX 11/780

PDP-10 TRACES SHOWED THAT GOING FROM A 2-WAY TO 1-WAY ASSOCIATIVE CAHE EFFECTIVELY HALVED THE CACHE SIZE

LESS ASSOCIATIVE (VAX 11/780 - 2-WAY)

MISS RATE SHOULD BE BETTER THAN MISS RATE ON VAX 11/780

LESS PINS REQUIRED TO IMPLEMENT MCA

NO MATCHES REQUIRED BEFORE PA BITS TO CACHE CAN BE SELECTED

DATA CACHE

1. SAME SIZE AND ASSOCIATIVITY AS VAX 11/780
2. 2-WAY ASSOCIATIVE
3. BLOCK SIZE OF 16 BYTES
4. TOTAL SIZE 8 KBYTES
5. PHYSICALLY ADDRESSED
6. WRITEBACK
7. LRU REPLACEMENT ALGORITHM
8. PARITY CHECKING ON A PER BYTE BASIS
9. RAMP FEATURES
    1. ALL DATA BUSES PARITY CHECKED
    2. ERROR CORRECTION PROVIDED ON WRITTEN BIT
    3. EACH CACHE CAN BE TURNED OFF SEPARATELY
    4. REDUNDANT WRITE MODE

WHY WRITEBACK?

ADVANTAGES:

1. BETTER BYTE WRITE PERFORMANCE
2. REDUCES MEMORY COMPLEXITY
3. LESS MEMORY CONTENTION
4. MORE MEMORY BANDWIDTH FOR I/O BANDWIDTH TO I/O
5. IMPROVES WORSE CASE ACCESS TIME TO MEMORY (IMPORTANT FOR SBI)
6. SINCE WRITE ALLOCATE BY BLOCK COMPLEXITY REDUCED
7. TYPICAL AND WORSE CASE ACCESS TIMES BETTER THAN WRITETHRU

DISADVANTAGES:

1. RECOVERY FROM ERRORS
2. MULTIPROCESSING

WHY DOESN'T THE DATA CACHE HAVE ECC?

4. THE PARITY BITS MUST BE STORED ANYWAY IN ORDER TO RETURN DATA TO THE PROCESSOR SOON ENOUGH.

5. ADDS

    1. 2 RAMS TO DATA STORE AND 1 TO TAG STORE PER CACHE (20%) INCREASE

    2. 10K PARTS TO PIPELINE ECC BITS

    3. CONTROL LOGIC

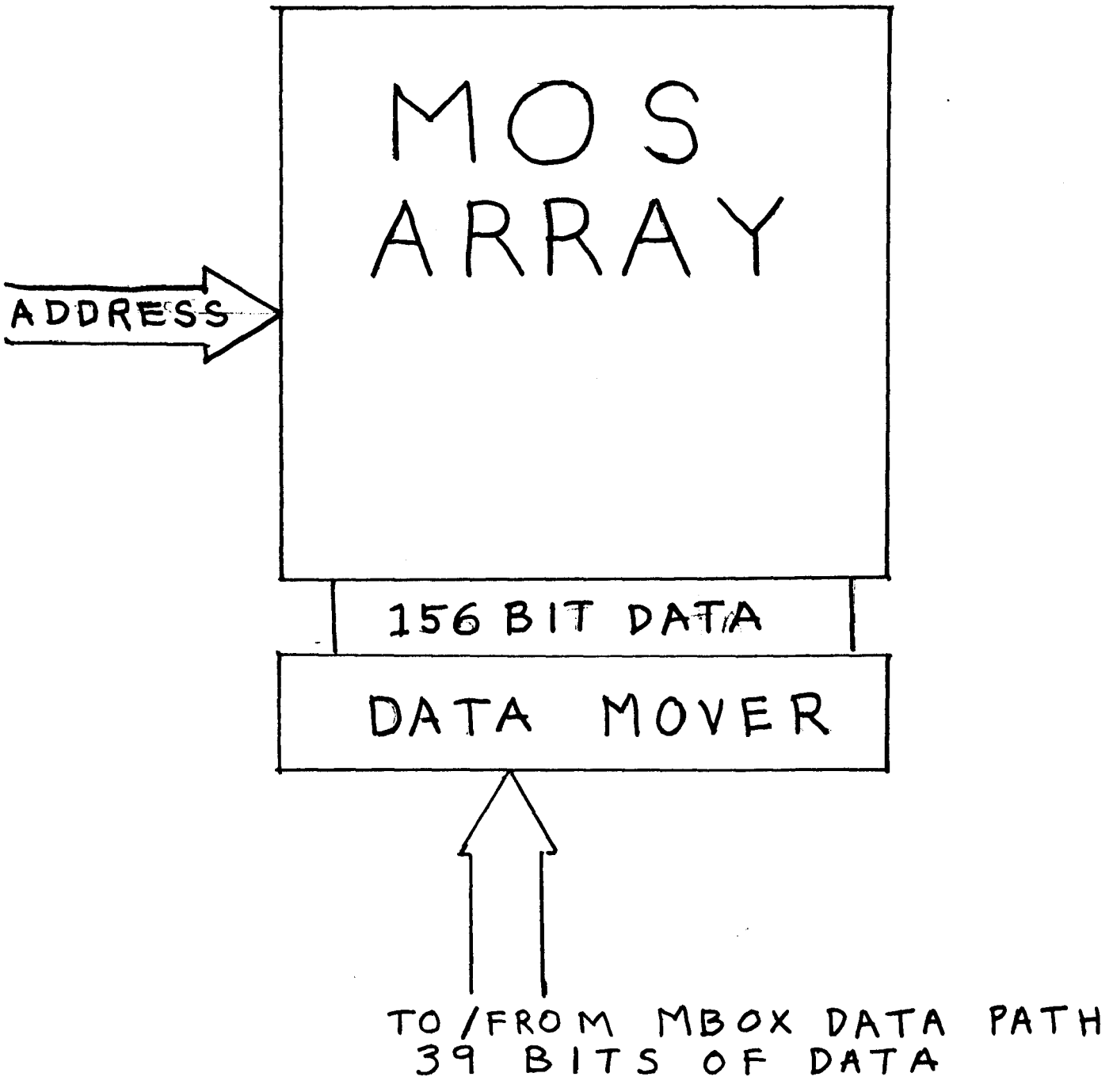
    4. LOGIC TO GENERATE TAG SYNDROME AND CORRECT TAG BITS

6. SPACE OF ADDED LOGIC WOULD OVERFLOW MODULE

7. REDUNDANT WRITE MODE AND CACHE DISABLE PROVIDE CAPABILITY TO HAVE EITHER HIGH PERFORMANCE OR HIGH AVAILABILITY CHOICES UPON DETECTION OF ERROR

MEMORY

1. SUPPORTS UP TO 8 ARRAY MODULES
2. UP TO 2 ARRAY MODULES CAN BE REPLACED WITH MEMORY REPEATERS
3. 8 MBYTE WITHOUT REPEATERS
4. ACCESS TIME FROM VA REGISTER BEING CLOCKED TO DATA CLOCKED INTO E/IBOX 9 TICS FOR FIRST WORD.
5. OPERATIONS PERFORMED
    1. 4-WORD READS
    2. 4-WORD WRITES
    3. 2-WORD WRITES
6. OVERLAPED OPERATIONS ALLOWED.
7. MAXIMUM MEMORY READ BANDWIDTH TO SAME ARRAY CARD 34 MBYTES
8. REFRESH ON ARRAY CARD
9. RAMP FEATURES
    1. ADDRESS PARITY APPENDED INTO ECC
    2. WRITTEN BIT SET IN CACHE SET WHEN CORRECTION PERFORMED
    3. FIXED ACCESS TIME FOR DIAGNOSTICS



ABUS INTERFACE

1. COMMON ADDRESS/DATA LINES - 25 OHM ECL BUS
2. CACHING ALGORITHM

I/O OPERATION	CACHE HIT	CACHE MISS
LONGWORD READ	DATA FROM CACHE	DATA FROM MEMORY
QUADWORD READ	DATA FROM CACHE	DATA FROM MEMORY CACHE DATA
OCTAWORD READ	DATA FROM CACHE	DATA FROM MEMORY
LONGWORD WRITE	WRITE IN CACHE	WRITE MEMORY
OCTAWORD WRITE	INVALIDATE CACHE, WRITE MEMORY	WRITE MEMORY
MASKED (BYTE) WRITE	WRITE CACHE	REFILL, THEN WRITE CACHE

3. BUS PARITY CHECKED

ADAPTER BUS

JIM LACEY

1

A BUS PRESENTATION OUTLINE

JIM LACY

23-FEB-80

1. GOALS
2. NON-GOALS
3. GENERAL A BUS CHARACTERISTICS
4. ADDRESS SPACE PARTITIONING
5. READ AND WRITE TRANSACTIONS
6. A BUS THROUGHPUT

2

GOALS

1. SUPPORT 4 DEVICES

   - FIRST TWO CAN BE BI/BI, SBI/SBI, BI/SBI
   - SECOND TWO MAY BE ADAPTERS TO CI, DR-32 INTERFACE, OR OTHER INTERCONNECT OR DEVICE

2. SUPPORT MEMORY ON BI AND SBI

3. SUPPORT OCTAWORD, QUADWORD, AND LONGWORD DMA OPERATIONS AS SINGLE A BUS TRANSACTIONS.

4. SUPPORT BYTE MASKING FOR CPU AND I/O ADAPTER INITIATED WRITE OPERATIONS.

5. SUPPORT ADAPTERS WHICH INTERFACE TO INTERCONNECTS WHICH OPERATE ASYNCHRONOUSLY WITH RESPECT TO CPU.

6. PARITY CHECKING ON ALL ADDRESS/DATA LINES AND AS MANY CONTROL LINES AS PRACTICAL AND CHECKING OF CONTROL FUNCTIONS WHERE FEASIBLE.

NON-GOALS

1. DIRECT COMMUNICATION BETWEEN I/O ADAPTERS.
2. SUPPORT OF CPU REQUESTS LONGER THAN A LONGWORD.
3. BYTE MASKING CAPABILITY FOR I/O ADAPTER INITIATED READS TO VENUS MEMORY.

GENERAL A BUS CHARACTERISTCS

1. A BUS INTERCONNECTS I/O ADAPTERS, MBOX, AND E/IBOX

   - CPU INITIATED REQUESTS TO I/O ADAPTERS VIA MBOX.
   - I/O ADAPTER INITIATED DMA REQUESTS TO VENUS MEMORY VIA MBOX.
   - INTERRUPT REQUESTS FROM I/O ADAPTERS TO EBOX.
   - A BUS CLOCKS FROM MODULE IN E/IBOX.

2. A BUS SIGNALS

   - 70 (-0,+10) ECL SIGNALS.
   - SIGNALS BETWEEN MBOX AND I/O ADAPTERS TRAVERSE MOST OF CPU AND I/O ADAPTER BACKPLANES (43" TOTAL LENGTH).
   - BI-DIRECTIONAL LINES TERMINATED AT BOTH ENDS.

3. CUSTOM A BUS INTERFACE CHIP

   - 4-BIT SLICE IN 28-PIN DIP.
   - PERFORMS ECL/TTL SIGNAL CONVERSION.
   - HIGH SPEED
   - CONTAINS DUAL PORT 16X4 REGISTER FILE.
   - ALLOWS INDEPENDENT, ASYNCHRONOUS OPERATIONS BETWEEN PORTS.

4. A BUS IS SYNCHRONOUS WITH VENUS CPU

   - I/O ADAPTERS PERFORM ALL SYNCHRONIZATION BETWEEN A BUS AND ADAPTED BUSES OR INTERFACES.
   - WELL DEFINED POINTS OF SYNCHRONIZATION IN ADAPTERS.
   - CONTROL OF BUFFERS IN REGISTER FILES PASSED BACK AND FORTH BETWEEN I/O ADAPTERS AND MBOX.
   - INTERRUPT REQUESTS ARE SYNCHRONIZED BEFORE BEING PASSED TO EBOX.

5. A BUS PROTOCOL

   - BI-DIRECTIONAL DATA LINES SHARED FOR COMMAND/ADDRESS INFORMATION AND READ/WRITE DATA.
   - 67 NANOSECOND BUS CYCLE.
   - INFORMATION TRANSFER RATE CONTROLLED BY MBOX.
   - CONTROL SIGNALS ARE PIPELINED.
   - ARBITRATION OVERLAPPED WITH DATA TRANSFERS.
   - INTERLOCKED PROTOCOL WITH RESPECT TO EACH I/O ADAPTER, BUT MBOX MAY OVERLAP TRANSACTIONS BETWEEN I/O ADAPTERS TO INCREASE THROUGHPUT.

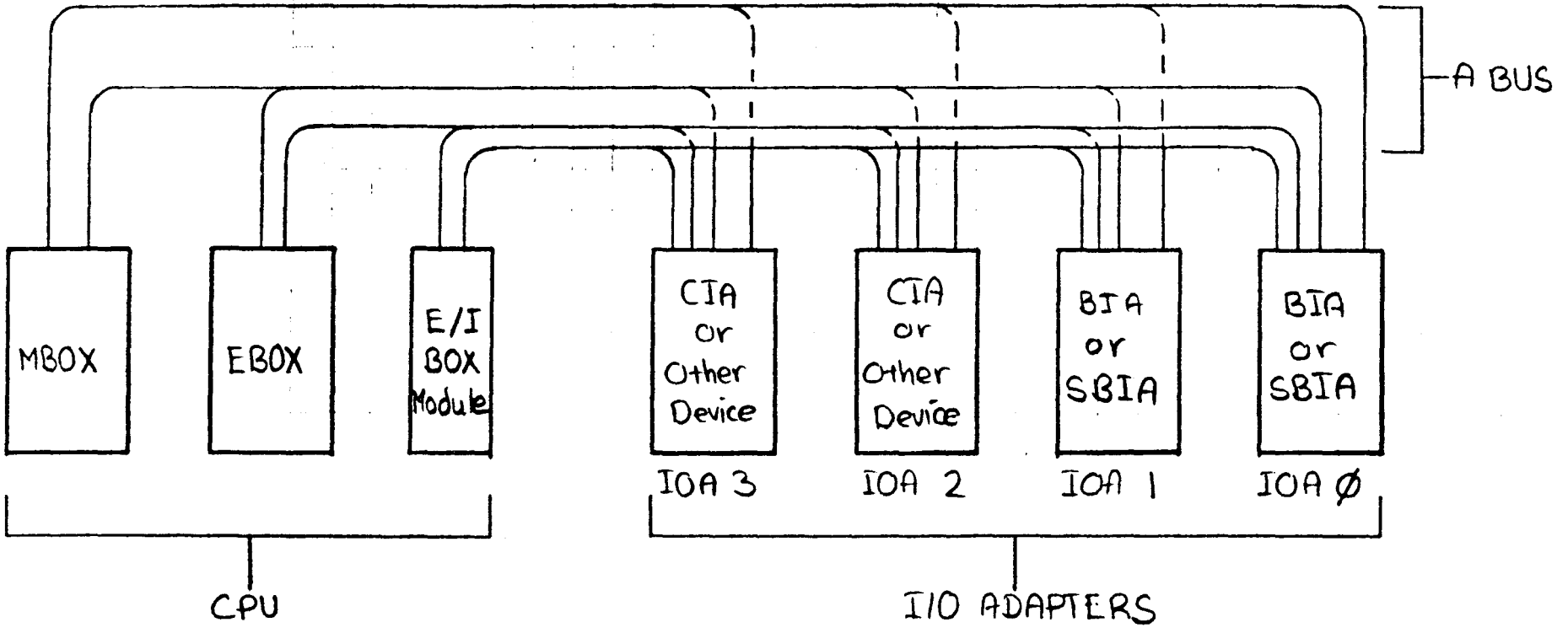


FIGURE 1 : SIMPLIFIED A BUS BLOCK DIAGRAM

J. LACY
25-FEB-80

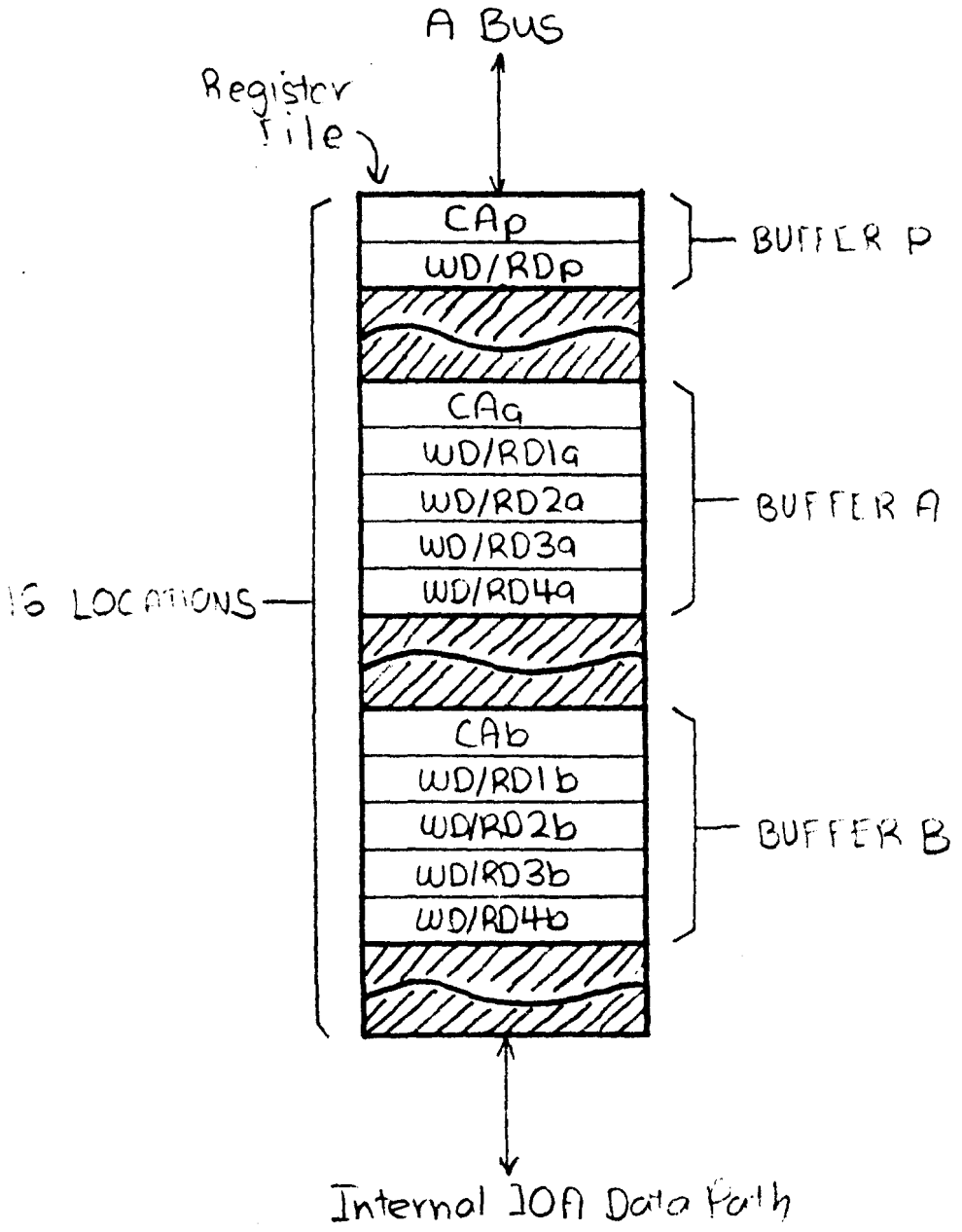


FIGURE 6:

PHYSICAL IMPLEMENTATION OF TRANSACTION BUFFERS

COM

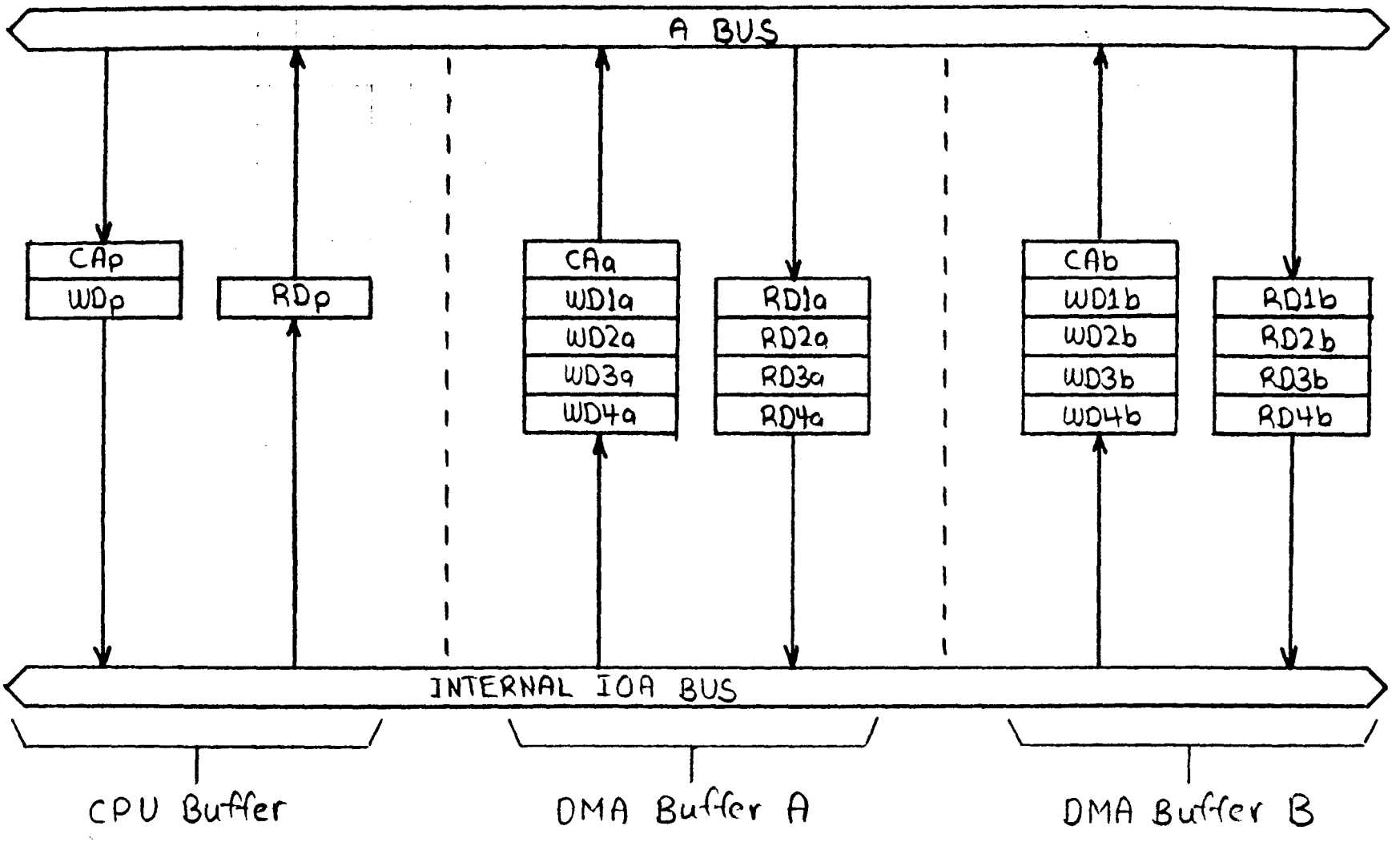


FIGURE 2: A BUS TRANSACTION BUFFERING

COMPANY CONFIDENTIAL

J. LACY
25-FEB-80

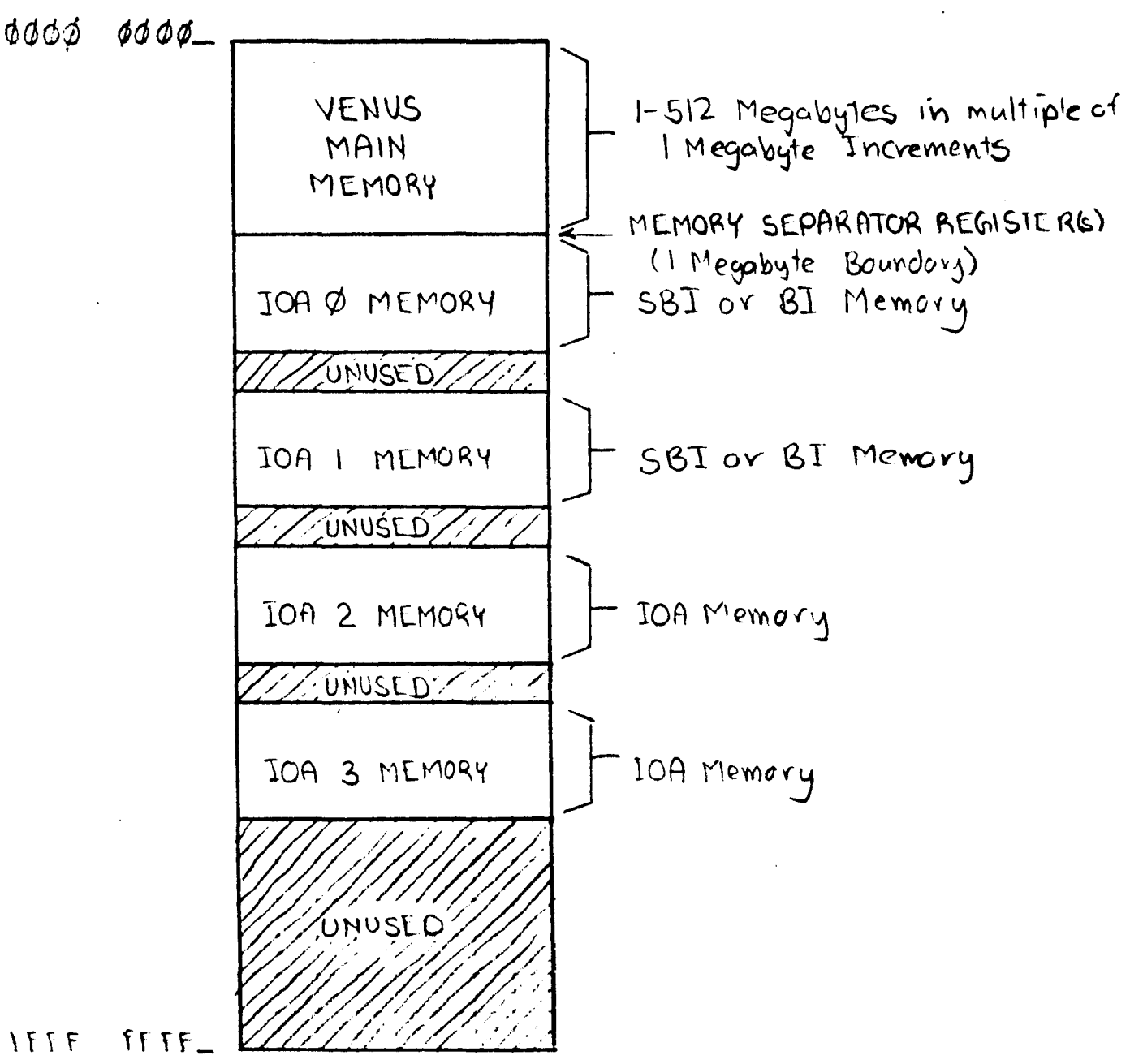


FIGURE 3:

PHYSICAL MEMORY ADDRESS SPACE ALLOCATION

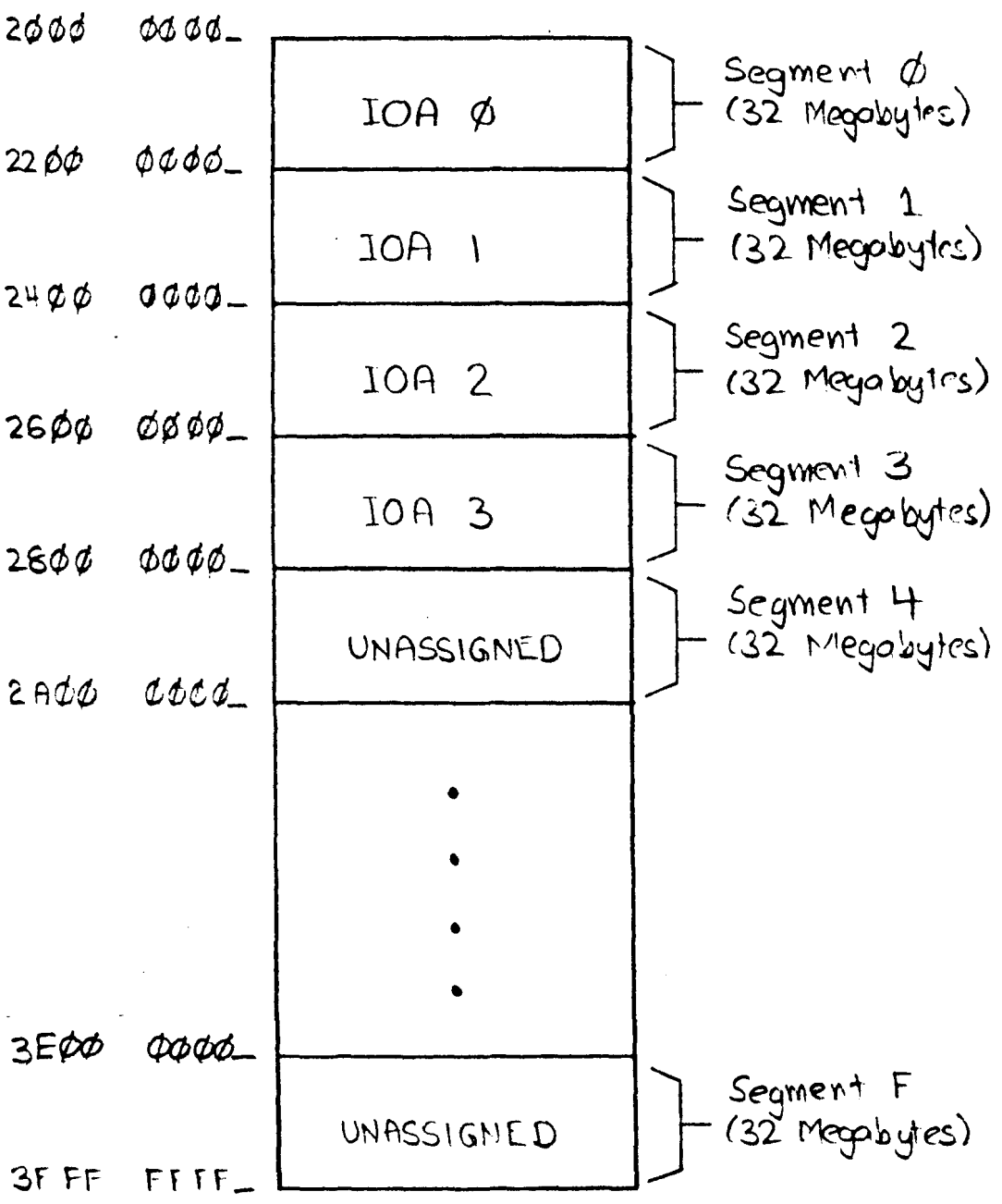


FIGURE 4:

I/O ADDRESS SPACE ALLOCATION

COMPANY CONFIDENTIAL

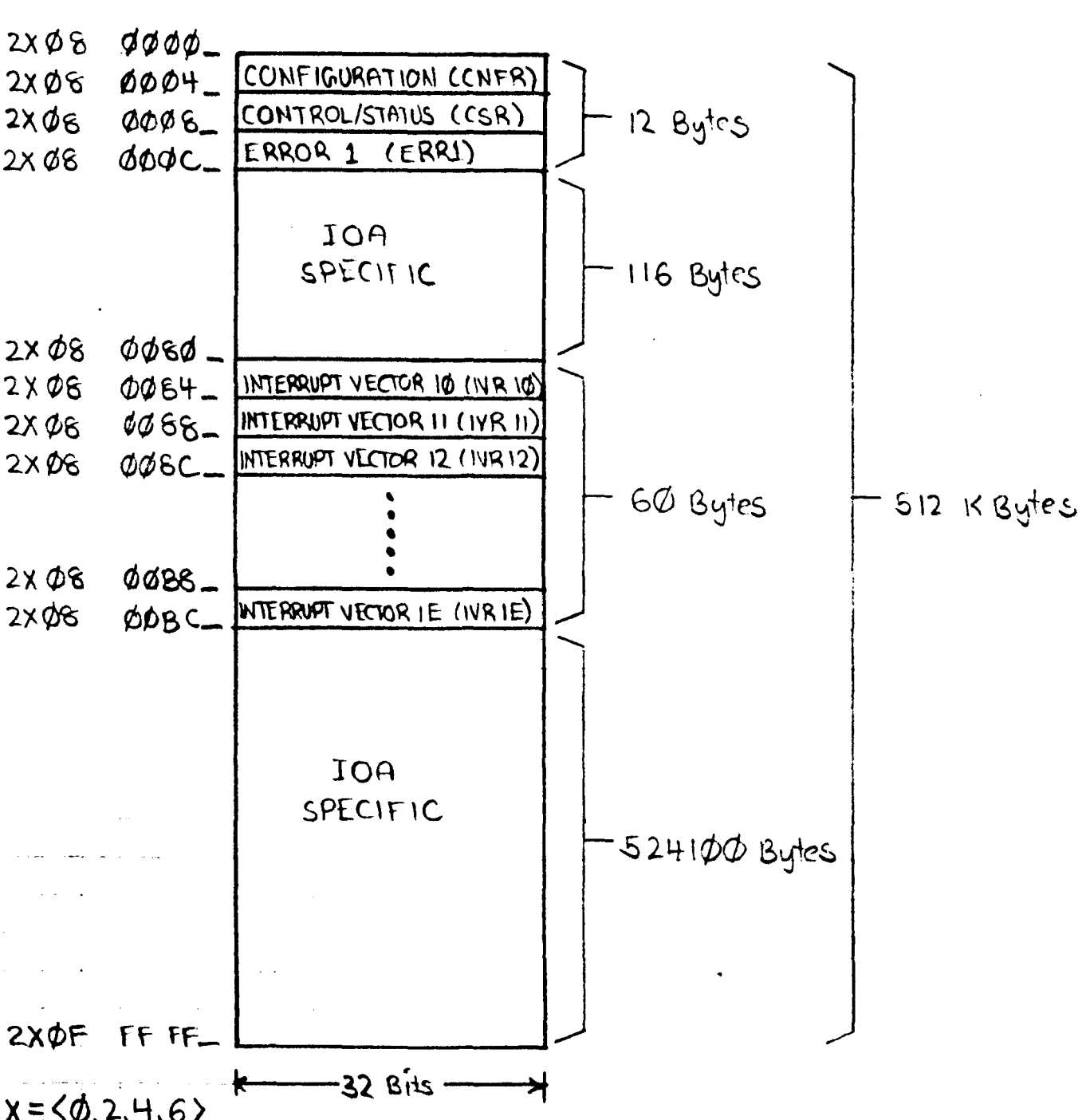


FIGURE 5:

INTERNAL I/O ADAPTER REGISTER ASSIGNMENTS

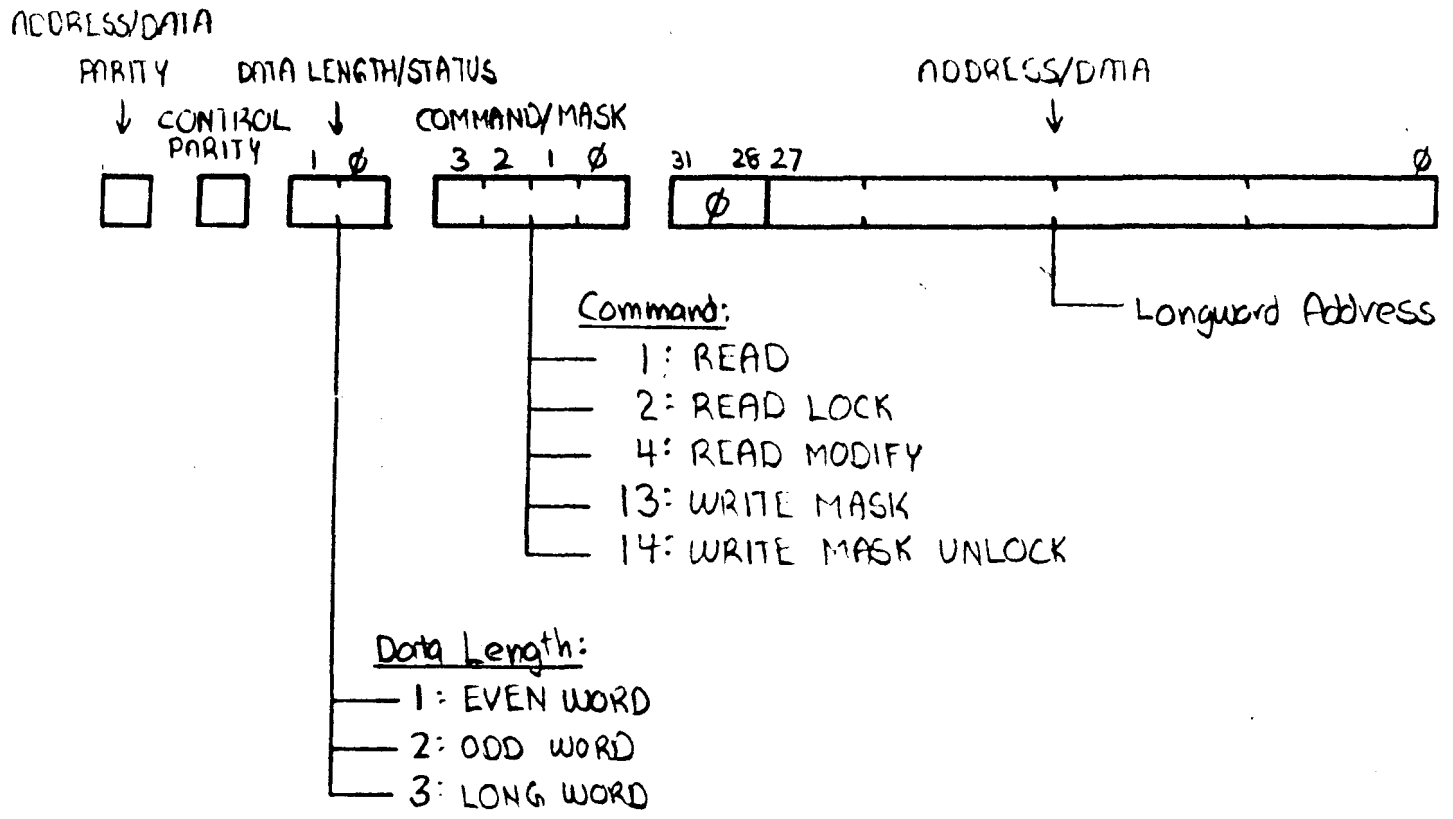


FIGURE 7 : CPU COMMAND/ADDRESS CYCLE FORMAT

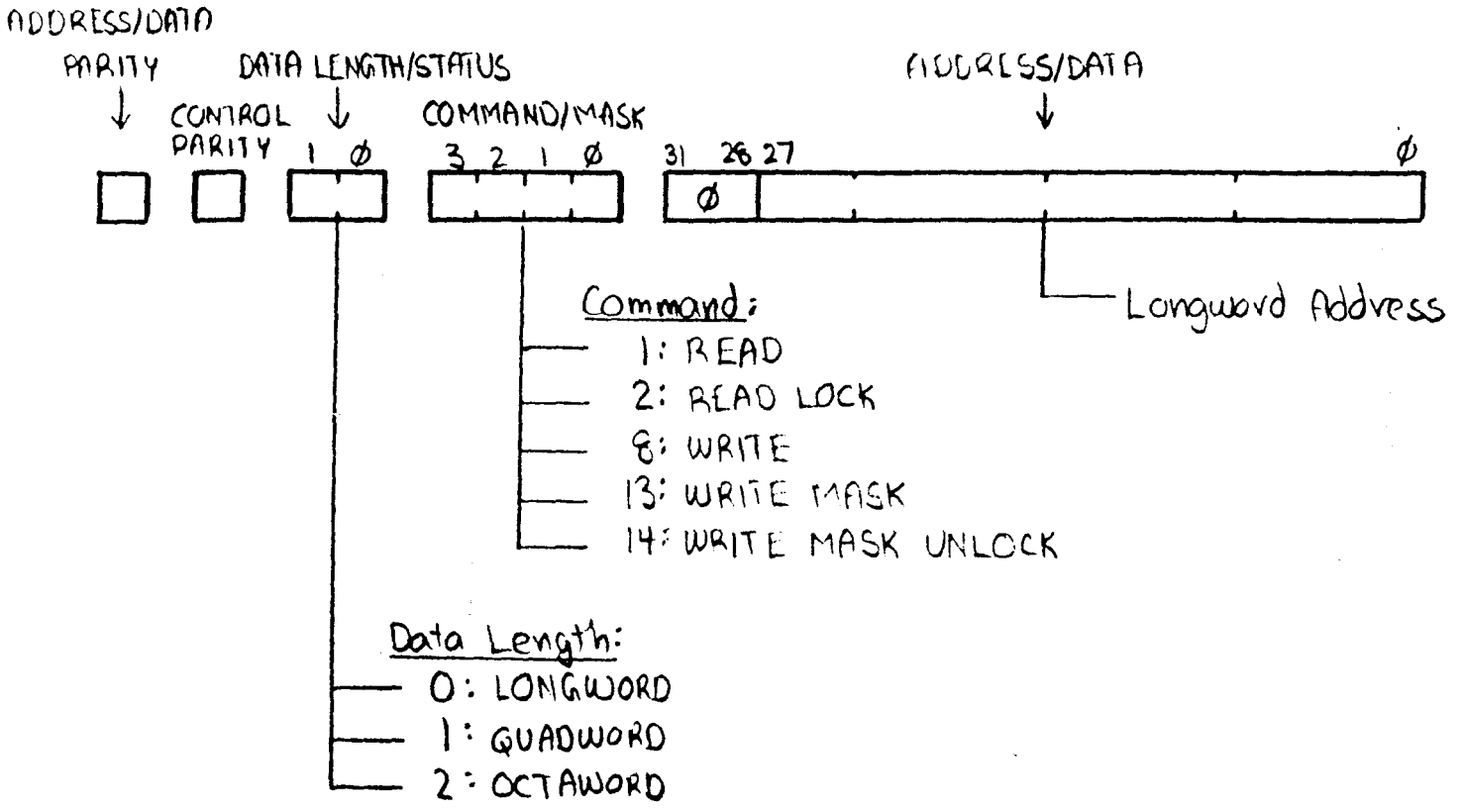


FIGURE 8: DMA COMMAND/ADDRESS CYCLE FORMAT

COMPANY CONFIDENTIAL

J. LACY
25-1LB-80

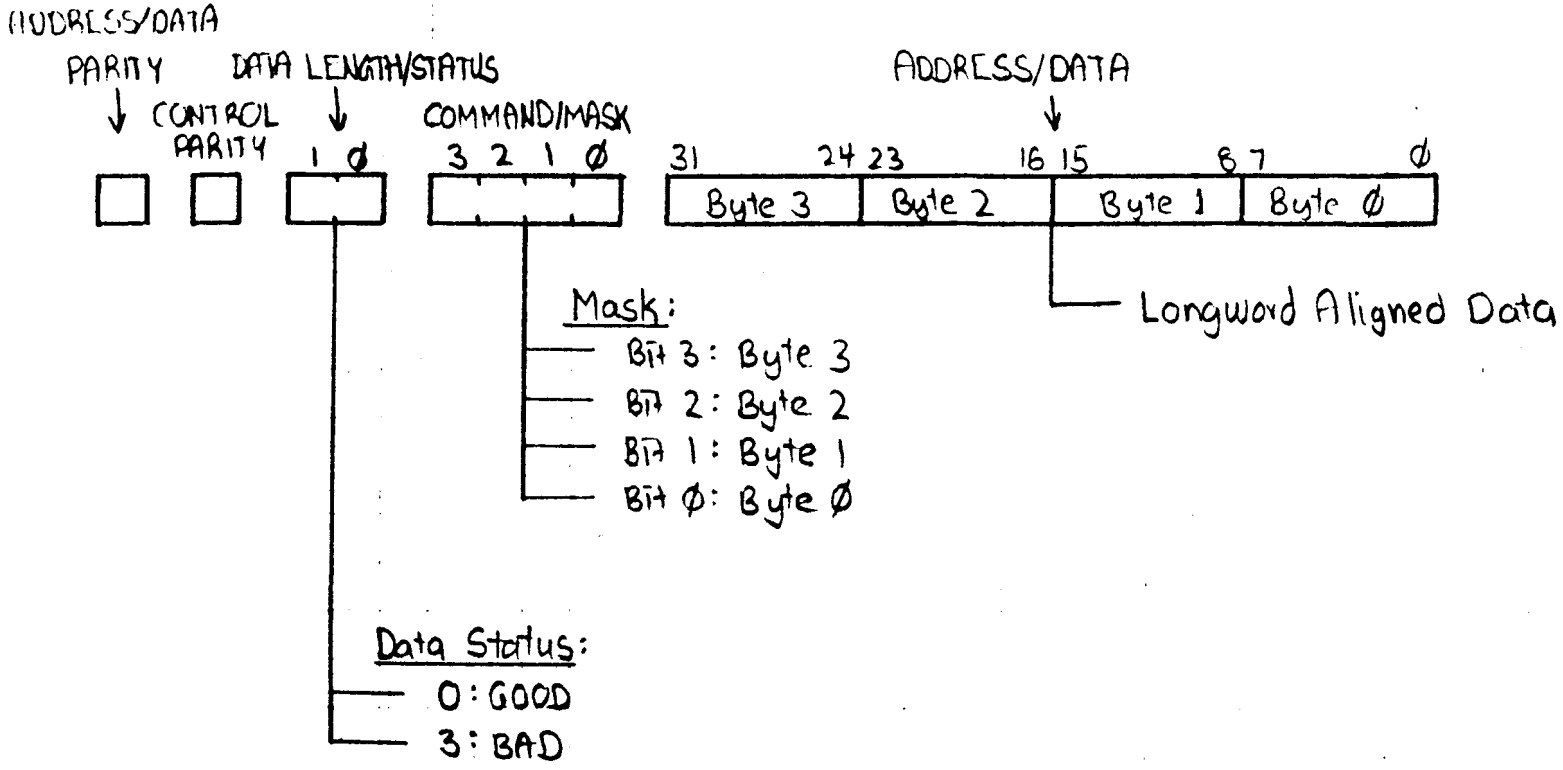


FIGURE 9: WRITE DATA CYCLE FORMAT

J. LACY
25-FEB-80

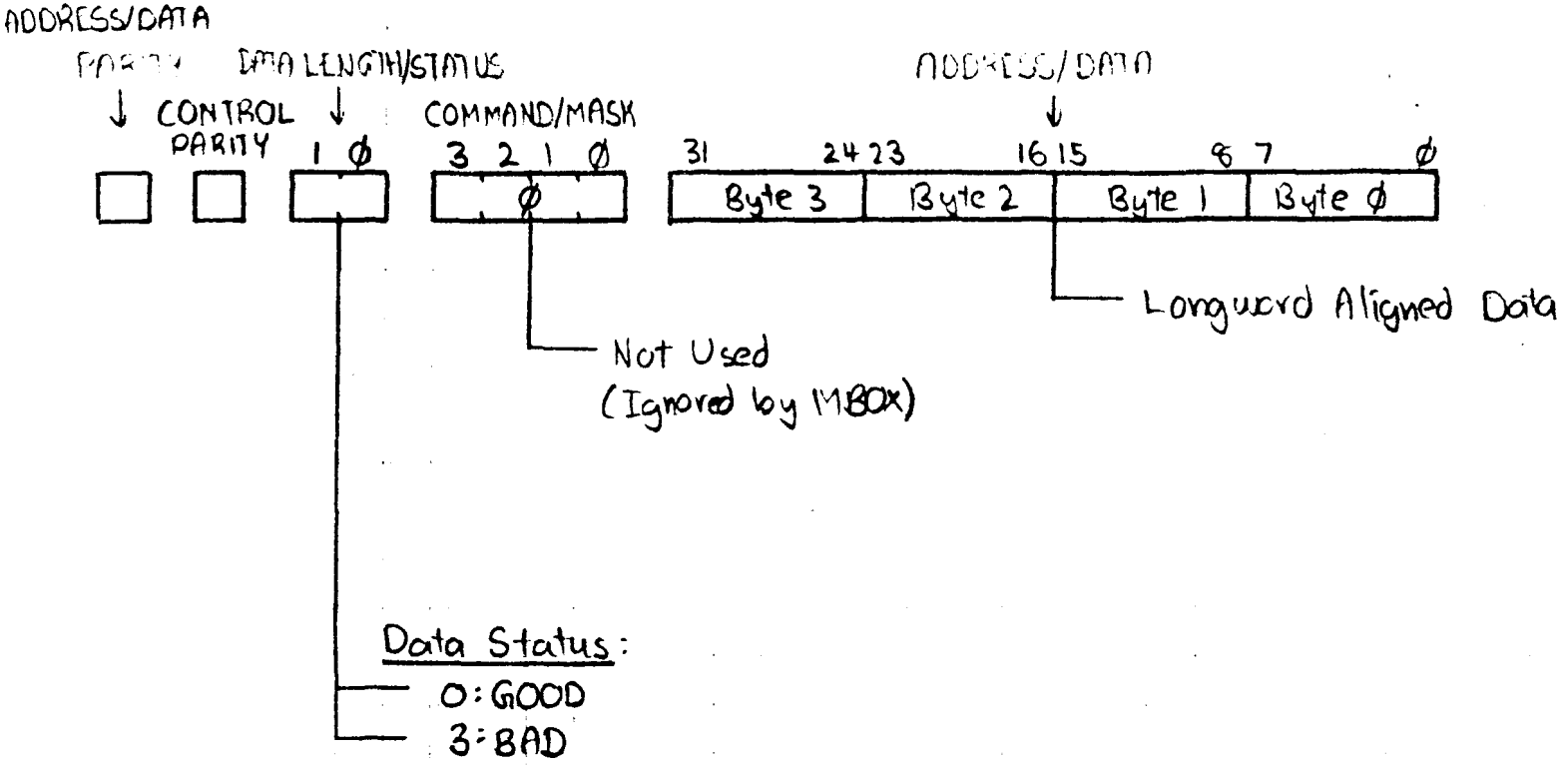


FIGURE 1Ø: READ DATA RETURN CYCLE FORMAT

J. LACY
25-FEB-8Ø

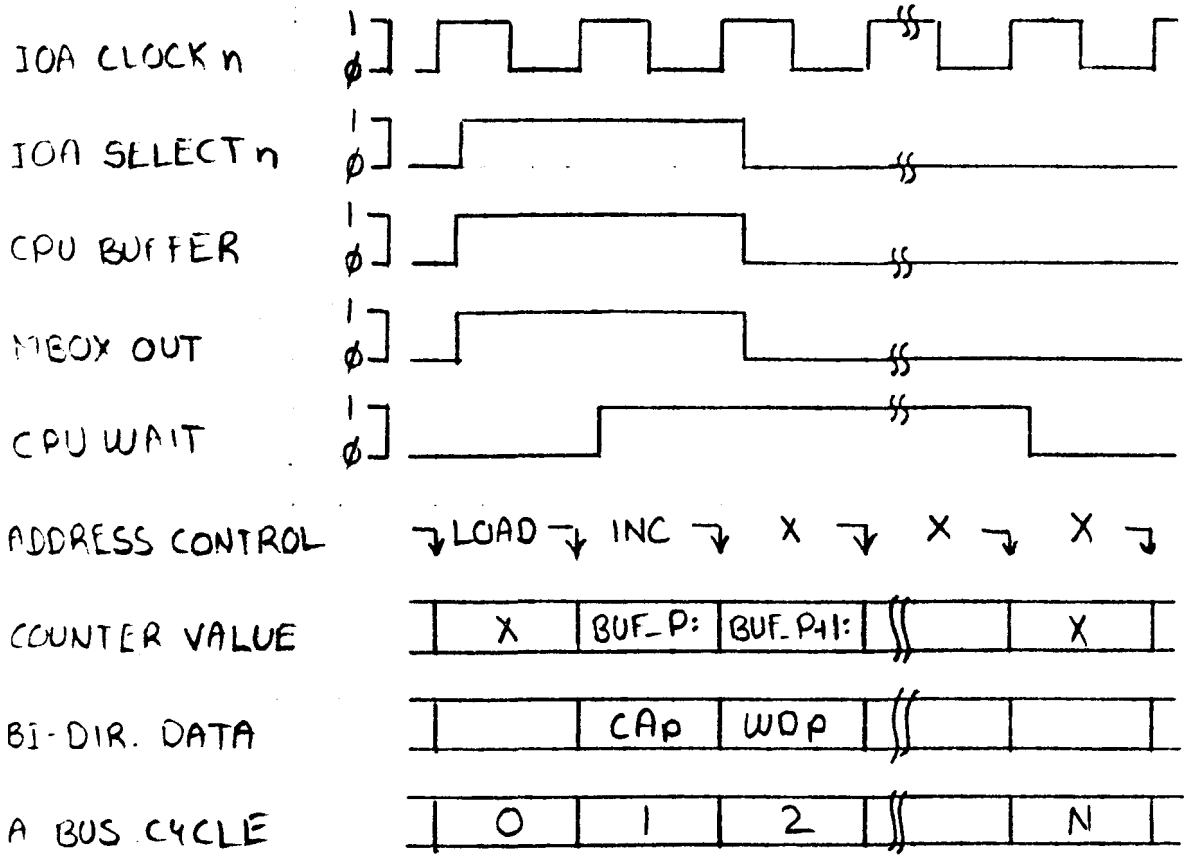


FIGURE 13: CPU WRITE CLASS TRANSACTION

J. LACY
27-FEB-80

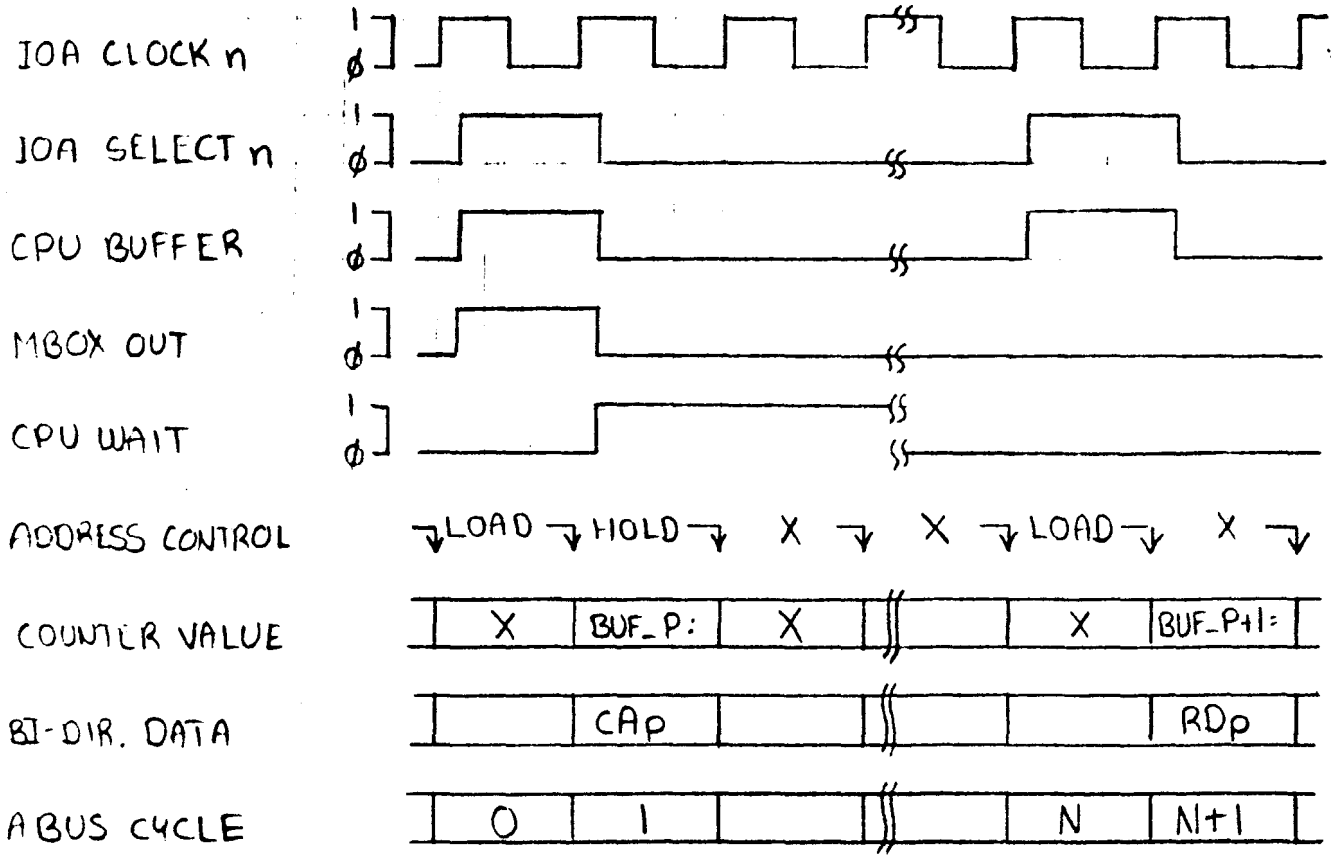


FIGURE 14: CPU READ CLASS TRANSACTION

J. LACY
27-FEB-80

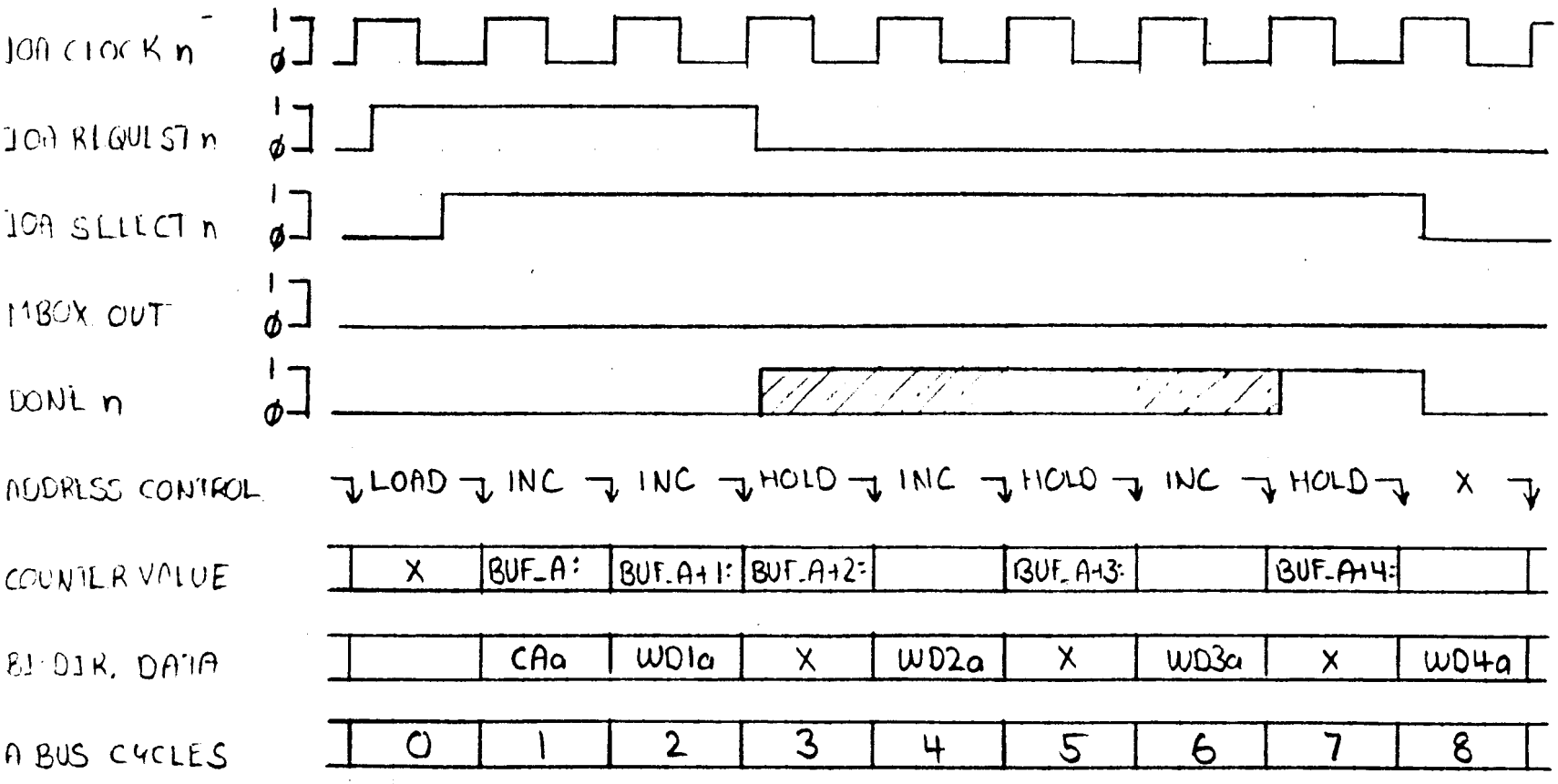


FIGURE 15: DMA WRITE CLASS TRANSACTION

J. LACY
27-FEB-80

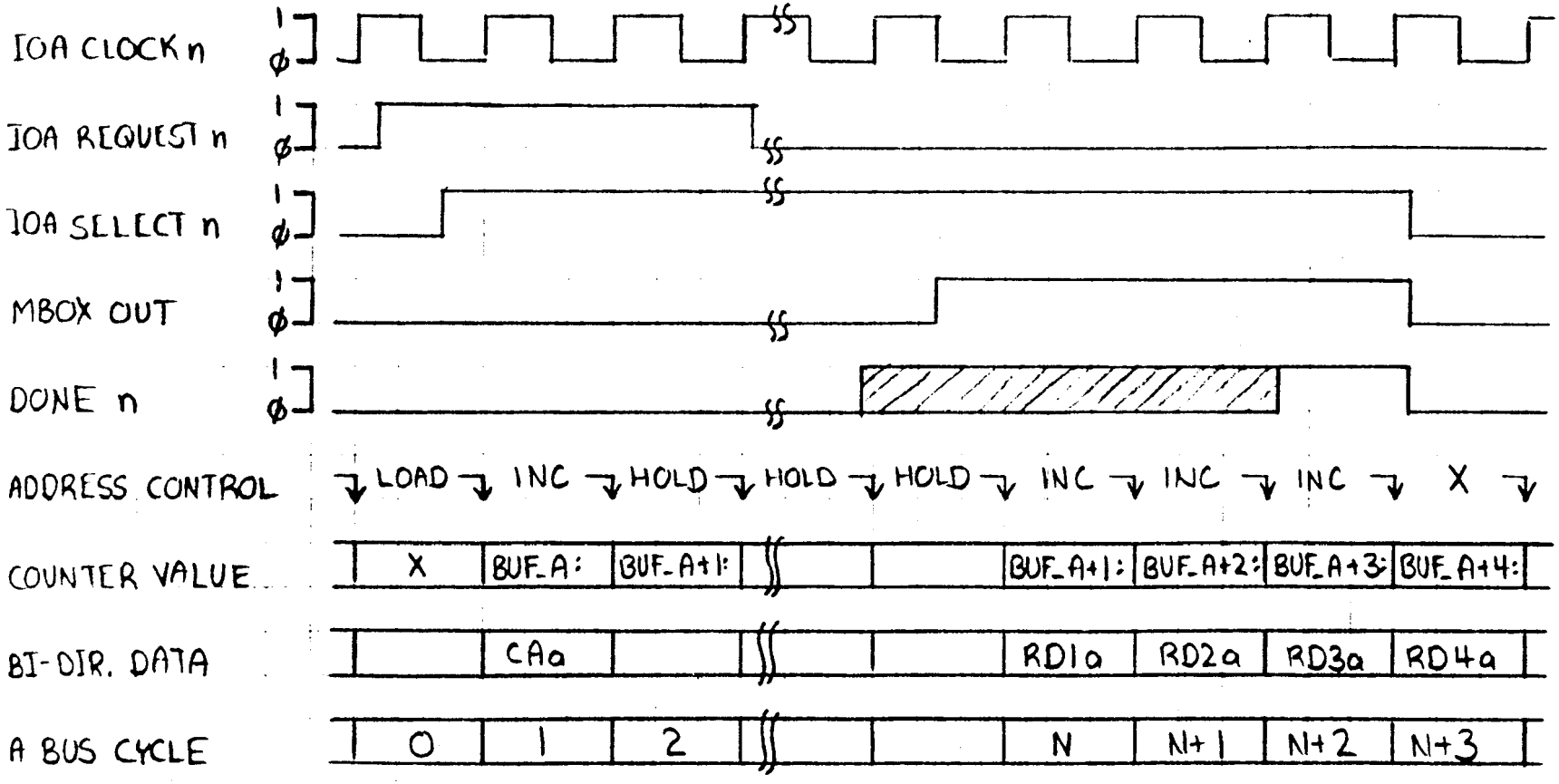


FIGURE 16: DMA READ CLASS TRANSACTION

COMPANY CONFIDENTIAL

J. LACY
28-FEB-80

A BUS THROUGHPUT

1. MAXIMUM DATA RATE TO AN I/O ADAPTER

   - SEQUENTIAL ACCESSES TO SAME MEMORY ARRAY CARD
   - OCTAWORD WRITE = 40.0 MEGABYTES/SEC
   - OCTAWORD READ = 34.3 MEGABYTES/SEC

2. AGGREGATE DATA RATE

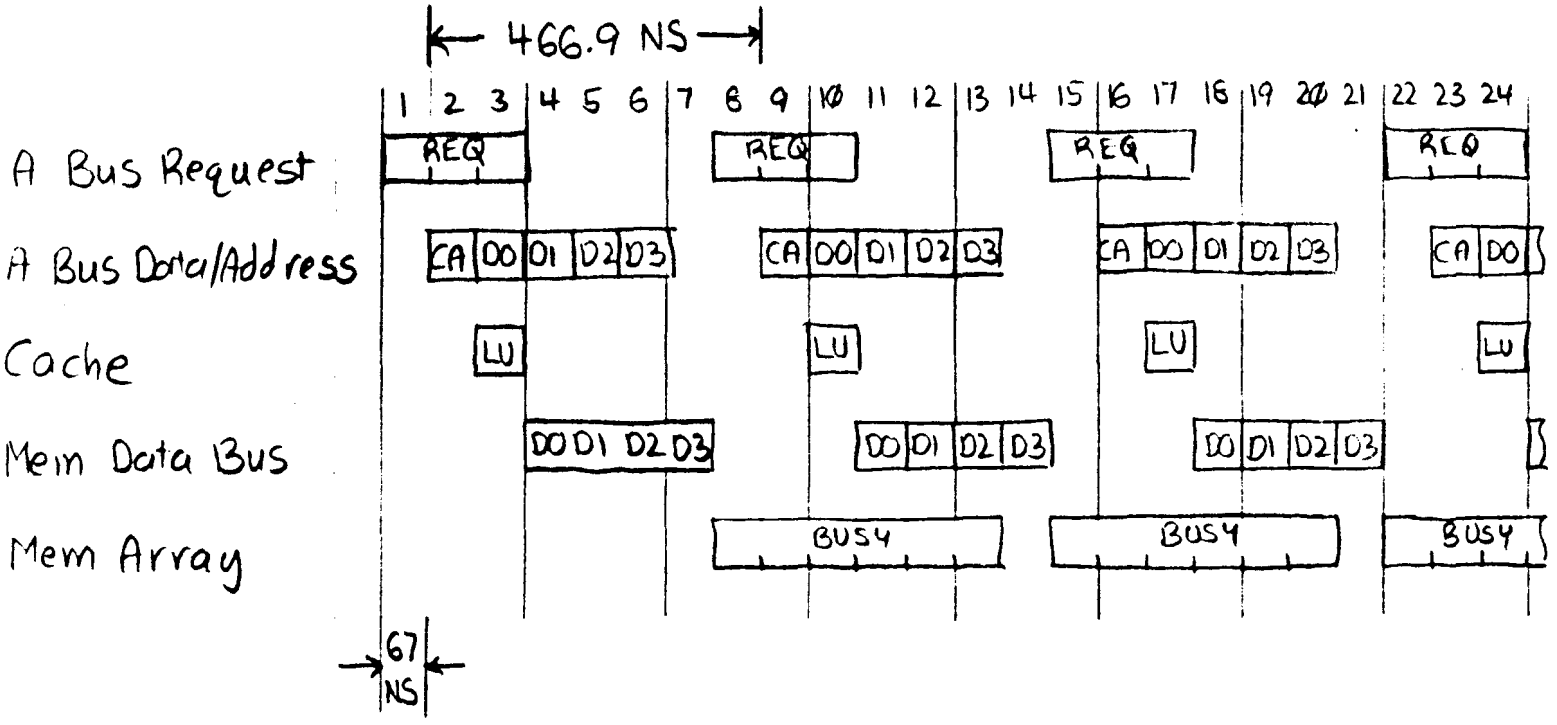
   - 2 I/O ADAPTERS TO DIFFERENT MEMORY ARRAY CARDS
   - OCTAWORD WRITE = 34.3 MEGABYTES/SEC
   - OCTAWORD READ = 18.5 MEGABYTES/SEC

3. BI ADAPTER DMA DATA RATE

   - OCTAWORD WRITE = 13.3 MEGABYTES/SEC
   - OCTAWORD READ = 7.3 MEGABYTES/SEC

4. SBI ADAPTER DMA DATA RATE

   - COVERED IN BARRY FLAHIVE'S PRESENTATION



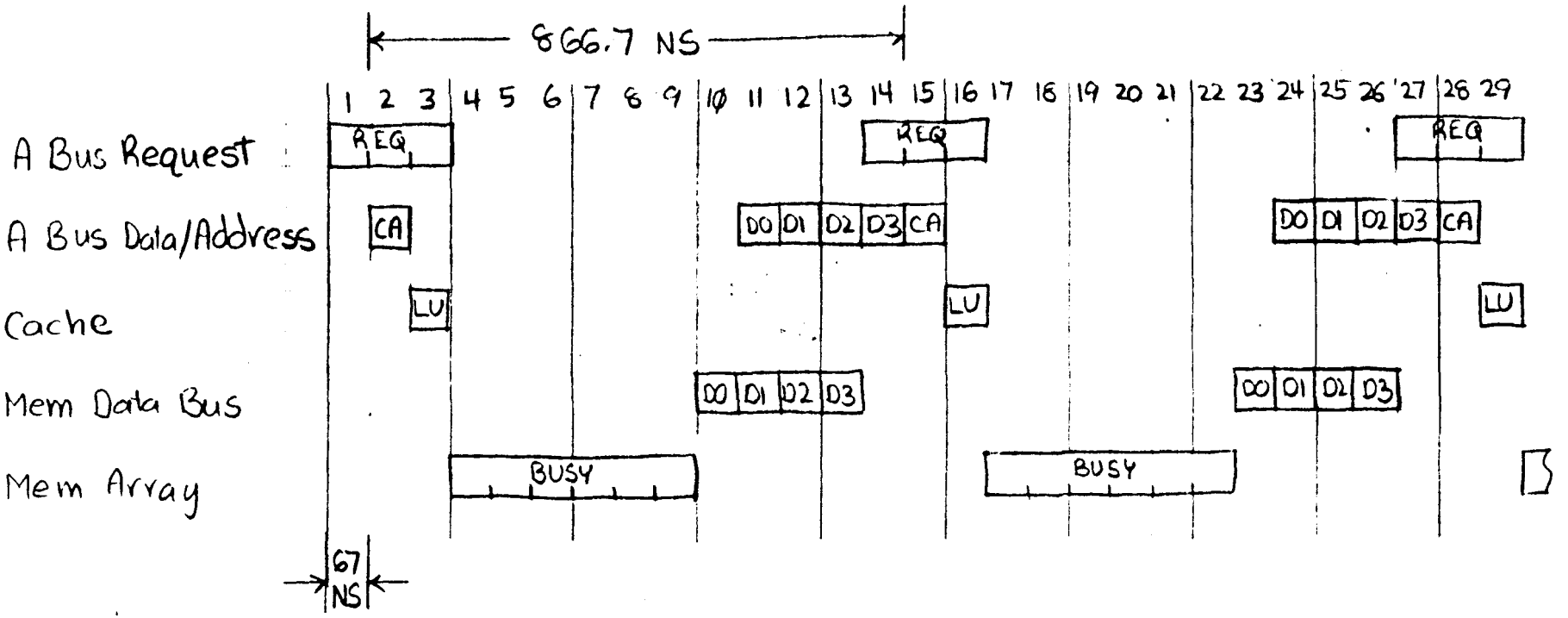
LU = Cache Lock-Up

16 Bytes / 466.9 NS = 34.3 Megabytes/sec

OCTAWORD WRITE TIMING DIAGRAM FOR HIGH SPEED I/O ADAPTER

COMPANY CONFIDENTIAL

J LACY
28-FEB-80



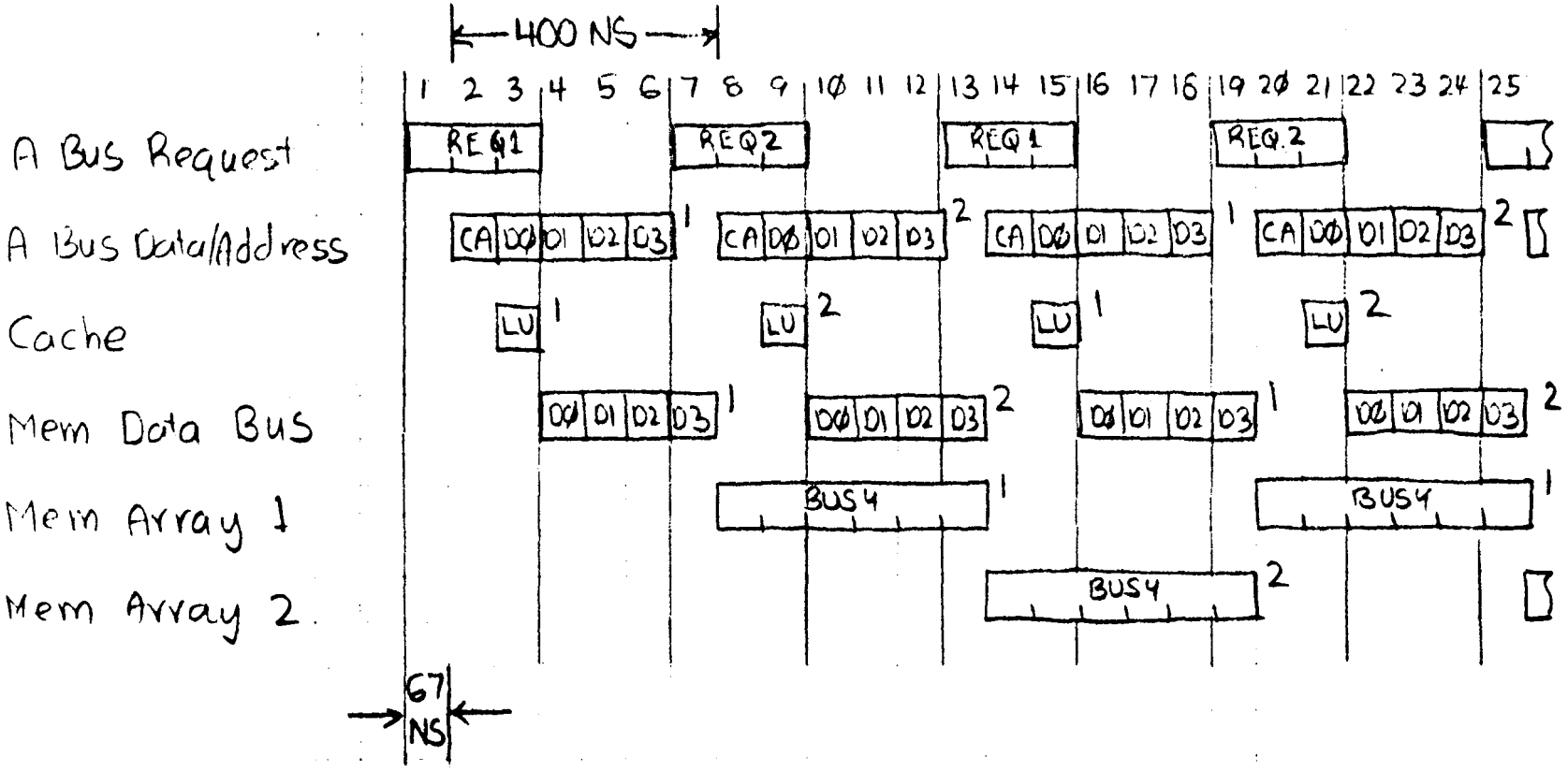
LU: Cache Look-Up

16 Bytes / 866.7 NS = 18.5 Megabytes/sec

OCTAWORD READ TIMING DIAGRAM FOR HIGH SPEED I/O ADAPTER

COMPANY CONFIDENTIAL

J. LACY
28-TLB-82

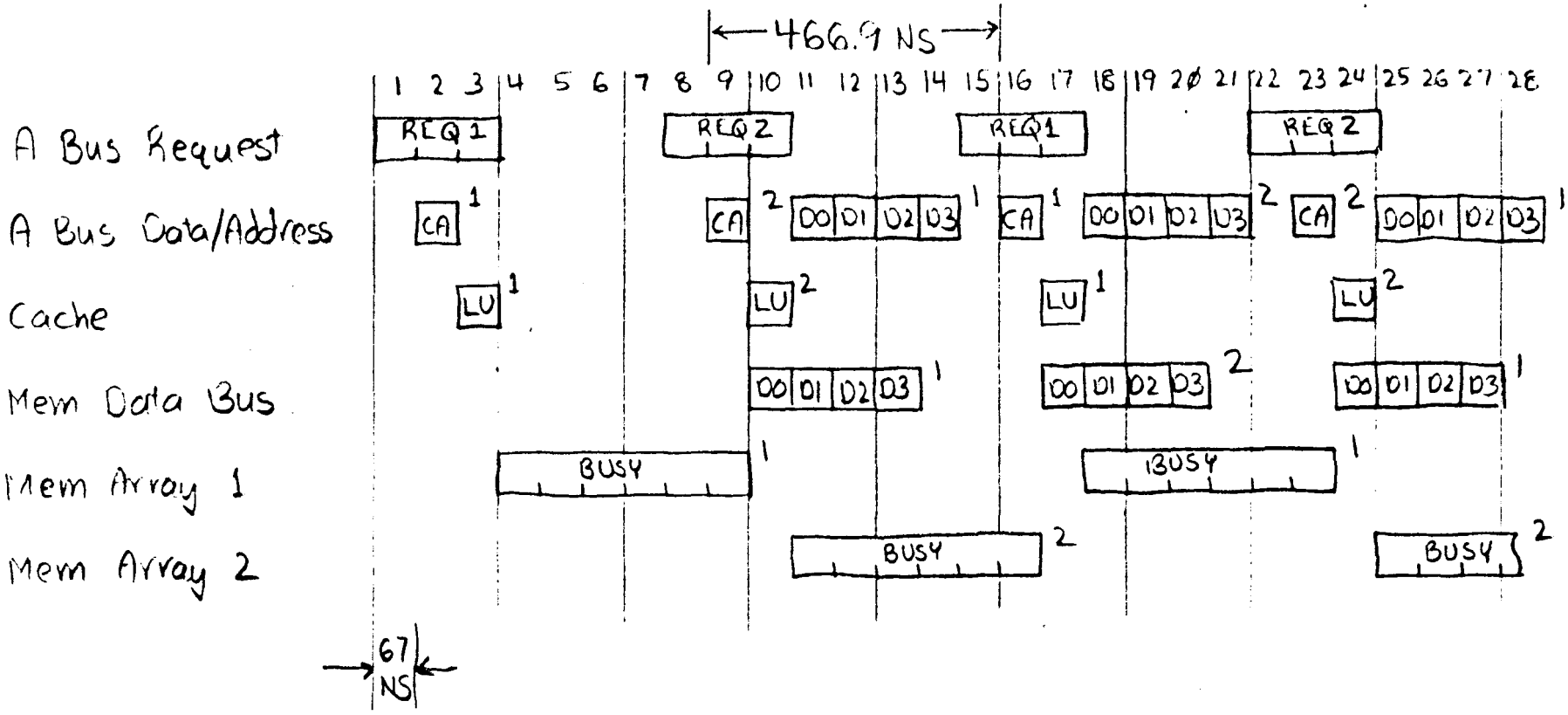


16 Bytes/400 NS = 40.0 Megabytes/Sec

OCTAWORD WRITE TIMING DIAGRAM WITH OVERLAPPING BETWEEN ADAPTERS

COMPANY CONFIDENTIAL

J. LACY
26 FEB-E

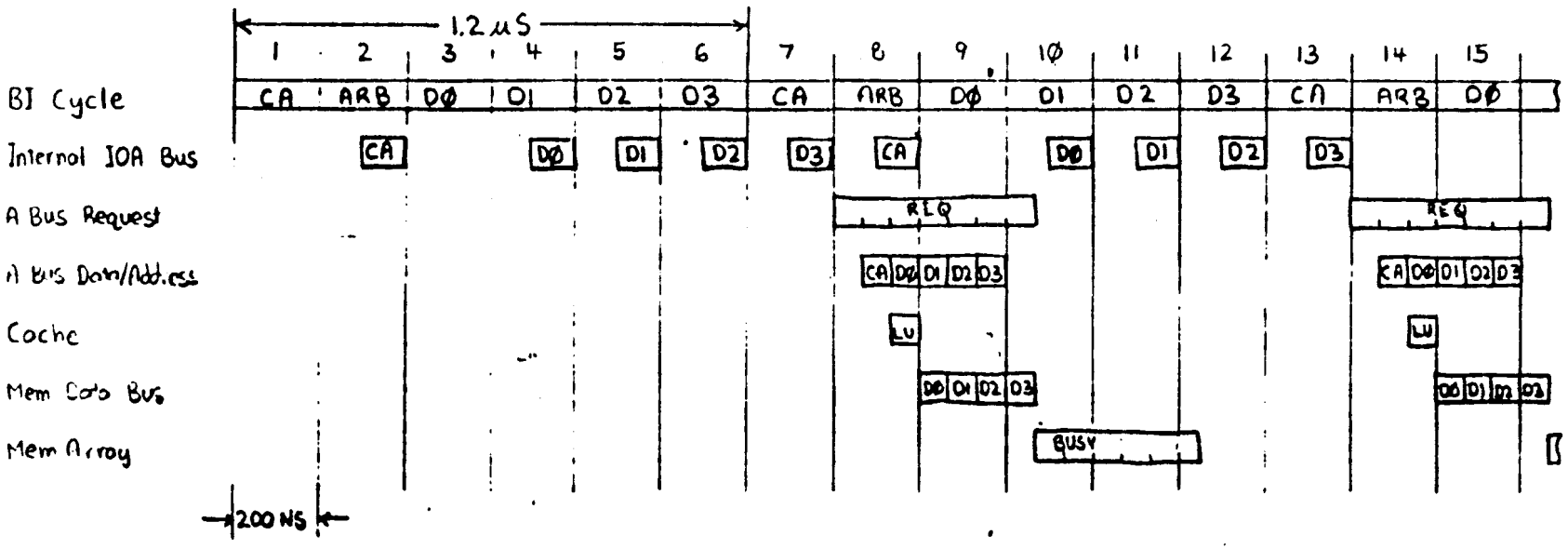


LU = Cache Look-Up

16 Bytes / 466.9 NS = 34.3 Megabytes/sec

OCTAWORD READ TIMING DIAGRAM WITH OVERLAPPING BETWEEN ADAPTERS

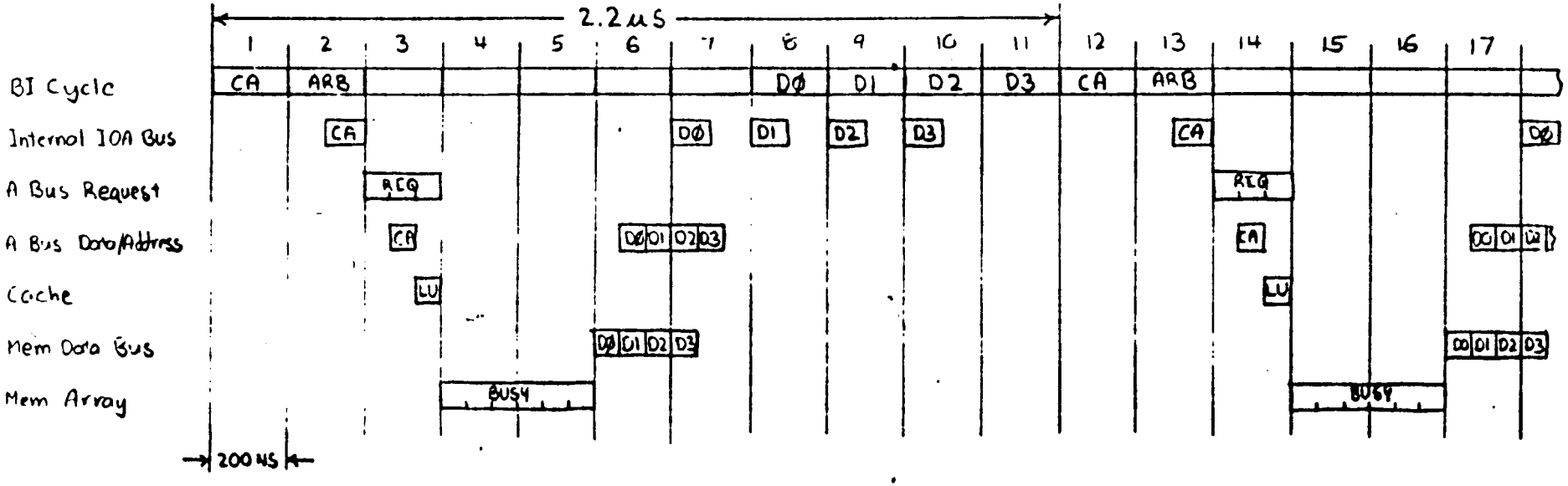
J. LACY
28-FEB-80



LU = Cache Look-Up

FIGURE 11: OCTAWORD WRITE TIMING DIAGRAM FOR BI ADAPTER

J. LACY
25-FEB-82



LU = Cache Look-Up

FIGURE 12: OCTAWORD READ TIMING DIAGRAM FOR BI ADAPTER

J LACY
25-FEB-80

SBI ADAPTER

BARRY FLAHIVE

VENUS SBIA

29-Feb-80
Barry Flahive

1. SBI Overview
2. SBIA Design Goals
3. SBIA in the VENUS System
4. Block Diagram
5. Buffer Organization
6. DMA Operation Timing

Synchronous Backplane Interconnect

- 84 Signal Lines
- 32 Bit Data Path
- 30 Bit Physical Address Space (1 GByte)
- 200 nS Bus Cycle
- Distributed Fixed Priority Arbitration
- 3 Meter Maximum Length
- TTL Open Collector
- 75 Ω Characteristic Impedance

SBI Protocol

- Checked, Parallel Information Exchanges Synchronous with a Common Clock

- Time Multiplexed to allow an Arbitrary Number of Data Exchanges to be in Progress Simultaneously

- Each Clock Period Interconnect Arbitration, Information Transfer, Transfer Confirmation and Interrupt Request Can Occur in Parallel

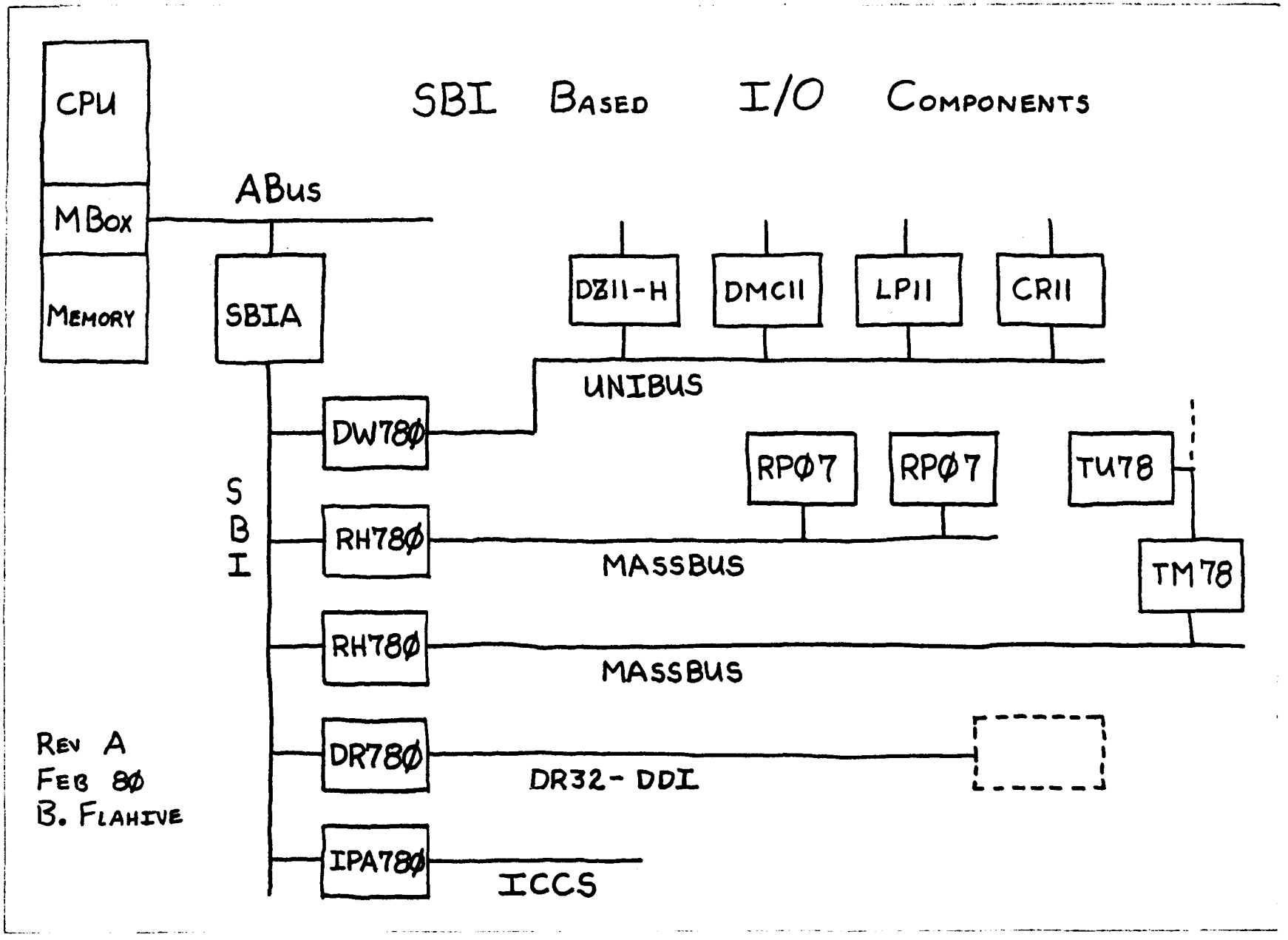
- Single Bit Errors Detected on the Information Path

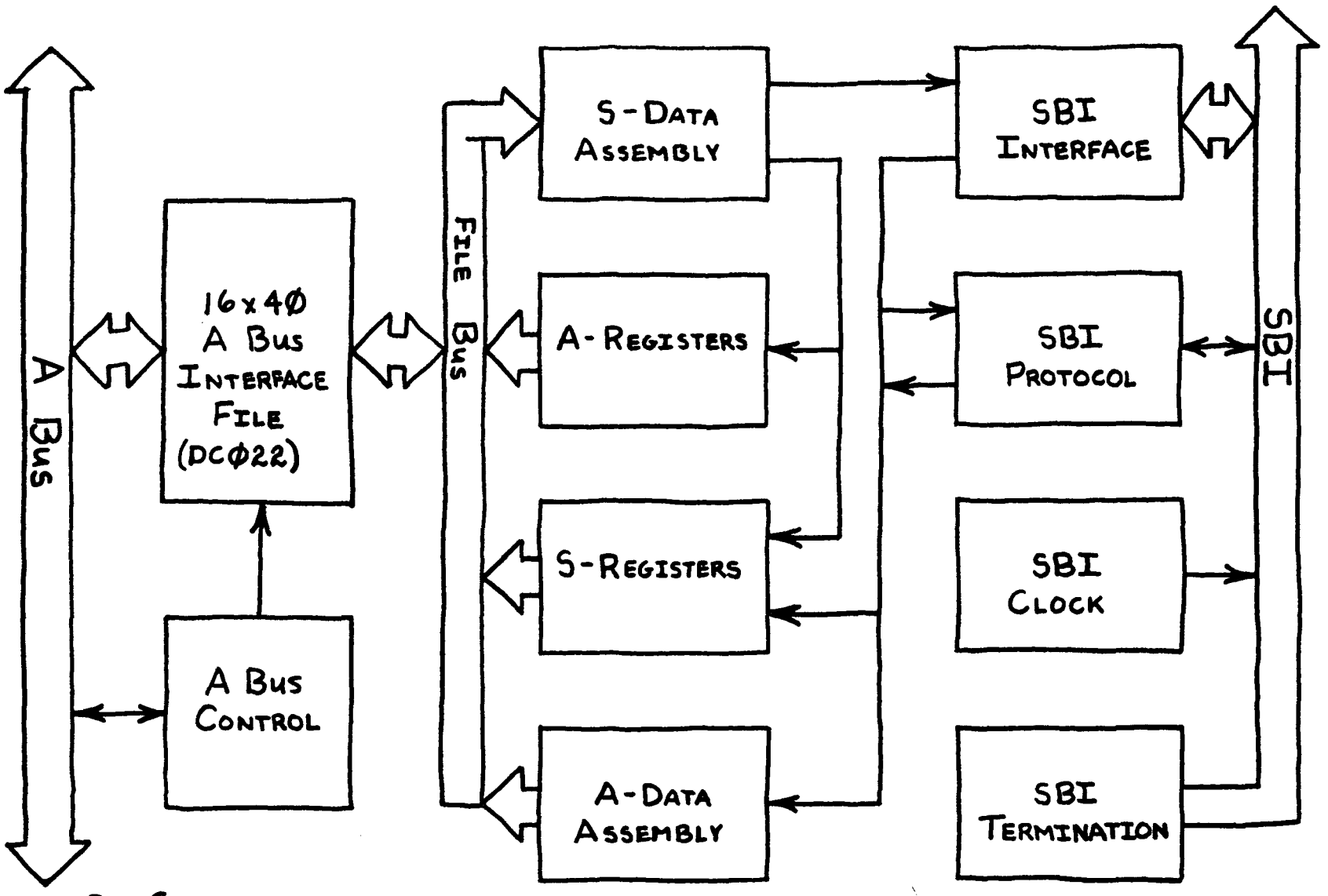
- Automatic Retry of Failed Transfer

- Data Types Byte, Word, Longword, Quadword

SBIA Design Goals

- SBI Based I/O Subsystem with
  - * Massbus
  - * Unibus
  - * DR32
  - * ICCS
- DMA Performance at least Equivalent to 11/780
- Must Work With All SBI Devices Except CPUs
- SBIA Should be Invisible to All Run Time Software Except for Error Service Routines
- SBI Devices and Registers Should Appear the Same as on 11/780

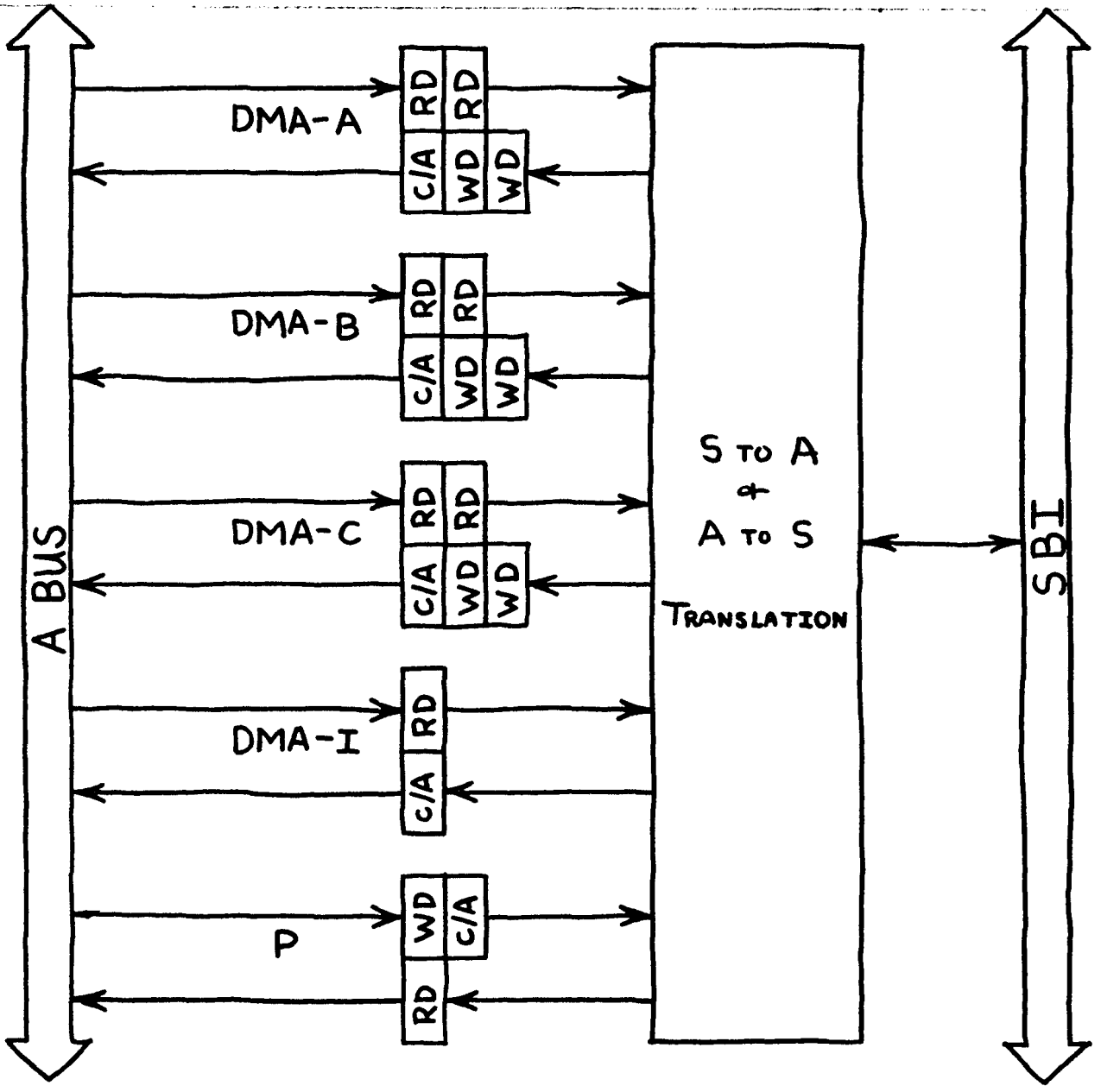




Rev C
Feb 80
B Flanive

SBIA Block Diagram

COMPANY CONFIDENTIAL



SBIA

CONCEPTUAL BUFFER ORGANIZATION

REV C
FEB. 80
B. FLANIVE

FIGURE 3.1

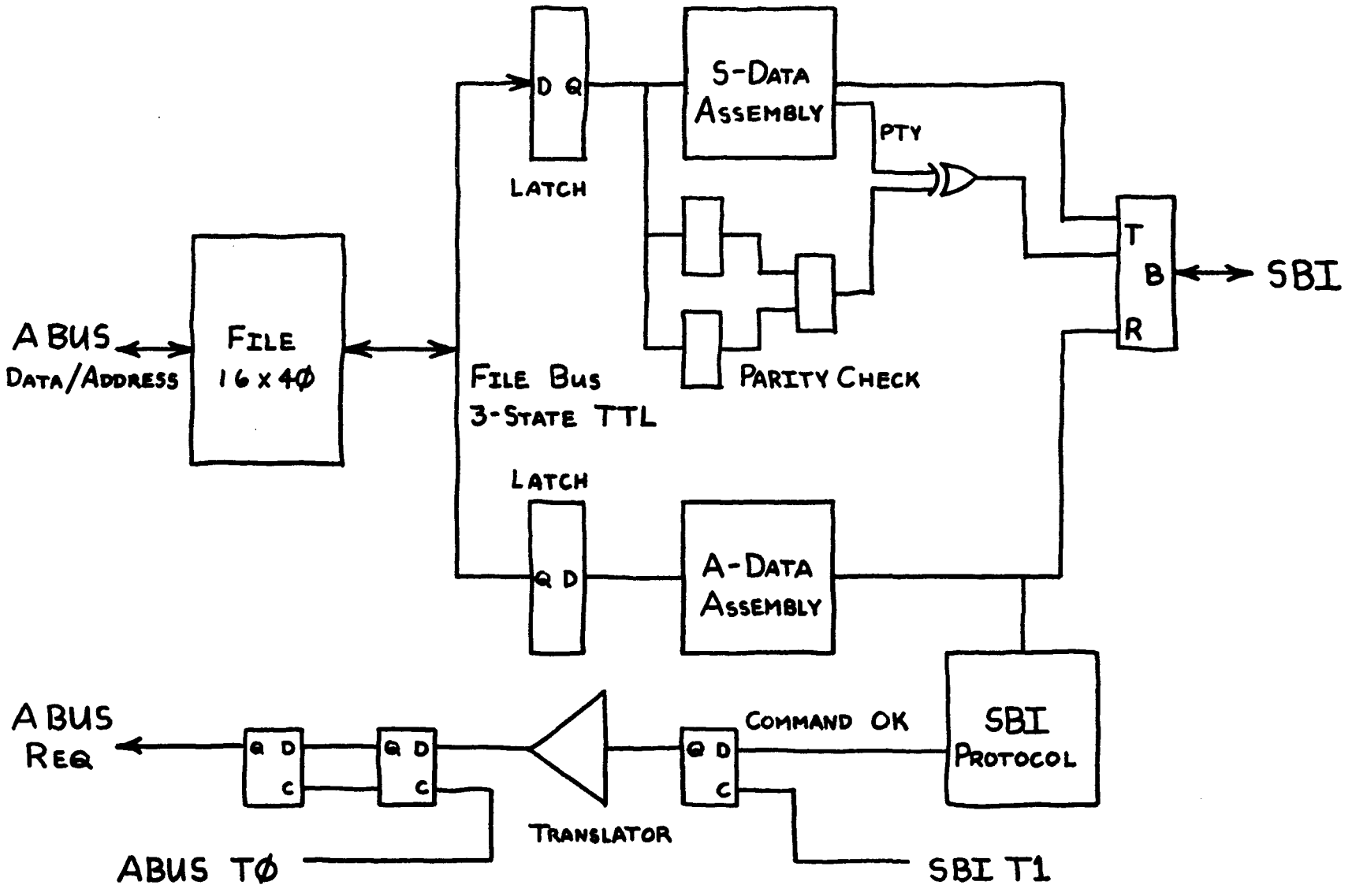
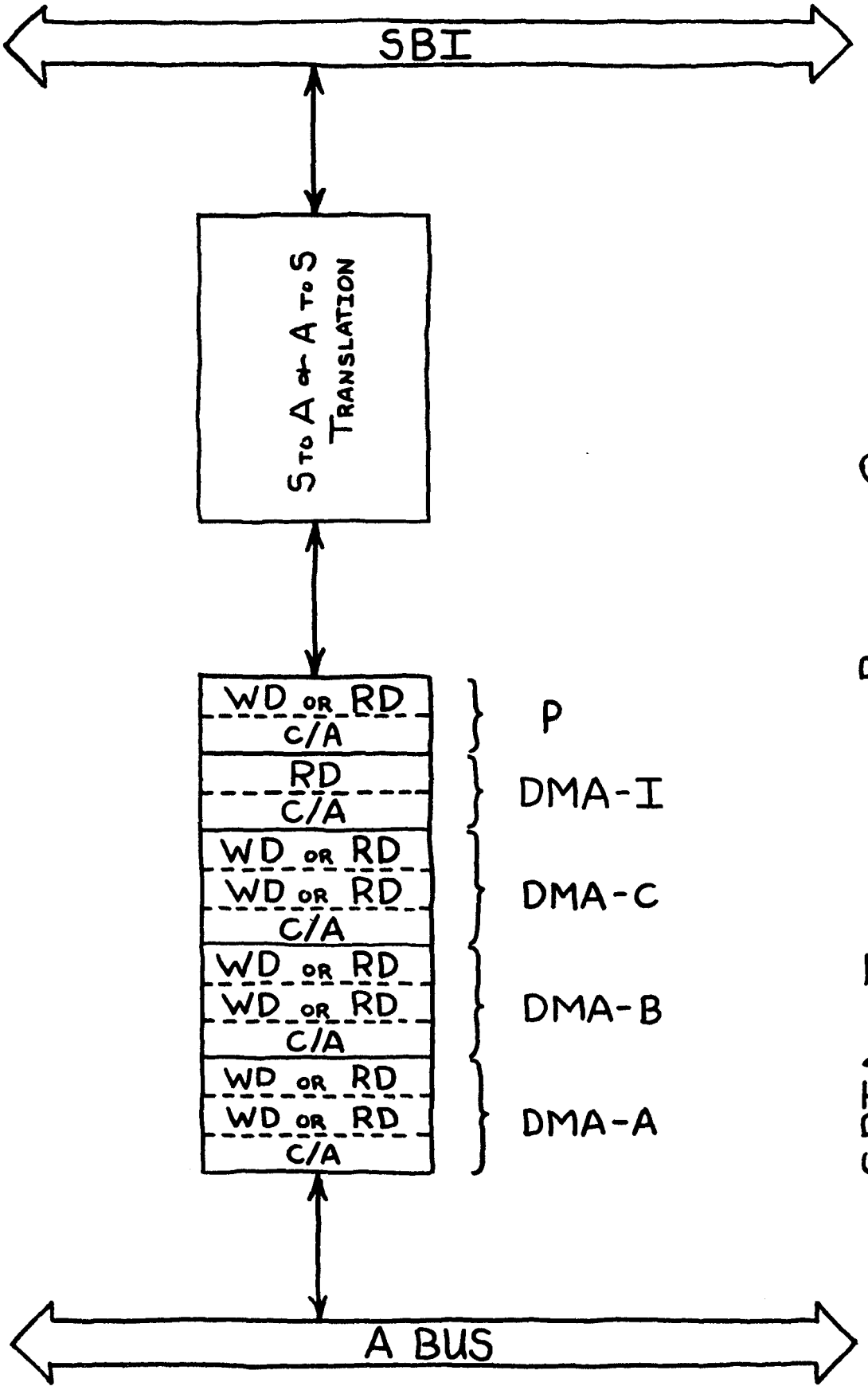


FIGURE 3.3

SBIA DMA DATA PATH

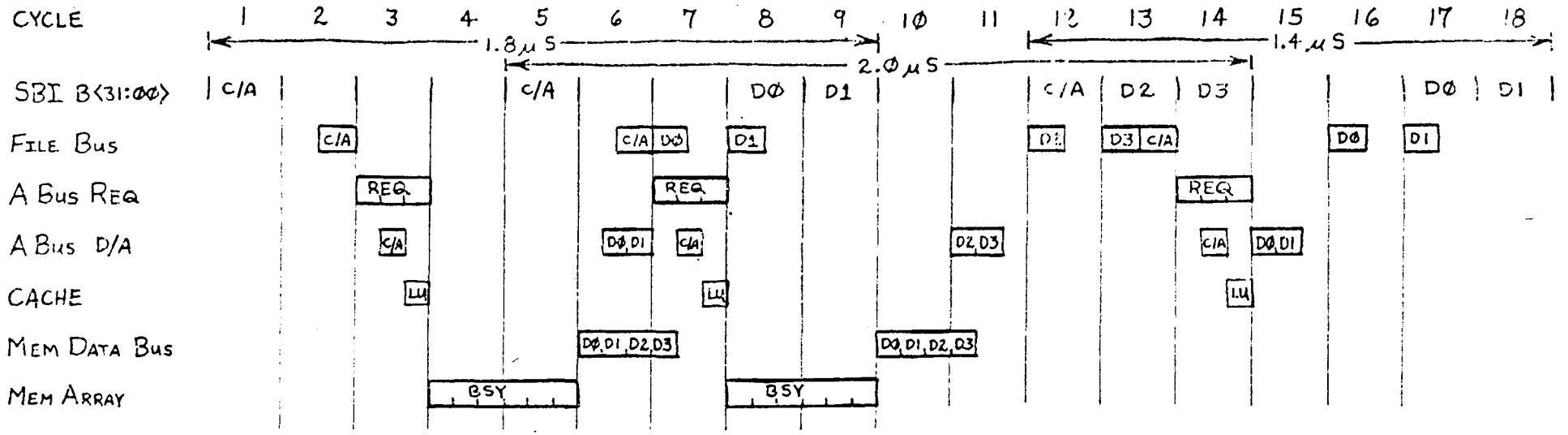
REV B
FEB. 80
B. FLAHIVE



SBIA TRANSACTION BUFFER ORGANIZATION

REV B
FEB. 80
B. FLAHIVE

FIGURE 3.2

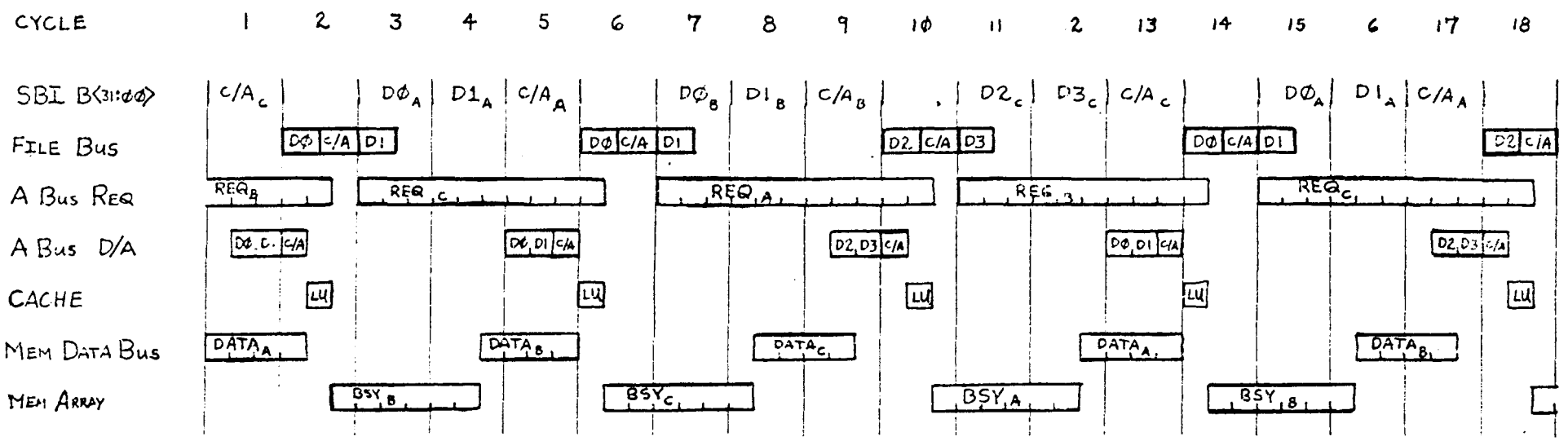


READ ACCESS FROM SBI

LOW QUADWORD – 1.8μS
HIGH QUADWORD – 2.0μS
CACHE HIT – 1.4μS

REV A
FEB. 80
B. FLAHIVE

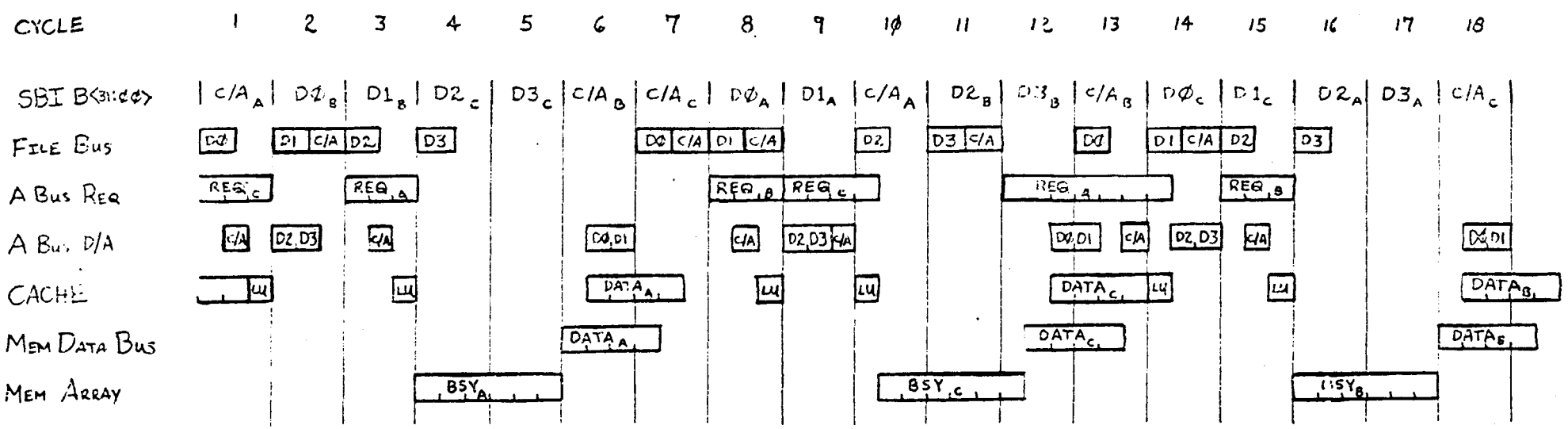
FIGURE 3.4



DMA READ STREAM — NO CACHE HITS

REV A
FEB. 80
B. FLAHIVE

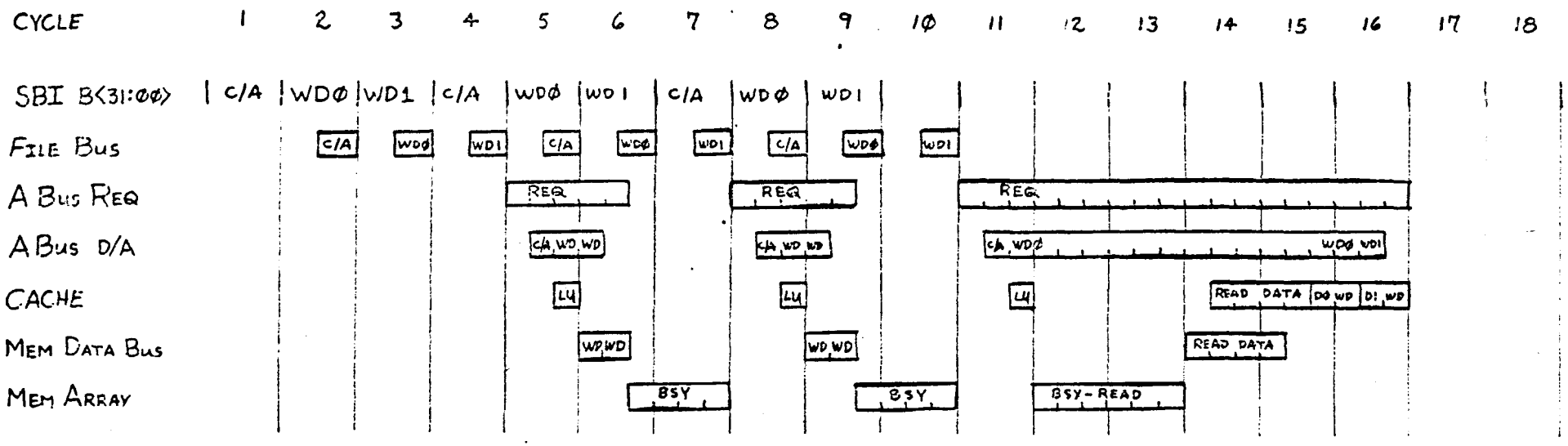
FIGURE 3.5



DMA READ STREAM - 50% CACHE HITS

REV A
FEB 80
B. FLAHIVE

FIGURE 3.6



DMA WRITES AND WRITE MASKED

Rev A
Feb. 80
B. FLAHIVE

FIGURE 3.7

FEB 19 1980

digital

INTEROFFICE MEMORANDUM

TO: List

LOC/MAIL STOP

DATE: 18 FEB 1980
FROM: Sas Durvasula
DEPT: L.C.E.G.
EXT: 231-4426
LOC/MAIL STOP: MR1-2/E47

SUBJ: VENUS CPU READING MATERIAL

Attached, please find the "Venus CPU Reading Material" for your review prior to the February 29th meeting (PK3-1, 8:30 to 5:00).

The SBIA and A-Bus information will be mailed to you on the 25th (Blue Mail).

It is very helpful if all the attendees read the material ahead of time to make the discussions productive for the design team members.

Thank you and contact me if you have any questions in the meantime.

NOTE: Recommended Reading Sequence of the Material:

1. Venus Processor Overview
2. I Box
3. E Box
4. F Box
5. M Box
6. Console
7. RAMP Strategy
8. E/I Boxes Data Path Parity Scheme

attachments

/pas

v.rm 68

+----------------+
! D I G I T A L ! I N T E R O F F I C E M E M O R A N D U M
+----------------+

TO: Venus Technical List

DATE: 28 SEPT 1979
FROM: Paul Guglielmi
DEPT: LCEG
LOC: MR1-2/E47
EXT: 6506

SUBJ: E/Ibox Data Path Parity Plan

The following is a description of the data path parity scheme that we plan to include in the VENUS IBOX and EBOX. The goal of the scheme is to carry parity on all data as long as the expense does not become excessive. The expense is considered excessive when the logic to carry the parity costs more than 20 % of the cost of the logic being checked or results in new MCA types.

By carrying parity on data it will be possible to isolate detected failures to the major unit which was the cause of the failure. This is important when the failure is intermittent because this class of errors is usually not reproducible when diagnostics are run.

The scheme that is described herein results in no new MCA types in either the EBOX or IBOX. In the IBOX an extra copy of an existing MCA and 8-10 10K ECL DIPS are added to support the data path parity. In the EBOX spare cells in the VMQ MCA are used to create a parity data path for the EBOX. The remaining half of the ALU carry lookahead MCA is used to predict ALU parity. Thus 1.5 MCA's and 8-10 10K ECL parts are added to the EBOX data path to support data path parity. In addition to this some control logic must be added to make the parity useful. We feel that this cost is approximately one MCA worth of cells scattered among the existing control MCA's.

Therefore, for the addition of 2.5 MCA's, no new MCA types, and 16-20 10K ECL parts we are able to check 24 data path MCA's.

When data path parity errors are detected the the processor will take a machine check. In most cases enough information will be pushed onto the interrupt stack to allow instruction retry by the system software. Included in this information will be a bit indicating whether the instruction has modified any state. If the bit is off the instruction will be retryable. The retry should be successful if the parity error is a transient error. If the error is a solid

fault then the hardware is broken and the machine must be fixed.

1.0 IBOX PARITY

The IBOX is shown in the right half of figure 1. Its data/address path consists of 8 ADR MCA's, 9 Byte Buffer MCA's, and some ECL 10K mixers that do sign extension, zero extension and short literal unpacks.

The 8 ADR MCA's form the address arithmetic data path for the IBOX. It contains three adders, Current PC register, Virtual Instruction Buffer Address register, Virtual Address register, GPR data latch, EBOX PC register, IBOX PC register, and the IBOX W register.

The Current PC register points to the place in the I-Stream on which the IBOX is currently working. The VIBA register contains the address of the next group of bytes that are to be loaded into the Instruction Buffer register. The VA register is used to hold the address of operands that the IBOX is fetching for the EBOX and is occasionally used to hold GPR data that is to be passed to the EBOX. The GPR data latch latches data read out of the GPR's in the first half of a microcycle so that the GPR's may be written during the last half of the microcycle. The EBOX PC register contains the PC of the instruction the EBOX is currently working on so that the instruction may be backed up in case of a exception. The IBOX PC register contains the PC of the instruction the IBOX is currently working on for use in exception handling. The IBOX W register is used to hold the result of auto increment and auto decrement address calculations for writing into the GPR's over the W BUS and to hold the index so it can be multiplied by the context for indexed address calculations via the shift path into the A input of the three input ALU.

The IBOX talks to the MBOX over the VA BUS and the MD BUS.

1.1 VA BUS

Virtual addresses are sent to the MBOX over the VA BUS. This bus is not parity checked because the cost of carrying parity through the IBOX ADR MCA was deemed to be excessive compared to the benefits.

1.2 Writes Over MD BUS

Data is sent to and received from the MBOX over the Memory Data Bus (MD BUS). All transfers over this bus carry byte parity.

Data going to memory comes to the IBOX from the WBUS with byte parity appended. The data proceeds from the WBUS through the WR latch to the byte rotator. Data from the WBUS is always right justified and 1, 2, 3, or 4 of the bytes may want to be written into memory. The byte rotator looks at the low two bits of the address on the VA BUS and rotates the bytes so that they are in the correct position on the MD BUS. The byte parity bits are rotated also so that the correct parity bit accompanies each byte. The correct number of data valid bits are also set on the WBUS. The data valid bits are sent over the MD BUS to control byte writes in the MBOX.

No parity check is performed on write data in the IBOX. The data is parity checked by the MBOX before it is written into the cache or memory.

If the MBOX detects a parity error on the data it will not write the data and will cause the EBOX to take a microtrap indicating that the write never completed.(Note: The IBOX never writes memory; only the EBOX and FBOX write memory) This will result in a machine check to the macro software. The data placed on the interrupt stack will indicate whether or not the instruction for which the data was being written is retryable. The type of error, and the source and address of the data will also be placed on the stack.

If instruction retry is possible it must be initiated by the macro software.

Note that if a parity error was detected by the MBOX the error must have occurred in the IBOX or on the MD BUS because the EBOX checks the parity of all data on the WBUS.

1.3 Reads Over The MD BUS

Two types of data are read from the MBOX over the MD BUS: D-stream data, I-stream data.

D-stream data comes from the cache over the MD BUS to the MD register in the IBOX with byte parity appended to the data. On the way to the MD register the data passes through a byte rotator where the data is right justified. The data is then passed through sign extender multiplexers and on to the EBOX and FBOX. At this time the byte parity is collapsed to longword parity. As a result the EBOX and FBOX

see data with longword parity.

The IBOX does not check the parity of data in the MD register. The parity check is done in the EBOX or FBOX as appropriate. The parity of the data on the MD BUS is checked by the MBOX when it is driving the bus.

If either the EBOX or FBOX detects a parity error then the error must have occurred in the IBOX or in the path between the IBOX and EBOX or FBOX.

If I-stream data is being read from the MBOX then the MD BUS data will include byte parity. The data will be sent through the same byte rotator as the D-stream data and will then be loaded into the Instruction Buffer (IBUF) an 8 byte register. Data is loaded into free byte locations in the left end of the IBUF register starting at the rightmost non-valid byte. Data is removed from the register by shifting it right one or more bytes at a time as the IBOX processes opcodes and specifiers.

The specifier decode logic attached to the IBUF has byte parity checking. The parity of the opcode byte B0 is also parity checked.

If a parity error is detected on any of these bytes a machine check will be taken and information relevant to the source of the error will be pushed onto the interrupt stack.

From the IBUF I-stream data can be routed to the B input of the three input ALU in the ADR MCA. The B input to this ALU has a parity check network. This parity checker checks I-stream data that comes out of IBUF bytes B1 thru B5 when the IBOX is taking byte, word and longword address displacements from the I-stream. Any data from the MD register or IBUF that is passed to the B input of the ALU will be checked by this network if it has valid byte parity.

If a parity error is detected by this parity network a machine check will be taken the next time the IBOX tries to do a D-stream memory reference. Information about where the error was detected will be pushed onto the interrupt stack.

D-stream data may also be read using the IBUF. This is done during string operations where the IBUF is used as a string fetching facility. All data is removed from the four low-order bytes of the IBUF, B3 through B0. This data is then passed to the Ebox with longword parity. The longword parity is computed from the parity of each byte in the Ibox and it is checked in the Ebox or Fbox.

1.4 Reads From IBOX GPR's

Data stored in the IBOX GPR's has byte parity. When data is read from the GPR's the parity of the data is checked and if an error is detected a microtrap will be taken the next time the EBOX tries to get an operand. The IBOX will stall until the EBOX corrects the fault.

While the IBOX is stalled the EBOX will determine if the parity error exists in its copy of the GPR's.

If there is no error in the the EBOX's copy then the EBOX will write good data into all copies of the GPR's. This will restore the data in the IBOX GPR's if the error was a soft error. The EBOX will then restart the IBOX and if the error was soft the IBOX will be successfully continued. If the error was a hard error then the IBOX will detect a parity error on the read and the EBOX will take a second microtrap. This will cause a machine check to be taken indicating that the IBOX GPR's have a hard parity error.

If the IBOX reads data out of the GPR's that is to be passed to the EBOX or FBOX the byte parity from the GPR's will be turned into longword parity and sent along with the data to the EBOX and FBOX. The EBOX and FBOX will check the longword parity of this data.

Anytime the IBOX writes a GPR it will generate byte parity for the data to be placed on the W BUS.

2.0 EBOX PARITY

The EBOX data path is shown in the left half of figure 1. The data path consists 9 MCA'S used to implement the ALU one of which is the carry lookahead MCA, 8 VMQ MCA's which contain most of the EBOX registers, and a shift matrix which is implemented with 2 SHIFTER MCA's and 8 10173 two way multiplexers. Not shown is a version of the VMQ MCA which implements the data path for the parity bit. The EBOX has longword parity throughout most of its data path.

2.1 AMUX And BMUX Parity Checking And Generation

Data from the IBOX arrives at the AMUX and BMUX inputs of the ALU MCA. If the data is coming from memory or from the IBOX GPR's it will have long word parity. If the data is an address from the IBOX then no parity will be provided and the EBOX will have to generate the parity. The IBOX will supply a signal along with the data that will tell the EBOX if parity has to be generated.

To allow the data going into the ALU A and B inputs to have parity there are parity generator/checkers on the outputs of the AMUX and BMUX.

Data arrives at the inputs to the ALU AMUX and BMUX from one of four sources: IBOX, EBOX scratchpads, EBOX QMUX, WBUS. The IBOX supplies parity for data operands and does not supply parity for address operands. The EBOX scratchpads store byte parity which gets converted to longword parity before going into the ALU AMUX and BMUX. The QMUX supplies longword parity. The WBUS byte parity gets converted to longword parity before the data goes to the AMUX and BMUX.

If the data arrives at the AMUX or BMUX with bad parity then the EBOX will initiate a machine check and will push data on the stack indicating which checker detected the error, multiplexer selection, and whether or not the instruction is restartable.

If the data comes from the EBOX scratchpads then the machine check will be held off until the EBOX determines whether or not the parity error exists in both scratchpads.

If the parity error exists in only one copy then the EBOX will determine which bit is bad by EXORing the good and bad copy of the data. This information will be written into a scratchpad error logging location. Then the bad location will be rewritten with the good copy of the data. The reference will then be retried.

If it succeeds no machine check will be taken.

If it fails the machine check will be taken.

If the error exists in both scratchpads then the machine check will be taken immediately.

If no error is detected at the AMUX or BMUX then the data can go either to the ALU or to the shifter.

2.2 Parity And The Shifter

If the data goes to the shifter the parity will be dropped and recomputed at the output of the shifter. This is done because the MCA count for the shifter would have to go from 2 to 4 in order to carry parity through the shifter.

2.3 ALU Parity Prediction

If the data goes thru the ALU then the parity will be predicted from the input parity and the operation being performed. This parity is carried along with the data through the rest of the EBOX data path. The SHF CNT, W, and VMQ registers all have longword parity.

If the ALU is performing a logic function then the parity of the result of the AND, OR, and XOR functions are all generated. Separate circuits are used to compute the parity of each of these operations within the ALU. Next the logic identity

PAR(a OR b)=PAR(a XOR b) XOR PAR(a AND b)

is used to check that the correct parity has been generated for the logic function that was performed. If the identity is satisfied the hardware has functioned correctly.

If an error is detected when the above check is being made a machine check will be taken.

2.4 WBUS Parity

Byte parity is computed for data in the W register and these parity bits are placed on the WBUS when the EBOX writes memory or GPR's. The parity of the bytes is checked against the longword parity for the W register and if an error is detected a machine check is taken.

The EBOX also has a parity checker that looks at the parity of the data on the WBUS. This checker checks all data sent over the WBUS for the EBOX, FBOX, and IBOX.

If a parity error is detected on the WBUS the write on the WBUS will be aborted and a machine check will be taken. This will prevent all copies of the GPR's from being written with bad data.

2.5 VA BUS

No parity is implemented on the VA BUS because the IBOX is unable to generate VA BUS parity.

3.0 FBOX PARITY

The parity scheme for the FBOX data path is not worked out yet.

However,a minimum implementation of FBOX data path parity will at least check the parity of data sent over the operand bus from the IBOX to FBOX, and the parity of data data sent over the WBUS to the FBOX from the EBOX or IBOX.

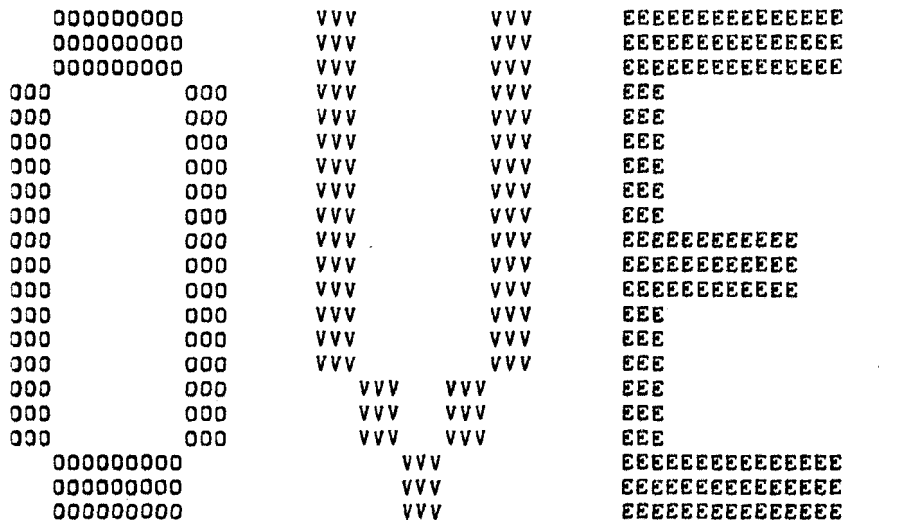
Also when the FBOX puts data on the WBUS it must generate byte parity for that data.

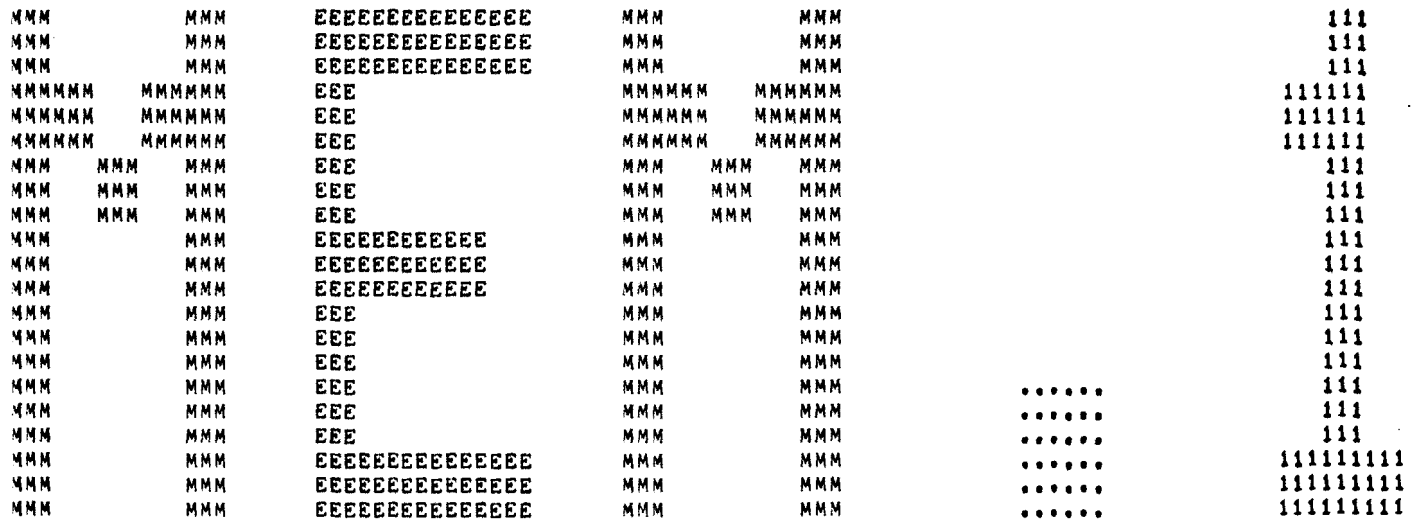
```
   000000000         VVV         VVV      EEEEEEEEEEEEEEE       RRRRRRRRRRRR
   000000000         VVV         VVV      EEEEEEEEEEEEEEE       RRRRRRRRRRRR
   000000000         VVV         VVV      EEEEEEEEEEEEEEE       RRRRRRRRRRRR
000         000      VVV         VVV      EEE                   RRR         RRR
000         000      VVV         VVV      EEE                   RRR         RRR
000         000      VVV         VVV      EEE                   RRR         RRR
000         000      VVV         VVV      EEE                   RRR         RRR
000         000      VVV         VVV      EEE                   RRR         RRR
000         000      VVV         VVV      EEE                   RRR         RRR
000         000      VVV         VVV      EEEEEEEEEEEE          RRRRRRRRRRRR
000         000      VVV         VVV      EEEEEEEEEEEE          RRRRRRRRRRRR
000         000      VVV         VVV      EEEEEEEEEEEE          RRRRRRRRRRRR
000         000      VVV         VVV      EEE                   RRR   RRR
000         000      VVV         VVV      EEE                   RRR   RRR
000         000      VVV         VVV      EEE                   RRR   RRR
000         000         VVV   VVV         EEE                   RRR      RRR
000         000         VVV   VVV         EEE                   RRR      RRR
000         000         VVV   VVV         EEE                   RRR      RRR
   000000000               VVV            EEEEEEEEEEEEEEE       RRR         RRR
   000000000               VVV            EEEEEEEEEEEEEEE       RRR         RRR
   000000000               VVV            EEEEEEEEEEEEEEE       RRR         RRR
```

FEB 19 1980

```
MMM         MMM      EEEEEEEEEEEEEEE      MMM         MMM                                  111
MMM         MMM      EEEEEEEEEEEEEEE      MMM         MMM                                  111
MMM         MMM      EEEEEEEEEEEEEEE      MMM         MMM                                  111
MMMMMM   MMMMMM      EEE                  MMMMMM   MMMMMM                               111111
MMMMMM   MMMMMM      EEE                  MMMMMM   MMMMMM                               111111
MMMMMM   MMMMMM      EEE                  MMMMMM   MMMMMM                               111111
MMM   MMM   MMM      EEE                  MMM   MMM   MMM                                  111
MMM   MMM   MMM      EEE                  MMM   MMM   MMM                                  111
MMM   MMM   MMM      EEE                  MMM   MMM   MMM                                  111
MMM         MMM      EEEEEEEEEEEE         MMM         MMM                                  111
MMM         MMM      EEEEEEEEEEEE         MMM         MMM                                  111
MMM         MMM      EEEEEEEEEEEE         MMM         MMM                                  111
MMM         MMM      EEE                  MMM         MMM                                  111
MMM         MMM      EEE                  MMM         MMM                                  111
MMM         MMM      EEE                  MMM         MMM                                  111
MMM         MMM      EEE                  MMM         MMM             ......               111
MMM         MMM      EEE                  MMM         MMM             ......               111
MMM         MMM      EEE                  MMM         MMM             ......               111
MMM         MMM      EEEEEEEEEEEEEEE      MMM         MMM             ......            111111111
MMM         MMM      EEEEEEEEEEEEEEE      MMM         MMM             ......            111111111
MMM         MMM      EEEEEEEEEEEEEEE      MMM         MMM             ......            111111111
```

```
*START* Job OVER Req #176 for HELENIUS    Date 15-Feb-80 18:03:08 Monitor: 1031 Great Pumpkin, TOPS-20 Monitor 4(32 *START*
File PS:<HELENIUS>OVER.MEM.1, created: 15-Feb-80 16:36:25, printed: 15-Feb-80 18:12:13
Job parameters: Request created:15-Feb-80 17:00:00   Page limit:675   Forms:NORMAL  Account:1
File parameters: Copy: 17 of 30   Spacing:SINGLE   File format:ASCII   Print mode:ASCII
```





START Job OVER Req #176 for HELENIUS Date 15-Feb-80 18:03:08 Monitor: 1031 Great Pumpkin, TOPS-20 Monitor 4(32 *ST
File PS:<HELENIUS>OVER,MEM.1, created: 15-Feb-80 16:36:25, printed: 15-Feb-80 18:12:18
Job parameters: Request created:15-Feb-80 17:00:00 Page limit:675 Forms:NORMAL Account:1
File parameters: Copy: 17 of 30 Spacing:SINGLE File format:ASCII Print mode:ASCII

1.0 SCOPE

The intent of this document is to provide an overview of the VENUS processor and memory system. Also to identify the various components of the VENUS processor and describe briefly what their role is in the pipeline structure of instruction execution. For further detail of any subsystem, the appropriate hardware subsystem specification should be referenced.

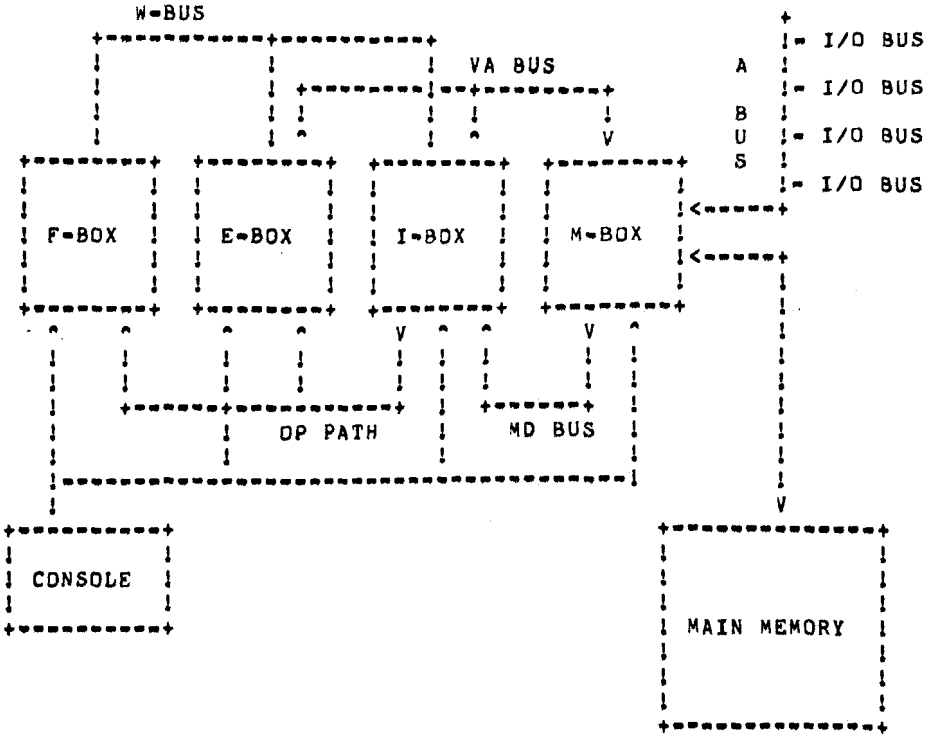
2.0 CONTENTS

The VENUS CPU is comprised of four major functional units. These units are independent specialized processors for performing some portion of the instruction execution. The major units are:

1. CACHE/TRANSLATION BUFFER (M-box)
2. INSTRUCTION UNIT (I-box)
3. EXECUTION UNIT (E-box)
4. FLOATING POINT ACCELERATOR (F-box)

Together, these units form a pipeline for the execution of macro instructions.

The following diagram indicates the major functional units and the inter connection paths.



VENUS BLOCK DIAGRAM

2.1 MAJOR BUSSES

There are four major busses for data and address flow within the VENUS processor. During pipeline operation, all of these busses may be active with some part of instruction execution.

1. W-bus: this bus is primarily for getting write data to all of the copies of the GPR files. Whenever a box writes it's own copy of the GPR's, it also writes all other copies. This bus also carries write data headed to the CACHE via the MD bus.

2. MD bus: This bus it the data connection between the CACHE and CPU for all read and write data.

3. VA bus: This bus has the virtual address from either the E-box or the I-box.

4. OP path: This is a unidirectional set of wires which deliver operands from the I-box to the execution units. (E-box and F-box)

5. A-bus: This is a high speed DMA path from I/O bus adaptors to the CACHE and main memory.

6. IIL bus: This is a serial data bus that connects chains of MCA diagnostic logic to the console processor.

2.2 M-BOX

The M-box is made up of four sections which constitute the memory and I/O connection for the main processor.

1. The primary section of the M-box is the data cache. This section will accept physical addresses from the address translation unit or A-bus and search the cache for data which may be in fast memory storage.

   The data cache is organized as two way set associative. The block size is sixteen bytes and the storage algorithm is write back. The total storage capacity is 8k bytes.

2. This high speed data cache is connected to main memory via a dedicated storage bus. This provides access for up to 8 megabytes of storage in the main processor cabinet and up to 24 additional megabytes of storage in an optional memory expansion cabinet.

3. As part of the data retrieval, the M-box will also perform the translation of processor virtual addresses into physical addresses. This is done in a section called the ADDRESS TRANSLATION BUFFER. This is a small one way associative cache which contains physical address information for up to 1024 pages of virtual address space.

   If there is a match of the contents of the translation buffer tag and the virtual address, the stored physical address is sent directly to the data cache. If the translation buffer lookup fails, the processor micro code will perform the necessary memory references to create the physical address and store that information into the translation buffer.

   The M-box design is organized such that address translation and cache read operation will occur in a 66 2/3 nanosecond period.

4. The last section of the M-box is an I/O path which will allow data transactions with an SBI interface, BI interface and an ICCS adaptors. This will allow a variety of system configurations for either SBI based, BI based peripherals or HSC and MERCURY subsystems.

2.3 I-BOX

The I-box is the central control station for the VENUS processor. It performs a variety of functions related with instruction stream data. Among these is the fetch and decode of macro instructions. The I-box will fetch up to eight bytes of instruction stream data into a high speed buffer. The primary purpose of this buffer is to collect together the instruction bytes from the memory system. This will eliminate most of the performance penalty imposed when instructions cross longword memory boundaries and require more than one memory access. This second reference will be overlapped by the previous instruction decode.

After the instruction stream data is decoded, the I-box will calculate the virtual address (if any) and fetch the operands from the cache. The decode and fetch operations are overlapped such that a new operand may be fetched from the memory every micro cycle.

The I-box is also used during character and decimal string instruction execution. The use here is to fetch the source string operand. This will eliminate the overhead of alining the source string and collecting longwords of data from the cache. This operation fits naturally within the I-box in that the fetch of instruction stream data has the same kinds of problems in dealing with byte addressing within a longword oriented memory system.

These operations allow the execution units to be tailored to the task of logical and arithmetic operations. They simply recive operands from the I-box, operate on the data and return the results to the I-box for storage.

2.4 EXECUTION UNIT

The execution unit serves the function of performing all instruction executions. The unit is basically a simple 32 bit data path with large (240 location) temporary storage, multibit shift capability and a binary/BCD ALU.

The E-box receives operands from the I-box along with instruction decode information. The execution and storage of results is either completed by the E-box for the case of register destination, or sent to the I-box for memory destination. The execution steps are the final stage of the pipeline operation and are overlapped with operand fetches for the next instruction in the I-box in the case of register destination.

The E-box contains the majority of the CPU microcode functions. The functionality of this code is:

1. Instruction execution
2. Machine initialization
3. Interrupt processing
4. Memory management
5. Fault and error processing
6. Console support

2.5 FLOATING POINT ACCELERATOR

The floating point accelerator is a dedicated micro engine for performing very fast execution of a subset of floating point instructions.

This box has a very high speed multiply data path, add/subtract data path and normalization unit. Together these functions reduce the execution time of single, double and grand floating point formats over the corresponding time required by the E-box.

The multiply data path creates and sums partial products of eight times thirty two bits per clock tick (33 1/3 nanoseconds). It is a three stage pipeline which yields single precision results in 266 2/3 nanoseconds for register to register operations.

The add/subtract path includes a 0-63 place shifter to align and sum fractions up to 64 bits wide. The same shifting hardware is used to normalize results and to format them for storage.

Certain F-box operations can be initiated by E-box microcode. This feature is used to enhance the performance of H-format instructions and such instructions as EMOD and POLY.

2.6 CONSOLE SYSTEM

The VENUS Console Subsystem is a microprocessor based single board system serving as a VMS operating system console and as a diagnostic console. This system will be responsible for the the following operations:

1. System initialization and boot
2. Power system and environmental monitoring
3. Micro diagnostic control and dispatch station
4. Operators console interface
5. Console TTY controller
6. Floppy disk controller
7. APT/RD interface
8. Processor clock control
9. Error reporting and logging

There are two major interface paths from the console to the VENUS CPU. One is the IIL visibility bus for initialization and diagnostic monitoring.

The second is a dual port buffer for message communication between the console processor and the E-box micro code. This is used for passing command messages for console operations, MTPR and MFPR operations, and error status information.

```
RRRRRRRRRRRR            AAAAAAAAA         MMM         MMM      PPPPPPPPPPPP
RRRRRRRRRRRR            AAAAAAAAA         MMM         MMM      PPPPPPPPPPPP
RRRRRRRRRRRR            AAAAAAAAA         MMM         MMM      PPPPPPPPPPPP
RRR         RRR      AAA         AAA      MMMMMM   MMMMMM      PPP         PPP
RRR         RRR      AAA         AAA      MMMMMM   MMMMMM      PPP         PPP
RRR         RRR      AAA         AAA      MMMMMM   MMMMMM      PPP         PPP
RRR         RRR      AAA         AAA      MMM   MMM   MMM      PPP         PPP
RRR         RRR      AAA         AAA      MMM   MMM   MMM      PPP         PPP
RRR         RRR      AAA         AAA      MMM   MMM   MMM      PPP         PPP
RRRRRRRRRRRR         AAA         AAA      MMM         MMM      PPPPPPPPPPPP
RRRRRRRRRRRR         AAA         AAA      MMM         MMM      PPPPPPPPPPPP
RRRRRRRRRRRR         AAA         AAA      MMM         MMM      PPPPPPPPPPPP
RRR   RRR            AAAAAAAAAAAAAAA      MMM         MMM      PPP
RRR   RRR            AAAAAAAAAAAAAAA      MMM         MMM      PPP
RRR   RRR            AAAAAAAAAAAAAAA      MMM         MMM      PPP
RRR      RRR         AAA         AAA      MMM         MMM      PPP
RRR      RRR         AAA         AAA      MMM         MMM      PPP
RRR      RRR         AAA         AAA      MMM         MMM      PPP
RRR         RRR      AAA         AAA      MMM         MMM      PPP
RRR         RRR      AAA         AAA      MMM         MMM      PPP
RRR         RRR      AAA         AAA      MMM         MMM      PPP



MMM         MMM      EEEEEEEEEEEEEEE      MMM         MMM                                  111
MMM         MMM      EEEEEEEEEEEEEEE      MMM         MMM                                  111
MMM         MMM      EEEEEEEEEEEEEEE      MMM         MMM                                  111
MMMMMM   MMMMMM      EEE                  MMMMMM   MMMMMM                               111111
MMMMMM   MMMMMM      EEE                  MMMMMM   MMMMMM                               111111
MMMMMM   MMMMMM      EEE                  MMMMMM   MMMMMM                               111111
MMM   MMM   MMM      EEE                  MMM   MMM   MMM                                  111
MMM   MMM   MMM      EEE                  MMM   MMM   MMM                                  111
MMM   MMM   MMM      EEE                  MMM   MMM   MMM                                  111
MMM         MMM      EEEEEEEEEEEE         MMM         MMM                                  111
MMM         MMM      EEEEEEEEEEEE         MMM         MMM                                  111
MMM         MMM      EEEEEEEEEEEE         MMM         MMM                                  111
MMM         MMM      EEE                  MMM         MMM                                  111
MMM         MMM      EEE                  MMM         MMM                                  111
MMM         MMM      EEE                  MMM         MMM                                  111
MMM         MMM      EEE                  MMM         MMM            ......                111
MMM         MMM      EEE                  MMM         MMM            ......                111
MMM         MMM      EEE                  MMM         MMM            ......                111
MMM         MMM      EEEEEEEEEEEEEEE      MMM         MMM            ......             111111111
MMM         MMM      EEEEEEEEEEEEEEE      MMM         MMM            ......             111111111
MMM         MMM      EEEEEEEEEEEEEEE      MMM         MMM            ......             111111111


*START* Job RAMP Req #175 for HELENIUS    Date 15-Feb-80 17:25:55 Monitor: 1031 Great Pumpkin, TOPS-20 Monitor 4(32 *STAR
File PS:<HELENIUS>RAMP.MEM.1, created: 29-Jan-80 17:10:35, printed: 15-Feb-80 17:33:23
Job parameters: Request created:15-Feb-80 17:00:00   Page limit:540   Forms:NORMAL  Account:1
File parameters: Copy: 17 of 30   Spacing:SINGLE   File format:ASCII   Print mode:ASCII
```

[redacted]

digital

interoffice memorandum

Company Confidential

To: VENUS General List

Date: Jan 25, 1980
From: Jud Leonard
Dept: LSEG/VAX
Tel: 231-6839
Loc: MR1-2/E47
File: RAMP-PHILOSOPHY.mem

Subject: VENUS RAMP Philosophy

This is a first attempt at defining a strategy for dealing with errors within VENUS/VMS systems in the field. (Clearly, the error detection and isolation strategies used impact the manufacturing process as well, but that is not the subject of this paper.)

I GOALS

The VENUS RAMP philosophy is driven by the overall highest-priority goal of the VENUS project, Customer Satisfaction better than comparable IBM systems. This translates into pressure to:

- Minimize the number of failures
- Minimize the user work lost as a result of each failure
- Minimize the repair time after a failure

II ASSUMPTIONS

There are a number of assumptions underlying our error-handling philosophy. Among them are:

1. Hard/repeatable faults can be isolated quickly and reliably to a small set of Field Replaceable Units (FRU's) by traditional diagnostics, given control of and visibility to the outputs of those components.

2. Traditional diagnostics, written either in macrocode or microcode, are the appropriate means for coping with hard errors.

3. Most component failures will manifest themselves as intermittent faults for some time before becomming repeatable.

4. Non-repeatable faults, if detected and reported while the information needed for retry is available, can be recovered by retrying the operation in which the fault was detected. (Viewed from a different perspective, this is the assumption that any failure which cannot be recovered by retry can be provoked

repeatably and isolated by diagnostics.)

5. Most intermittent failures are attributable to a single node whose behaviour is out of spec. This is the non-correlated failure model, and it fundamentally attributes faults to local conditions at the failing electrical node rather than to more global conditions which would affect many signals together. This assumption is fundamental to all the RAMP thinking I know about, even though there is (to my knowledge) astonishingly little empirical evidence to support it.

6. Very few failures are the result of faulty design. If this assumption were wrong, we would detect failures and "isolate" them to components which were working within spec; replacing those components clearly would not solve the problem. Since we do not have a satisfactory explanation for intermittent faults, we have to acknowledge the possibility that they result from faulty design, and that the optimal strategy for dealing with intermittent errors might be to retry until the operation succeeds, and then forget the error occurred.

III CONSTRAINTS

1. The basic component failure rates are given, and not meaningful variable under our control. This means that we have relatively little means (beyond worst-case design rules, which I take as given) of affecting the number of instances in which an electrical node is out of spec, and that our efforts must be spent on minimizing the work lost due to a failure.

2. System cost and performance goals hold. That is to say, we intend to meet many goals in addition to the customer satisfaction goal. This rules out Triple Modular Redundancy (TMR) schemes as a general approach.

3. The operating system is VMS. While we are allowing ourselves to specify changes in certain areas of VMS (boot, machine check recovery), we have disallowed strategies which would imply structural changes in VMS.

4. We must use standard DEC interconnects and the peripheral options which interface to them. In cases of current developments on those busses and options, we should try to impose RAMP-consciousness, but we cannot preclude support of existing or future UNIBUS, MASSBUS, S8I, DRI, ICCS, or BI devices.

IV STRATEGY

On the basis of these assumptions, we are taking a two-pronged approach to dealing with hardware failures. During normal system operation, we will assume that any detected hardware failure is intermittent, and will take steps to retry the unit of activity in which the error is believed to have occurred. (This unit of activity will be different depending on the type of error; eg, a single clock cycle in some cases, or an entire disk transfer in others.) In the event the retry fails repeatedly, or that the system is unable to operate normally, we will assume that there is

a hard fault, and depend on diagnostics to identify a faulty FRU.

Furthermore, because we assume that solid faults first show themselves as intermittents, we will attempt to record a description of the hardware state at the time of each detected fault, hoping that analysis of this state will allow field service to identify and replace an intermittent FRU before it fails solidly. This is an area of considerable uncertainty and difficult tradeoffs, because the success of this strategy depends on isolating the fault sufficiently accurately and precisely to allow the faulty component to be removed and destroyed. Attempts to return an intermittently failing component to a depot or manufacturing for repair will cause more trouble than they save, for two reasons:

1. Since those facilities have only diagnostics with which to isolate the failure, they cannot be expected to find intermittents.

2. Depending on the failure mode, many faults will disappear when the failing component is moved to a different environment. This problem already occurs, but will be exacerbated if intermittently failing components are recycled.

V RECOVERY MECHANISMS

Different kinds of errors are recovered in various ways, depending on historical precedent, cost, and the availability of information needed for retry. The major categories are listed below:

1. I/O device errors -- In general, these are recovered in a device-specific way by the device driver, if recovery is possible. In the specific cases of Heirarchical Storage Controller (HSC) and Mercury, the outboard controller will attempt both recovery and diagnosis, sending error log information to the host for logging by either the port driver or the class driver.

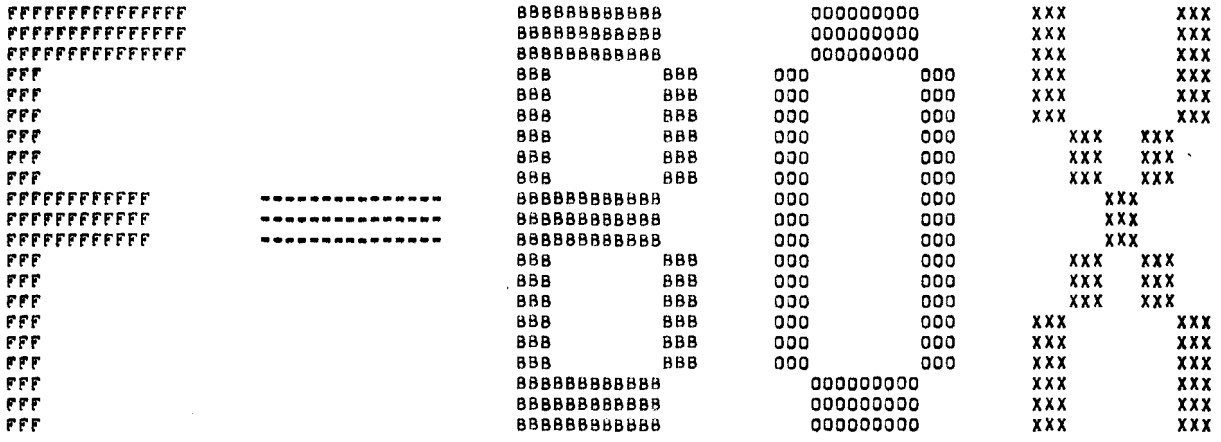
2. Main memory single bit errors -- MBox hardware will correct these on the fly, keeping track of the physical address and the syndrome of the error. The console will periodically check for such errors through the IIL visibility path, and will report its findings by interrupting the CPU to transfer an error log message in a manner analogous to CTY input.

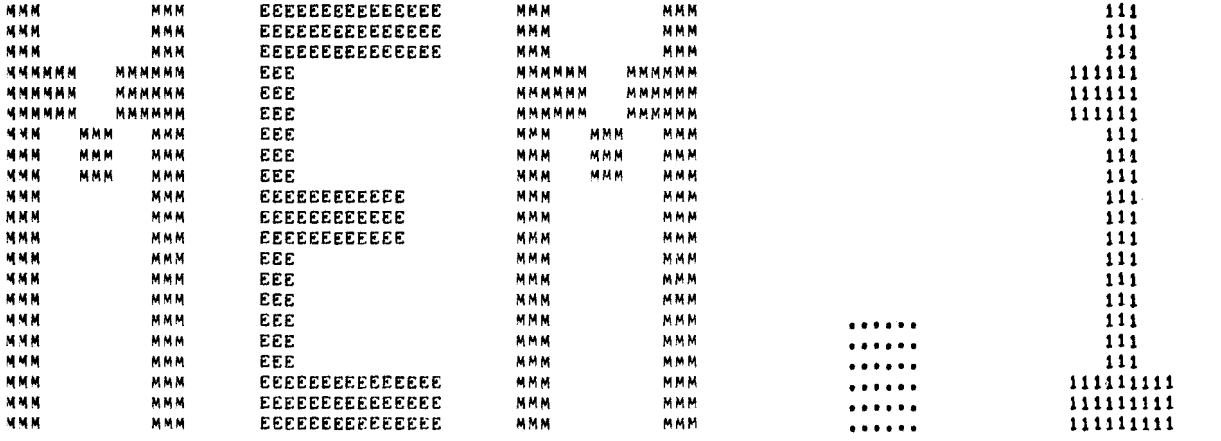
3. Control store parity errors -- Ebox, Ibox, FPA execution will stop while the console reads the faulty microinstruction, corrects it using an ECC code stored in console memory, and rewrites it. The console will then restart the processor (the Mbox has been running throughout to maintain I/O traffic), and interrupt the CPU to transfer an error log message.

4. Data Path and Scratch Pad parity errors -- A microtrap will prevent completion of the cycle in which the failure is detected, and microcode will attempt to record error log information in reserved scratch pad locations. Microcode will

return to the cycle which failed, and attempt retry. If it succeeds, the error log information will be stored on a machine check trap at the end of the instruction. If it fails (another error occurs in the same macroinstruction), the macroinstruction will be aborted and the error log information stored.

There will be a flag in the error logout which indicates whether the macroinstruction which failed had written memory or GPR's before the failure. If no such writes occurred, the operating system machine check software will restart the instruction after logging the error. If any writes had occurred, or if the restart fails, then the machine check exception must be reported up to any handlers in the current process. In the normal event that there are none, this will result in aborting the process.





START Job F-BOX Req #174 for HELENIUS Date 15-Feb-80 17:10:34 Monitor: 1031 Great Pumpkin, TOPS-20 Monitor 4(32 *START*
File PS:<HELENIUS>F-BOX.MEM.1, created: 15-Feb-80 15:50:40, printed: 15-Feb-80 17:18:21
Job parameters: Request created:15-Feb-80 17:00:00 Page limit:540 Forms:NORMAL Account:1
File parameters: Copy: 16 of 30 Spacing:SINGLE File format:ASCII Print mode:ASCII

CHAPTER 1

FBOX HARDWARE ORGANIZATION

1.1 HARDWARE PHILOSOPHY

FBOX operates in close co-operation with EBOX in performing floating-point instructions. As in all ordinary instructions, the IBOX performs the operand fetches and presents the operands via IBOX OP bus. The operands are delivered to both the EBOX and the FBOX.

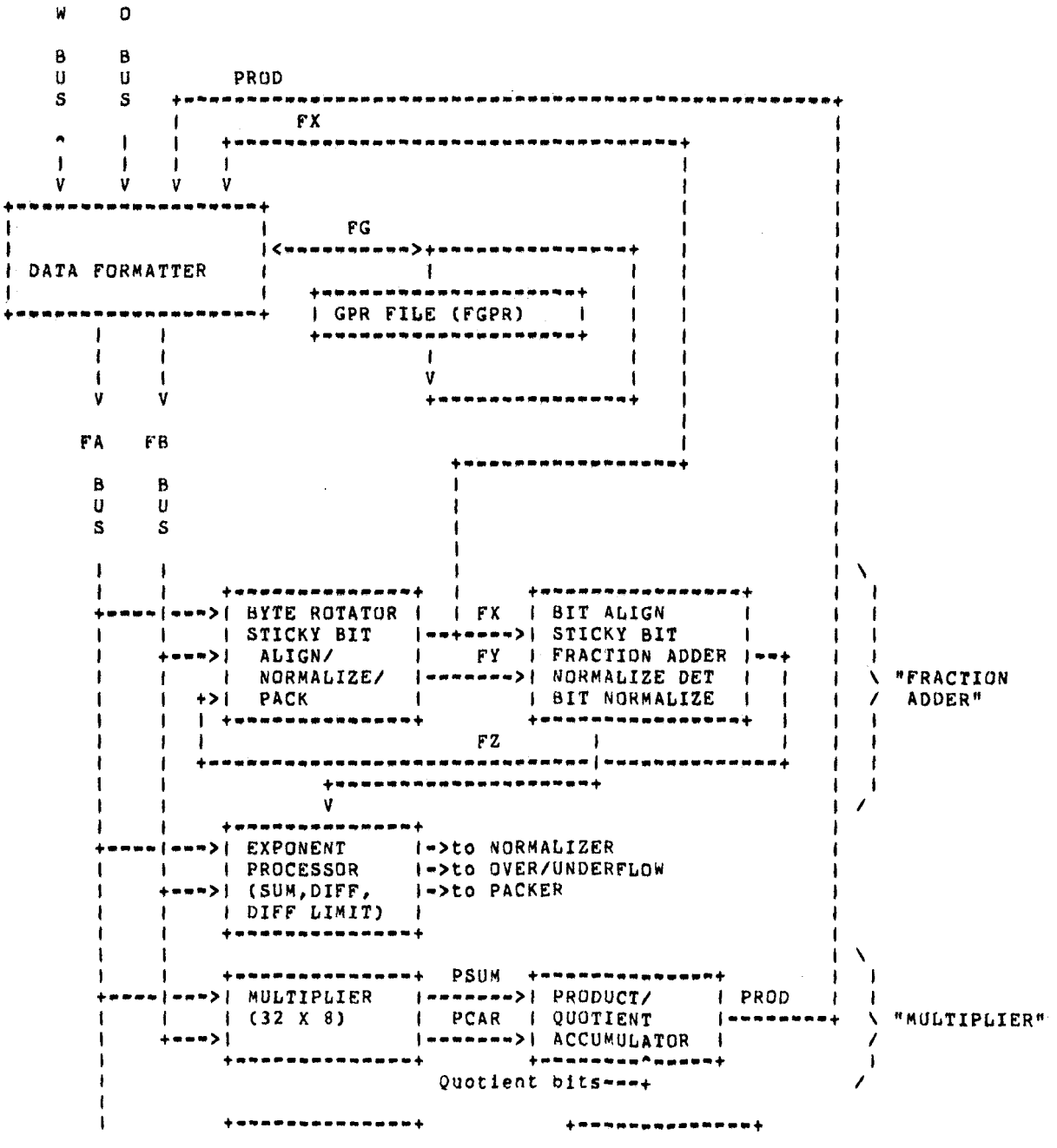
FBOX executes an IDLE state while awaiting an FBOX operation. At IRD, EBOX and FBOX both dispatch to OPCODE-dependent flows. FBOX operations dispatch to states which acquire operands in conjunction with the EBOX, then to execution states.

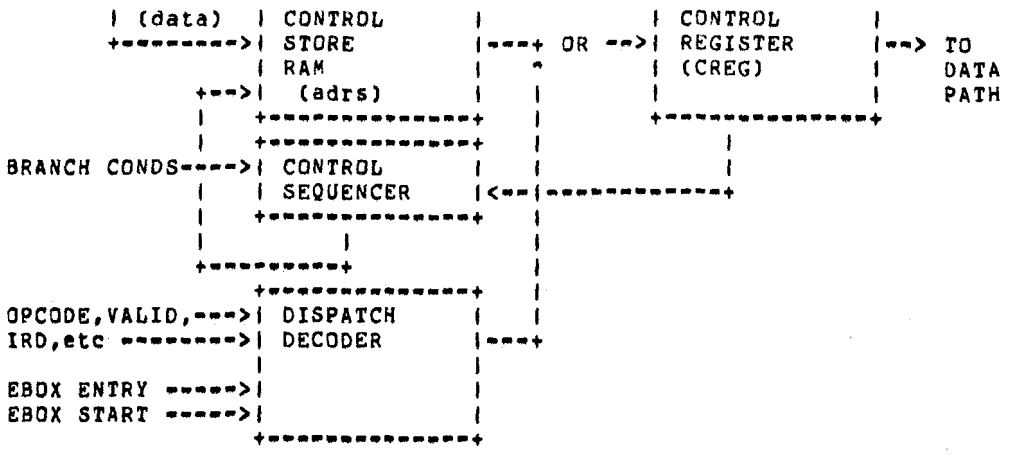
EBOX dispatches to operand-acquisition flows as though the FBOX were not present. These flows keep the IBOX in sync with E and FBOX. After acquiring the operands, EBOX notices that FBOX is present and proceeds to a destination fork, where it awaits FBOX SYNC. This signal indicates that the FBOX will place data on the WBUS in the next cycle (assuming successful arbitration). EBOX asserts the appropriate WBUS control signals to perform the correct destination transfer. FBOX provides signals to EBOX condition code logic to load the correct CC's and returns to an idle state to await the next FBOX instruction.

1.2 BLOCK DIAGRAM

The following page depicts the major hardware elements of the FBOX implementation. Subsequent pages describe the operation of the elements in increasing detail.

FBOX BLOCK DIAGRAM





1.2.1 DATA FORMATTER

The data formatter provides the interface between "the world" of GPRs and Memory operands, where data live in very peculiar formats (both disordered and incomplete), to the world of usable floating-point data. This transformation is accomplished by shifting, rotating, and masking (for multiply only) operations done primarily in the formatter section. The formatter also manages the local copy of the GPR's which are used to speed operand delivery.

1.2.2 FRACTION ADDER

The fraction adder is implemented in two major sections. The BYTE ROTATOR accomplishes alignment, normalization, and packing to the byte boundary. The BYTE ROTATOR outputs FX and FY.

The adder proper shifts the FX input 0-7 places, complements it if required, and accomplishes the 64-bit ADD operation. The adder output is tested for all-0's, and for normalization count. Normalize count is generated by the use of a network of priority encoders.

The adder output, FZ, is routed via the byte rotator, where bytewise normalization is accomplished, to FX. FX is bit-shifted to complete the normalization and is presented to the adder with the appropriate round bit. The rounded sum on FZ is routed via the byte rotator, where storage formatting is accomplished, to the DATA FORMATTER, then to the W-BUS for storage.

1.2.3 EXPONENT PROCESSOR

The exponent processor acquires the exponents of the operands and calculates their difference (ADDx, SUBx, or DIVx) or sum (MULx). The sum/difference is limit tested (Overflow, Underflow, or Shift Out Of Range) , corrected as required,and presented to the appropriate logic for further use.

If the operation results in an unnormalized fraction, the normalization count is presented to the exponent processor for further exponent arithmetic. The results of this normalization arithmetic are tested for underflow/overflow and are presented to the FZ bus for inclusion in the floating point result.

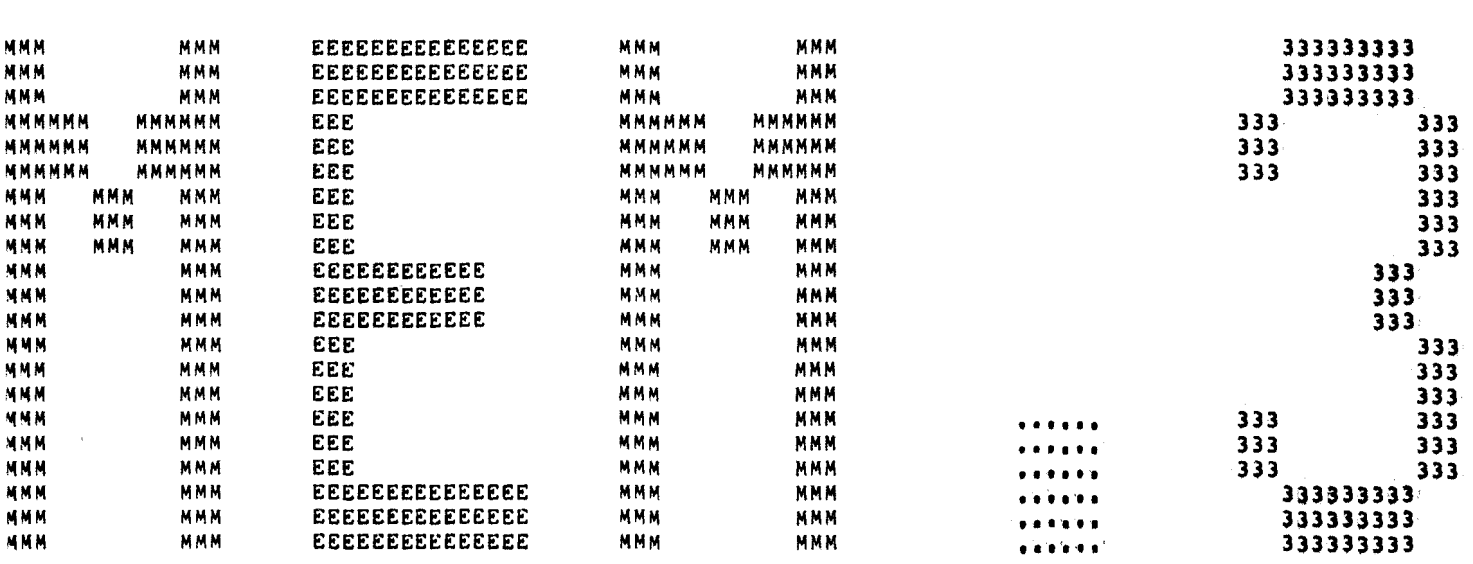
1.2.4 MULTIPLIER

The multiplier forms 40-bit (32x8) products and accumulates them to form the products necessary for multiply-class instructions (MUL, EMOD, POLY) The product may not be normalized and is not rounded. It is delivered to the fraction adder for normalization and rounding via the PROD bus. The multiplier also accumulates the quotient during divide operations.

1.2.5 CONTROL STORE RAM

The control store RAM contains the control information used to execute FBOX operations. It is loaded via the WBUS and FA bus. It accessed under the control of the FBOX control sequencer. The control store is parity checked. Parity errors will result in either an alternate data retry , if two copies of the microcode can be supported, or in

a machine check which can cause a reload of the microcode and a software retry.



```
*START* Job IREG Req #188 for KNIGHT    Date 15-Feb-80 17:48:04 Monitor: 1031 Great Pumpkin, TOPS-20 Monitor 4(32 *STAR
File VENUS:<IBOX,SPEC>IREG,MEM,3, created: 15-Feb-80 17:09:27, printed: 15-Feb-80 18:01:38
Job parameters: Request created:15-Feb-80 17:09:47   Page limit:540   Forms:NORMAL  Account:1
File parameters: Copy: 28 of 30   Spacing:SINGLE   File format:ASCII   Print mode:ASCII
```

1.0 GENERAL DESCRIPTION

The Ibox consists of the functional blocks needed to do instruction pre-fetching, effective address calculation, operand fetching and instruction optimization for simple instructions.

2.0 COMPONENTS OF THE IBOX

The main components of the Ibox are:

1. Instruction Buffer (IBUF)
   Eight byte buffer that holds the I-stream bytes that are to be processed. It is filled by I-stream prefetch cache cycles. Also used to fetch strings

2. Instruction Dispatch RAMs (VAX Mode & Compatibility Mode)
   Generates Ibox Control Information and Execute addresses for Ebox

3. Three input adder
   Used to calculate branch target and D-stream operand addresses

4. General Purpose Registers (GPRs)
   These are a copy of the Ebox GPRs for use by the Ibox

5. Instruction Optimization Logic
   Decodes addressing mode conditions which may be executed in less than the normal number of Ebox machine cycles. (GPR destinations)

6. Ibox Micromachine
   The Ibox Micromachine is used to control the Ibox and to perform complex operations in the Ibox such as GPR restoration after a trap or fault, unaligned cache references, and indexed address calculations.

7. Register Log
   This is a RAM file which records the changes that the Ibox makes to GPRs such that those changes may be "unwound" if it is necessary to retry the instruction.

8. Bus Interfaces to MD, OP, W, VA, and Diagnostic (IIL) buses

   The MD bus is the Memory Data bus and goes from the Ibox to the Mbox. Read and write data is transferred over this bus.

The OP bus is the Operand bus. It is used to transfer operands from the Ibox to the Ebox and Fbox.

The W bus is the Write bus which goes between the Ebox, Fbox and Ibox. It is used to write GPRs and memory. It is also used to load Ibox DRAMs and microcode.

The VA bus is the Virtual Address bus. It is used to transfer virtual addresses to the Mbox from either the Ebox or the Ibox.

The IIL Diagnostic bus is used to examine all points in the machine that must be visible to any diagnostic. It is also used to examine the contents of the Ibox control store and DRAMs when this is necessary. Generally the control store and DRAMs will only be written. However, they will be visible to IIL.

2.1 Ibox Registers

The Ibox data path handles data for all cache operations and also for address calculations and operand data. It consists of a three input adder, VIBA, VA, IMD, EMD, ID, IWR, CUR PC, ISA, DSA, ESA, ICTL, IERR registers and a copy of the GPRs. Results from the adder are stored in either the VA or IWR registers. The data may represent addresses, operands, or index values. New values for auto increment and auto decrement operations are temporarily stored in the IWR register and are written into the appropriate GPR on the next available W bus cycle as a pipelined operation.

The VIBA register is the Virtual Instruction Buffer Address register. It holds the address of the next byte of data that will be fetched from memory to fill the IBUF for either I-stream or string D-stream fetches. The VIBA register is loaded from either the VIBA ADDER for incrementing VIBA, or the 3-INPUT ADDER for loading the I-stream start address via the W bus.

The VA register is the Virtual Address register. It holds the address of the next D-stream fetch, or the initial I-stream start address. It is also used as a temporary register for Asources, literals, register operands, and indirect pointers. For destination addresses, the VA register holds the address to which the next memory write will occur when the Ibox has the address available. The Ibox micromachine both increments and decrements the VA Register for operands greater than 32 bits in length. The

VA register is normally loaded from the 3-INPUT ADDER except for Register Operands and Autoincrement mode addresses.

The IMD register is the Memory Data register for Ibox generated references. This register is loaded with data received from the cache on Ibox D-stream references. It is byte loadable to allow the Ibox to handle unaligned references.

The EMD register is the Memory Data register for Ebox generated references. This register is loaded with data received from the cache on Ebox D-stream references. It is byte loadable to allow the Ebox to do unaligned references.

The ID register is the Instruction Data register. This register is used to provide a stage of pipelining for operands from GPRs, literals, and Asources. There is no comparable pipeline register for the MD Registers since the cache itself is one of the stages in the pipe for memory operands.

The IWR register is the Ibox W bus register. It holds adder results that will be written into the GPR files via the W bus on the next available W bus cycle. The Ibox has second priority for use of the W bus.

The CUR PC register is the Current PC register. It holds the current value of macro PC which is the value representing the point at which the IBUF is processing operands. It is updated as each specifier is processed and by jump and branch instructions. It points to the opcode of the instruction following a string instruction during the entire time that the IBUF is used as a string fetching facility and therefore need not be updated when the Ibox is restarted after the string instruction is completed.

The ISA register is the Instruction Decode Start Address register. It holds the starting macro PC of the instruction that the Ibox is currently processing operands for. It is updated from CUR PC each time the Ibox starts a new instruction.

The ESA register is the Ebox Start Address register. It holds the starting macro PC of the instruction that the Ebox is currently processing. It is updated each time the Ebox leaves IRD.

The ICTL register is the Ibox Control register. This register holds bits which indicate the state of the Ibox and bits which allow the diagnostic microcode to force hardware errors.

The IERR register is the Ibox Error register. This register holds the exact cause of the error condition. This register latches the first error to occur and all subsequent errors that occur are not latched.

Some errors may be latched only in the Mbox. The Ebox microcode must read the Mbox IBUF Error Register in the Mbox if required.

3.0 IBOX DATA FLOW BLOCK DIAGRAM

The Attached Ibox Data Flow Block Diagram shows the possible data flow paths through the Ibox data path logic. The following is a summary of the functions performed by each piece of the data path.

The INPUT ROTATOR performs byte rotation from the longword-aligned Mbox data to right-justified alignment required by the data path and IBUFFER.

The OUTPUT ROTATOR converts right-justified data to longword-aligned Mbox data.

The IBUFFER is the Instruction buffer referred to above. The IBUFFER is used to hold the current I-stream being processed. It is also used to fetch string operands.

The WRITE LATCH is used to latch data from the W bus for cases involving unaligned writes to the Mbox.

The D MUX selects bytes from the IBUFFER or IMD register to be presented to the 3-INPUT ADDER. The selections are context dependent and sign extended as necessary.

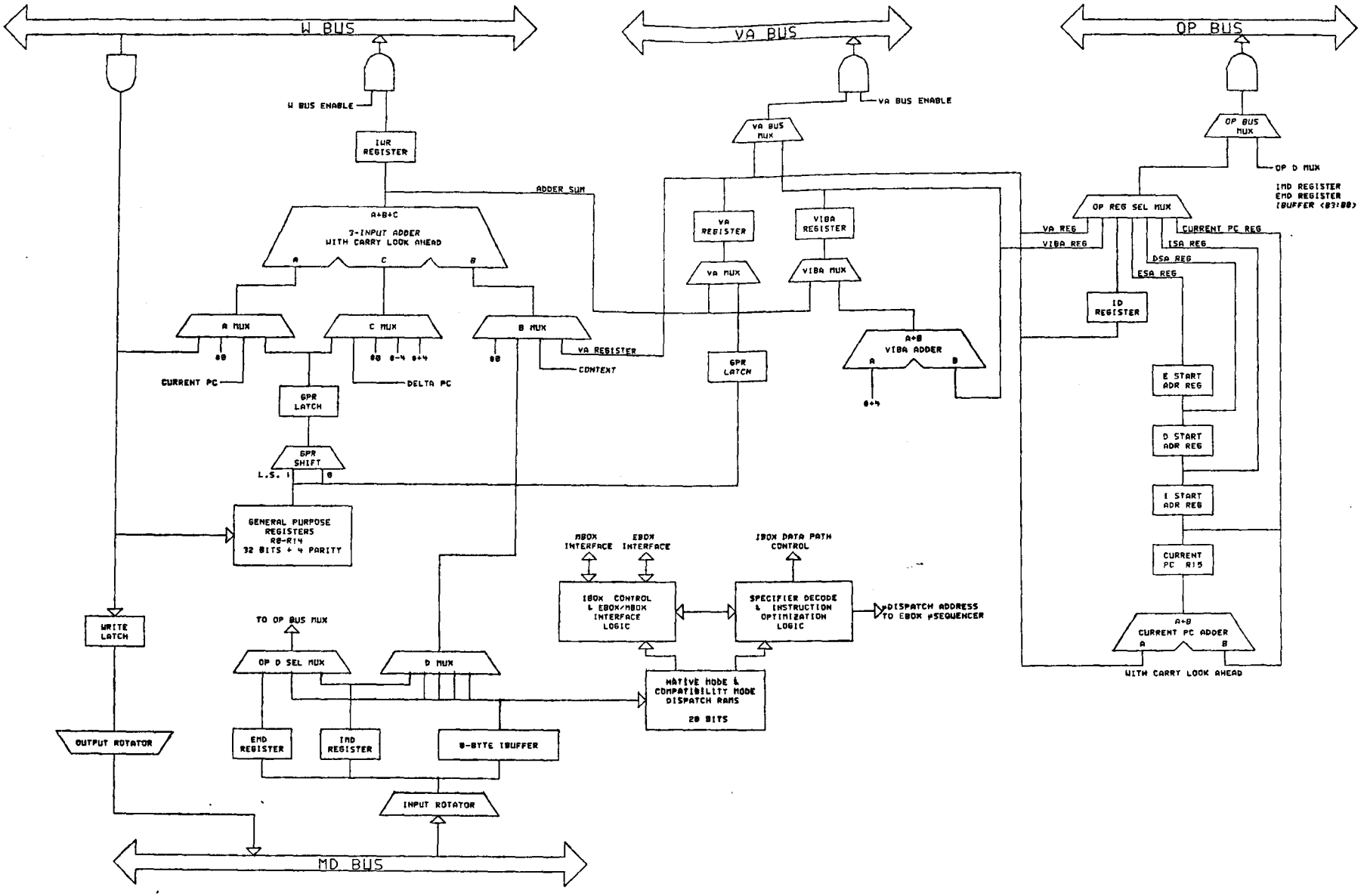
The GPR SHIFT mux is used to produce a multiplication of the GPR data times 2 for some contexts of Indexed Addressing.

The GPR LATCH is used to latch read data from the GPR file during the second half of the machine cycle in which the GPR file may be being written.

The A MUX, B MUX, and C MUX are used to provide the correct inputs to the 3-INPUT ADDER to accomplish the address calculation specified in the instruction stream.

The OP BUS MUX, OP REG SEL MUX, and OP D SEL MUX act together to form the data to be presented to the operand bus. The operand bus contains a data formatter (not shown) for floating data types which is used to present the data to the Ebox in the correct format for these data types.

This group of multiplexors is under control of the Ebox for diagnostic and service routine purposes. It is under control of the Ibox during operand delivery.



REVISIONS		
CHK	CHANGE NO.	REV

digital

DRN. Tommy
CHK'D.

DATE 18-FEB-88 ENG.
DATE BOARD LOCATION:
SHEET 1 OF 1

VENUS:<IBOX.DIAGRAM>IBLK.DRW 15-FEB-88 13:38
NEXT HIGHER ASSEMBLY: VENUS
FIRST USED ON OPTION/MODEL: VENUS

TITLE: VENUS IBOX BLOCK DIAGRAM

SIZE	CODE	NUMBER	REV.
D	CS		

```
M    M   222
MM  MM  2   2
M M M      2
M   M     2
M   M    2
M   M   2
M   M  22222



M    M   III    CCCC           1    4  4    000
MM  MM    I    C             11    4  4   0   0
M M M     I    C              1    4  4   0  00
M   M     I    C              1   44444  0 0 0
M   M     I    C              1       4  00  0
M   M     I    C        ..    1       4  0   0
M   M    III    CCCC    ..   111      4   000
```

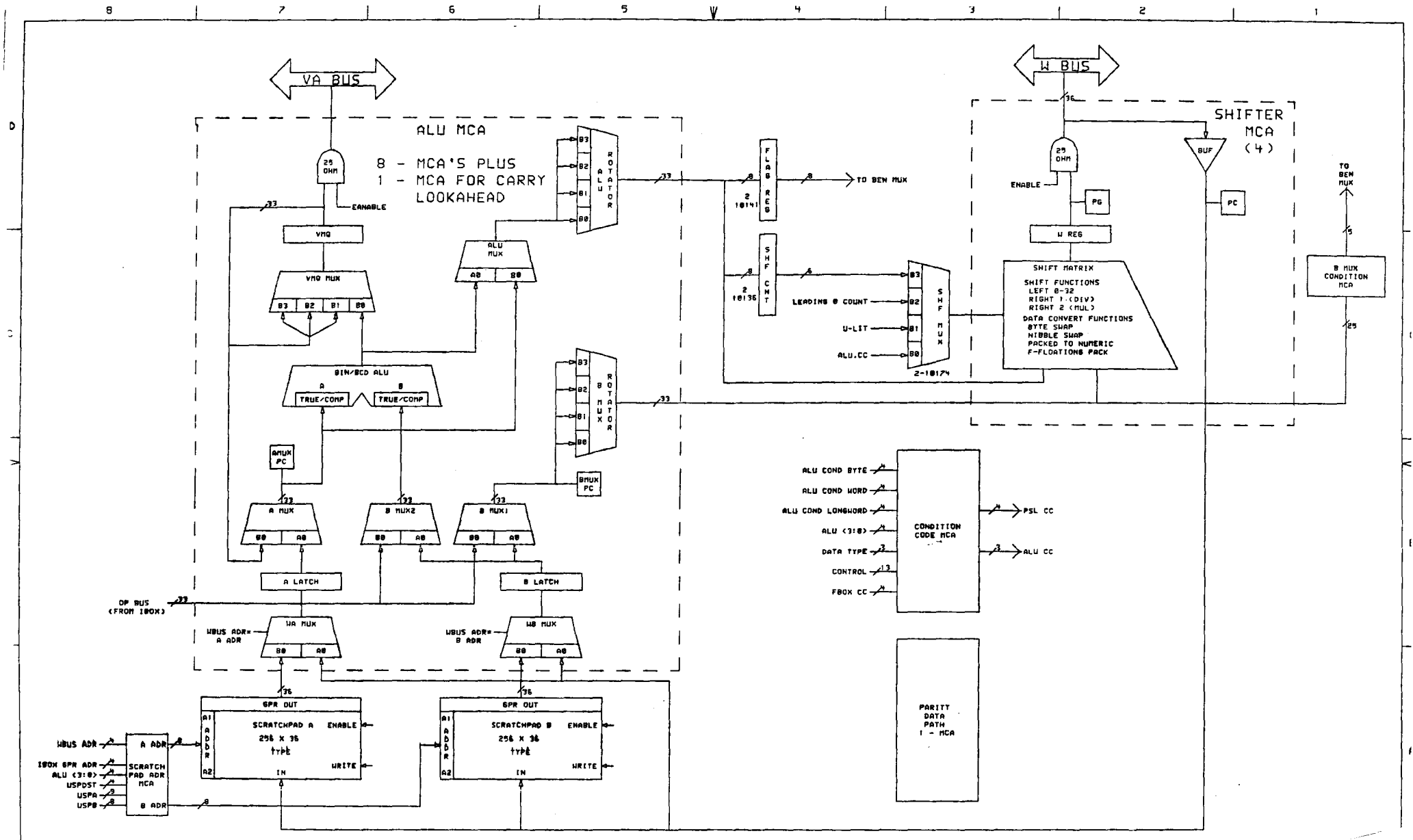
START Job M2 Req #147 for HELENIUS Date 15-Feb-80 14:57:21 Monitor: 1031 Great Pumpkin, TOPS-20 Monitor 4(32 *START*
File PS:<HELENIUS>M2.MIC.140, created: 12-Feb-80 15:29:35, printed: 15-Feb-80 15:02:39
Job parameters: Request created:15-Feb-80 14:57:20 Page limit:270 Forms:NORMAL Account:1
File parameters: Copy: 17 of 30 Spacing:SINGLE File format:ASCII Print mode:ASCII

.HEXADECIMAL
.RTDI

```
.HEXADECIMAL
.RTOL
.NOBIN
.UCODE
J/=<12:0>

.CCODE
; Dram format
;
; 19  17 16 15 14  13 12 11  10      9       8     7   6    0
; +-----+-----+-----+-----+-----+-------+-------+-----+------+
; ! CTX ! REF ! TYPE ! CTL ! SUS ! BDEST ! LAST ! PAR ! ADRS !
; +-----+-----+-----+-----+-----+-------+-------+-----+------+
;
; This is a dummy address field to keep the micro assembler happy
;
DJ/=<19>,,ADDRESS,,VALIDITY=0


; This is a dummy field which allows both context and operand type
; to be expressed in the the same word.
;
OP_TYPE/=<19>,,VALIDITY=0

        BYTE=0
        WORD=4
        LONG=8
        QUAD=0C
        FLOAT=9
        DOUBLE=0D
        GRAND=0E
        HUGE=12

; This is a real field which defines the number of bytes of an operand.
;
CTX/=<19:17>

        BYTE=0
        WORD=1
        LONG=2
        QUAD=3
        HUGE=4
.PAGE
; This field defines the kind of memory operation to be performed or
; address source.
;
REF/=<16:15>

        ASRC=0
        READ=1
        WRITE=2
        MODIFY=3
        VSRC=4

; This field defines the operand type as field address, integer or floating
; point datum. This together with the context field describes the operand type.
;
TYPE/=<14:13>

        INT=0
```

```
        FLOAT=1
        G_FLOAT=2
        VSRC=3

; This field control the execution address generation for the e-box.
; It also indicates to the i-box decode hardware how to operate on the
; bytes within the byte buffer.
;
CTL/=<12:11>

                        ; R-mode finder = 0       R-mode finder = 1
        EXECUTE=0       ; dram adrs             /       dram adrs
        SINGLE=1        ; dram adrs             /       (dram adrs),or.1
        OPT_TWO=2       ; adrs=ctx field        /       (dram adrs),or.1
        OPT_TWO&EXC=3   ; dram adrs             /       (dram adrs),or.1

; This bit will suspend the i-box operation until further notice from
; the e-box.
;
SUSPEND/=<10>

        OFF=0
        ON=1

; This bit indicates to the i-box decode hardware that the next operand
; to be evaluated is a branch displacement
;
B_DEST/=<9>

        FALSE=0
        TRUE=1

; This bit indicates to the i-box hardware that this is the last operation
; for this instruction. This means that the opcode byte will be discarded
; along with any other bytes which may be removed.
;
LAST/=<8>

        OFF=0
        ON=1

; This is the parity of the entry
;
PAR/=<7>,,DEFAULT=<,PARITY[<CTX/>,<TYPE/>,<REF/>,<CTL/>,<SUSPEND/>,<B_DEST/>,<LAST/>,<ADRS/>]>

; This field will get the low order seven bits of the jump address of the e-box
;
ADRS/=<6:0>

.page
; Macros to support entry of DRAM contents
;
; Arguments are allowed as follows:
;
; EXC ACCESS[     ]  TYPE[      ]  ADRS[        ]
;             READ          BYTE            LABEL
;             MODIFY        WORD
;             WRITE         LONG
;             ASRC          QUAD
;             VSRC          FLOAT
```

```
;                       DOUBLE
;                       GRAND
;                       HUGE
;
; Macros for dram entries
;
; This macro is used inside the following macros  only.
;
REF[] AD[]                      "REF/<.AND[<REF/@1>,3]>,ADRS/<.AND[<J/@2>,7F]>"


; These macros describe the dram actions
;
EXC ADRS[]                      "ADRS/<.AND[<J/@1>,07F]>,CTL/EXECUTE"

EXC ACCESS[] TYPE[] ADRS[]      "TYPE/<.CASE[<.EQL[<REF/@1>,4]>]DF[<.AND[<OP_TYPE/@2>,3]>,<TYPE/VSRC>]>,REF[@1] AD[@3]
X/<.SHIFT[<OP_TYPE/@2>,-2]>"

OPT2 ACCESS[] TYPE[] ADRS[]     "TYPE/<.CASE[<.EQL[<REF/@1>,4]>]DF[<.AND[<OP_TYPE/@2>,3]>,<TYPE/VSRC>]>,REF[@1] AD[@3]
OP_TYPE/@2>,-2]>,CTL/OPT_TWO"

OPT2&EXC ACCESS[] TYPE[] ADRS[] "TYPE/<.CASE[<.EQL[<REF/@1>,4]>]DF[<.AND[<OP_TYPE/@2>,3]>,<TYPE/VSRC>]>,REF[@1] AD[@3]
XC,CTX/<.SHIFT[<OP_TYPE/@2>,-2]>"

SUSPEND                         "SUSPEND/ON"
LAST_ENTRY                      "LAST/ON"
BDEST_NEXT                      "B_DEST/TRUE"

.BIN
.PAGE
; DRAM CONTENTS:
;
000:
HALT:   EXC ADRS[0E], LAST_ENTRY
        EXC ADRS[0]
        EXC ADRS[0]
        EXC ADRS[0]
        EXC ADRS[0]
        EXC ADRS[0]
        EXC ADRS[0]
        EXC ADRS[0]
        EXC ADRS[0]

NOP:    EXC ADRS[0E], LAST_ENTRY
        EXC ADRS[0]
        EXC ADRS[0]
        EXC ADRS[0]
        EXC ADRS[0]
        EXC ADRS[0]
        EXC ADRS[0]
        EXC ADRS[0]
```



	DRN.	DATE 15-FEB-80	ENG.	DATE	TITLE: EBOX BLOCK DI
	CHK'D.	DATE	BOARD LOCATION: SHEET OF		
PS:<BISSON>EBOX80.DRW 15-FEB-80 11:57			NEXT HIGHER ASSEMBLY: VENUS		SIZE D / CODE CS / NUMBER MXXXX-0
FIRST USED ON OPTION/MODEL:					

```
M    M   222
MM  MM  2   2
M M M       2
M    M     2
M    M    2
M    M   2
M    M  22222
```

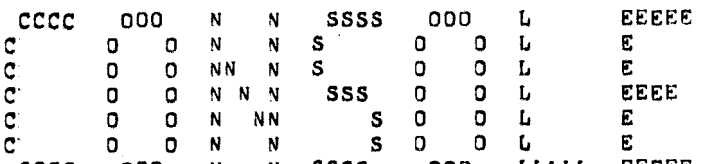
```
M    M   III     CCCC            1     4  4     000
MM  MM    I     C               11     4  4    0   0
M M M     I     C                1     4  4    0  00
M    M    I     C                1     44444  0 0 0
M    M    I     C                1        4   00  0
M    M    I     C          ..    1        4   0   0
M    M   III     CCCC      ..   111       4    000
```

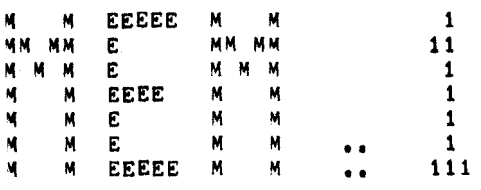
START Job M2 Req #147 for HELENIUS Date 15-Feb-80 14:57:21 Monitor: 1031 Great Pumpkin, TOPS-20 Monitor 4(32 *STA
File PS:<HELENIUS>M2.MIC.140, created: 12-Feb-80 15:29:35, printed: 15-Feb-80 15:02:39
Job parameters: Request created:15-Feb-80 14:57:20 Page limit:270 Forms:NORMAL Account:1
File parameters: Copy: 17 of 30 Spacing:SINGLE File format:ASCII Print mode:ASCII

```
.HEXADECIMAL
.RTOL
.NOBIN
.UCODE
J/=<12:0>

.CCODE
; Dram format
;
; 19  17 16 15 14  13 12 11  10      9       8     7   6    0
; +-----+-----+-----+-----+-----+-------+-------+-----+------+
; ! CTX ! REF ! TYPE ! CTL ! SUS ! BDEST ! LAST ! PAR ! ADRS !
; +-----+-----+-----+-----+-----+-------+-------+-----+------+
;
; This is a dummy address field to keep the micro assembler happy
;
DJ/=<19>,.ADDRESS,.VALIDITY=0


; This is a dummy field which allows both context and operand type
; to be expressed in the the same word.
;
OP_TYPE/=<19>,.VALIDITY=0

        BYTE=0
        WORD=4
        LONG=8
        QUAD=0C
        FLOAT=9
        DOUBLE=0D
        GRAND=0E
        HUGE=12

; This is a real field which defines the number of bytes of an operand.
;
CTX/=<19:17>

        BYTE=0
        WORD=1
        LONG=2
        QUAD=3
        HUGE=4
.PAGE
; This field defines the kind of memory operation to be performed or
; address source.
;
REF/=<16:15>

        ASRC=0
        READ=1
        WRITE=2
        MODIFY=3
        VSRC=4

; This field defines the operand type as field address, integer or floating
; point datum. This together with the context field describes the operand type.
;
TYPE/=<14:13>

        INT=0
```

```
        FLOAT=1
        G_FLOAT=2
        VSRC=3

; This field control the execution address generation for the e-box.
; It also indicates to the i-box decode hardware how to operate on the
; bytes within the byte buffer.
;
CTL/=<12:11>

                        ; R-mode finder = 0       R-mode finder = 1
        EXECUTE=0       ; dram adrs             /       dram adrs
        SINGLE=1        ; dram adrs             /       (dram adrs),or.1
        OPT_TWO=2       ; adrs=ctx field        /       (dram adrs),or.1
        OPT_TWO&EXC=3   ; dram adrs             /       (dram adrs),or.1

; This bit will suspend the i-box operation until further notice from
; the e-box.
;
SUSPEND/=<10>

        OFF=0
        ON=1

; This bit indicates to the i-box decode hardware that the next operand
; to be evaluated is a branch displacement
;
B_DEST/=<9>

        FALSE=0
        TRUE=1

; This bit indicates to the i-box hardware that this is the last operation
; for this instruction. This means that the opcode byte will be discarded
; along with any other bytes which may be removed.
;
LAST/=<8>

        OFF=0
        ON=1

; This is the parity of the entry
;
PAR/=<7>,,DEFAULT=<,PARITY[<CTX/>,<TYPE/>,<REF/>,<CTL/>,<SUSPEND/>,<B_DEST/>,<LAST/>,<ADRS/>]>

; This field will get the low order seven bits of the jump address of the e-box
;
ADRS/=<6:0>

.page
; Macros to support entry of DRAM contents
;
; Arguments are allowed as follows:
;
; EXC ACCESS[     ]  TYPE[      ]  ADRS[        ]
;             READ         BYTE           LABEL
;             MODIFY       WORD
;             WRITE        LONG
;             ASRC         QUAD
;             VSRC         FLOAT
```

```
;                       DOUBLE
;                       GRAND
;                       HUGE
;
; Macros for dram entries
;
; This macro is used inside the following macros  only.
;
REF[] AD[]                      "REF/<,AND[<REF/@1>,3]>,ADRS/<,AND[<J/@2>,7F]>"


; These macros describe the dram actions
;
EXC ADRS[]                      "ADRS/<,AND[<J/@1>,07F]>,CTL/EXECUTE"

EXC ACCESS[] TYPE[] ADRS[]      "TYPE/<,CASE[<,EQL[<REF/@1>,4]>]OF[<,AND[<OP_TYPE/@2>,3]>,<TYPE/VSRC>]>,REF[@1] AD[@3],C
X/<,SHIFT[<OP_TYPE/@2>,-2]>"

OPT2 ACCESS[] TYPE[] ADRS[]     "TYPE/<,CASE[<,EQL[<REF/@1>,4]>]OF[<,AND[<OP_TYPE/@2>,3]>,<TYPE/VSRC>]>,REF[@1] AD[@3],C
OP_TYPE/@2>,-2]>,CTL/OPT_TWO"

OPT2&EXC ACCESS[] TYPE[] ADRS[] "TYPE/<,CASE[<,EQL[<REF/@1>,4]>]OF[<,AND[<OP_TYPE/@2>,3]>,<TYPE/VSRC>]>,REF[@1] AD[@3],C
XC,CTX/<,SHIFT[<OP_TYPE/@2>,-2]>"

SUSPEND                         "SUSPEND/ON"
LAST_ENTRY                      "LAST/ON"
BDEST_NEXT                      "B_DEST/TRUE"

.BIN
.PAGE
; DRAM CONTENTS:
;
000:
HALT:   EXC ADRS[0E], LAST_ENTRY
        EXC ADRS[0]
        EXC ADRS[0]
        EXC ADRS[0]
        EXC ADRS[0]
        EXC ADRS[0]
        EXC ADRS[0]
        EXC ADRS[0]
        EXC ADRS[0]

NOP:    EXC ADRS[0E], LAST_ENTRY
        EXC ADRS[0]
        EXC ADRS[0]
        EXC ADRS[0]
        EXC ADRS[0]
        EXC ADRS[0]
        EXC ADRS[0]
        EXC ADRS[0]
```





START Job CONSOL Req #177 for HELENIUS Date 15-Feb-80 17:00:11 Monitor: 1031 Great Pumpkin, TOPS-20 Monitor 4(32 *
File PS:<HELENIUS>CONSOLE.MEM.1, created: 15-Feb-80 16:37:15, printed: 15-Feb-80 17:03:10
Job parameters: Request created:15-Feb-80 17:00:00 Page limit:270 Forms:NORMAL Account:1
File parameters: Copy: 13 of 30 Spacing:SINGLE File format:ASCII Print mode:ASCII

```
 CCCC   OOO   N   N   SSSS   OOO   L      EEEEE
C      O   O  N   N  S      O   O  L      E
C      O   O  NN  N  S      O   O  L      E
C      O   O  N N N   SSS   O   O  L      EEEE
C      O   O  N  NN      S  O   O  L      E
C      O   O  N   N      S  O   O  L      E
 CCCC   OOO   N   N  SSSS   OOO   LLLLL  EEEEE
```

```
M   M  EEEEE  M   M          1
MM MM  E      MM MM         11
M M M  E      M M M          1
M   M  EEEE   M   M          1
M   M  E      M   M          1
M   M  E      M   M    ..    1
M   M  EEEEE  M   M    ..  111
```

```
*START* Job CONSOL Req #177 for HELENIUS    Date 15-Feb-80 17:00:11 Monitor: 1031 Great Pumpkin, TOPS-20
File PS:<HELENIUS>CONSOLE.MEM.1, created: 15-Feb-80 16:37:15, printed: 15-Feb-80 17:03:14
Job parameters: Request created:15-Feb-80 17:00:00   Page limit:270   Forms:NORMAL  Account:1
File parameters: Copy: 13 of 30   Spacing:SINGLE   File format:ASCII   Print mode:ASCII
```

1.0 OVERVIEW

The console subsystem is contained on a single extended hex module. It consists of a PDP/11 based system built around a single chip T11 processor with 64K bytes of dynamic ram memory and 2K bytes of console boot rom. The console also includes controllers to an RX floppy drive, three async lines and two VENUS CPU interfaces. The console interface is used in passing command requests or data between the VENUS CPU and the console subsystem during VMS operations of console diagnostic mode operations. The second is the diagnostics interface which enables the console to access each module within the VENUS CPU logic to provide initialization, configuration and diagnosis visibility.

The VENUS system console is an integral part of the CPU operation which performs:

1. CPU power up sequencing and diagnosis verification of modular power supply operation integrity.

2. MCA functional logic configuration and RESET operations.

3. The loading of E-box, I-box control stores, E-box constants and miscellaneous decoding rams.

4. VENUS CPU monitoring and detection of ambient temperature limits.

5. Initiate the VMS boot process and provide and interface to the system operator terminal and load device.

6. Provide a time of year clock to VMS operating system.

7. Provide a path for error log information to VMS.

8. Perform correction of E-box control store parity errors.

9. Perform the traditional lights and switches console functions.

2.0 MAJOR SECTIONS

1. The console processor is a TINY-11 cpu chip. This will be configured to operate with an eight bit data bus and an eight bit address bus.

   The TINY-11 has the capability to respond to sixteen levels of interrupts and has the facility to execute and overlap the memory refresh and allow DMA access between ISP memory cycles.

2. The TINY-11 memory address space consists of 64KB. This will contain:

   1. 2KB of I/O address space.

   2. 2KB of boot rom

   3. 64KB of dynamic ram. (eight bits plus parity)

3. The console load device is a random access RX floppy drive. The RX interface will provide the console read or write access to floppy data and with program control can transfer drive read data into memory using the TINY-11 DMA mechanism.

   The floppy will be an RX02 double density drive having the storage capability of 512KB of data.

4. The local terminal interface is a programmable async line and implement R242-c. It has the capability of being programmed to operate at standard baud rates up to 19,2k and standard character formats. The purpose of the local serial interface is to provide the functionality of either the VMS system or console ASCII terminal.

5. The remote terminal interface is a programmable serial async line which is provided to be used in remote diagnosis or in manufacturing as an APT load device.

   The interface is designed to provide full modem control and has the program capability to operate with split baud rates and up to 19,2K. Also provided is a maintenance mode loopback facility for use in console hardware diagnosability.

6. The optional serial line is provided as an extra programmable async terminal interface which has the ability of being operated with an internal or external baud rate clocking. This interface may be used by an HSC controller or another device to send error status information to the VENUS CPU.

7. The time of year clock is a 32 bit counter which will be battery backed up to provide a continuous time for a minimum of 100 hours of power outage.

8. the console/CPU interface is a dual port ram file. It is used to pass messages between E-box microcode and the console processor. This will support:

   1. RX floppy drive

   2. local terminal

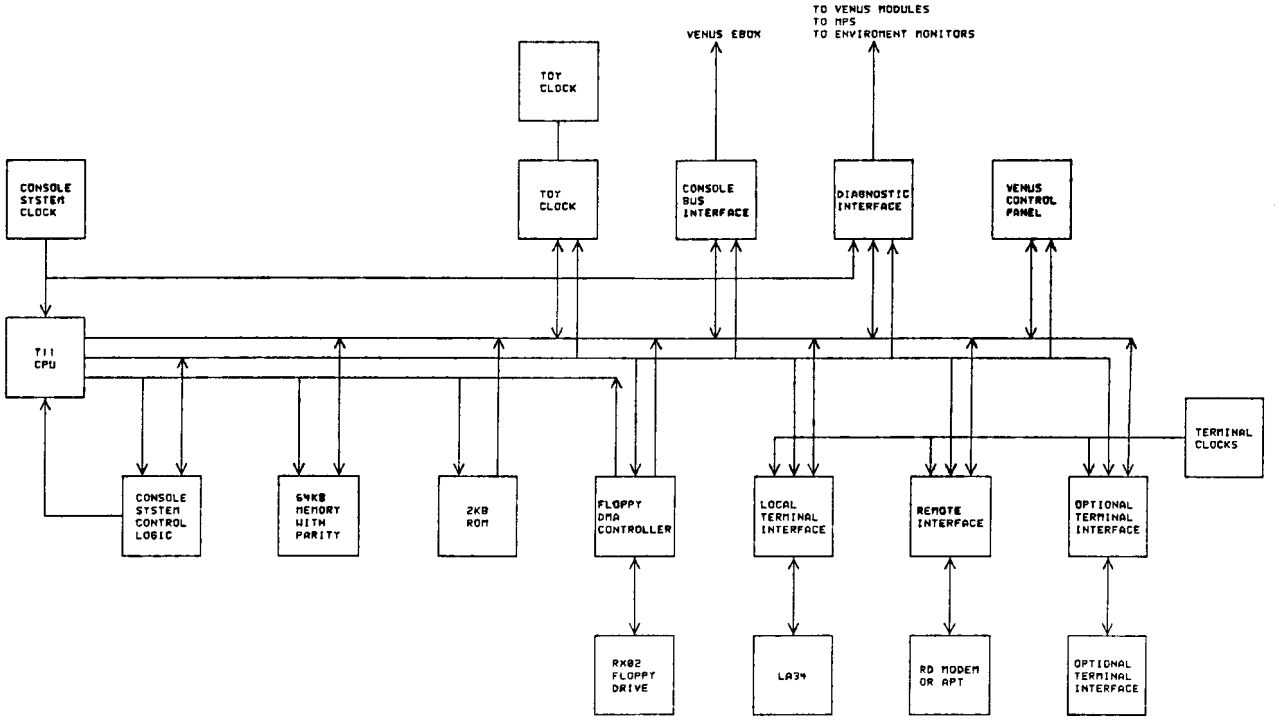
   3. remote terminal

   4. optional terminal

   5. time of year access

   6. system identity register

   7. system error messages



	DRN.	DATE 18-FEB-88	ENG.	DATE	TITLE:	
	CHK'D.	DATE	BOARD LOCATION: SHEET OF			
PS:<BISSON>CDC.DRW	15-FEB-88 15:26	NEXT HIGHER ASSEMBLY:			SIZE D	CODE CS
FIRST USED ON OPTION-MODEL:					NUMBER	REV.

MEMORY CONTROL FIELD

MCT <3:0> HEX CODE	P/H/Y/S	V/W/A	I/O/N/C	I/U/C/K	UTRAP ON T/S	UTRAP ON A/I	UTRAP ON C/S	UTRAP ON X/E	UTRAP ON A/G	UTRAP ON B/I	UTRAP ON M/T	MEMORY.CONTROL.FIELD
	0	V	I			N	N	N	N	N		MEM.NOP.I.OWN.VA
	0	V	E			N	N	N	N	N		MEM.NOP.E.OWN.VA
	0	V	E	I	R	Y	Y	N	*	N		READ.V.RCHK
	0	V	E	I	R	Y	N	-	-	N		READ.V.NCHK
	0	V	E	I	W	Y	Y	*	*	Y		READ.V.WCHK
	0	V	E	I	R	Y	Y	-	N	N		READ.V.RCHK.UNALIGNED
	0	V	E	I	R	Y	Y	-	-	N		READ.V.RCHK.UNALIGNED.NROT
	0	V	E	I	W	Y	Y	-	-	Y		LOCKREAD.V.WCHK
	0	V	E	I	R	Y	N	_	_	N		LOCKREAD.V.NCHK
	1	P	E	I	R	N	N	N	N	N		READ.P
	1	P	E	I	R	N	N	N	N	N		LOCKREAD.P
	0	V	E	I	W	Y	Y	*	*	Y		WRITE.V.WCHK
	0	V	E	I	W	Y	N	-	-	N		WRITE.V.NCHK
	0	V	E	I	W	Y	Y	-	N	Y		WRITE.V.UNALIGNED
	0	V	E	I	W	Y	Y	-	-	?		WRITE.UNLOCK.V.WCHK
	1	P	E	I	W	N	N	N	N	N		WRITE.UNLOCK.P
	1	P	E	I	W	N	N	N	N	N		WRITE.P
	0	V	E		W	N	N	N	N	N		PROBE.TB.V.WCHK
	0	V	E		R	N	N	N	N	N		PROBE.TB.V.RCHK
		X	E	NA		N	N	N	N	N		WRITE.TB.AND.TB.TAG
		X	E	NA		N	N	N	N	N		CLR.TB.PROC.SPACE
		X	E	NA		N	N	N	N	N		CLR.TB.PROC.AND.SYS.SPACE
		X	E	NA		N	N	N	N	N		READ.CACHE.WR.AND.VAL.BIT
		X	E	NA		N	N	N	N	N		WRITE.CACHE.WR.AND.VAL.BIT
		X	E	NA		N	N	N	N	N		WRITE.MBOX.STATE.REG
		X	E	NA		N	N	N	N	N		READ.MBOX.STATE.REG
		X	E	NA		N	N	N	N	N		READ.AND.CLR.MBOX.IB.ERROR.REG
		X	E	NA		N	N	N	N	N		READ.AND.CLR.MBOX.EBOX.MD.ERROR.REG
		X	E	NA		N	N	N	N	N		READ.AND.CLR.MBOX.IBOX.MD.ERROR.REG
		X	E	NA		N	N	N	N	N		READ.AND.CLR.MBOX.IO.ERROR.REG
		X	E	NA		N	N	N	N	N		READ.AND.CLR.MBOX.PBOX.ERROR.ADR.REG
		X	E	NA		N	N	N	N	N		READ.AND.CLR.MBOX.IO.ERROR.ADR.REG

Y=Utrap on condition
*=Utrap on condition unless MSC/ second reference is true
N=do not Utrap on condition
-=Hardware behavior undefined microcode must prevent condition

The TB hardware stores the following bits:

Field	No. of bits
Protection	4
Modify	1
Software valid	1
Ram contents valid	1
Address bits (29...09)	21
Parity bit for above	1
Tag bits (30..18)	13
Parity bit for tag	1
Total	43

4.0 INTERFACES

There are three interfaces between the Mbox and the rest of the system, the Pbox-Mbox interface, consisting of the MD bus and the VA bus, The I/O adapter bus (ABUS), and the memory interface, consisting of the memory address and data buses.

4.1 PBOX-MBOX INTERFACE

Addresses are sent from either the Ibox or Ebox to the Mbox on the VA bus. The cache always starts a cycle for a Pbox command if it is not busy. Approximately half way thru the cycle the Pbox must indicate whether the data VA lines contained a real request. Separate error registers are stored for various Pbox requestors to simplify error reporting and recovery.

The Mbox can handle a processor reference while the I/O is referencing main memory. This allows the Ebox/Ibox to get cache cycles while the Mbox is waiting for a memory response for the I/O request which is pending.

If there was an error in processing the request a trap occurs. In order of priority (highest first):

1. Hardware Error
2. TB Miss
3. Access Violation
4. TB Hit but Mbit write check

2.2 DATA CACHE TAG PARITY PROTECTION

The tag store contains 5 parity bits: P0, P1, P2 ,P3 and P4. The written bit is included in two of the parity bits so that an error in the written bit can be detected. The assignment of parity bits has been chosen so that the bad ram can be determined when an error occurs.

2.3 ERROR RECOVERY

The cache is a writeback cache without ECC appended to the data. Therefore, a backup mode is provided which can be turned on (when a ram or rams go flakey) which causes both caches to be written on all writes. Thus the cache becomes a 1 way associative cache for writes and a 2 way associative cache for reads. An extra copy (both tag and data) of all written cache locations are thus maintained. Whenever a parity error occurs the Ebox microcode invalidates the bad line in the cache with the error, forces a writeback from the good copy of the bad data failing location, and then invalidates both locations. Some effort is required to make the two copies of the cache identical for all locations within the line when a hit occurs in one cache only: A write to cache location whose written bit is off is declared to be a miss. By doing this the refill that will result will make both caches identical if the write pulse to each cache is enabled. No extra logic flows are required.

3.0 TRANSLATION BUFFER ORGANIZATION

The page table translation buffer (TB) is a one way set associative, write thru cache with a blocksize of one and 1024 entries. One way associativity was chosen so that the physical address bits 11:09 could be sent directly to the data cache without requiring a TB hit to first be determined. The translation buffer is indexed by VA <17:09> and VA bit 31 (system bit) so there are 512 entries for System space and 512 entries for Process space. Error recovery is accomplished by micro-code invalidating the entry. The valid bit is stored in the 256 X 4 RAMs so that 4 entries may be cleared in one write cycle. Since bit 31 is used as an index only 128 cycles are needed to clear either system or process space. This allows a clear of the user or system space in 8.5 microseconds.

1.0 MBOX OVERVIEW

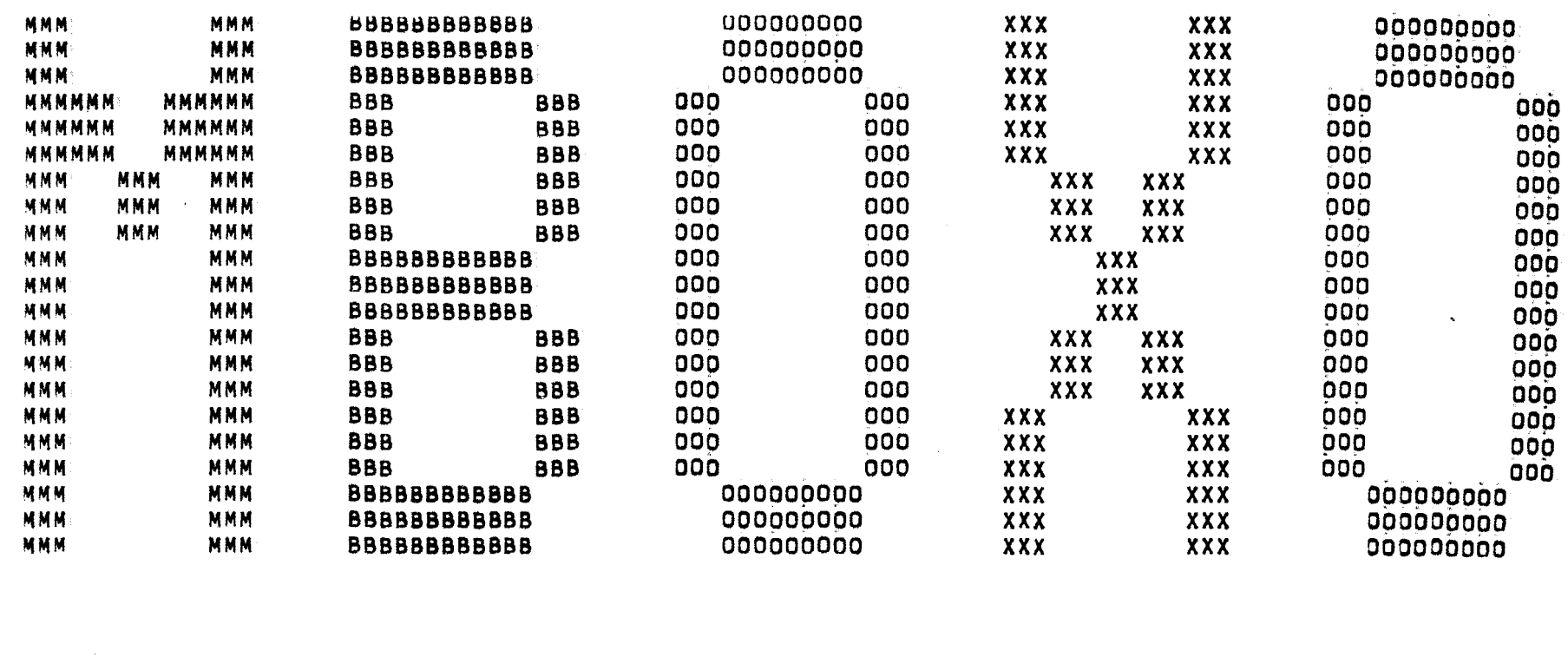
The Venus Mbox consists of a data cache, a page table translation buffer, an interface to main memory and an interface to the adapter bus (Abus) for the I/O controllers. This section details the operations performed by the Mbox and the interfaces between the Mbox and the rest of the Venus system.

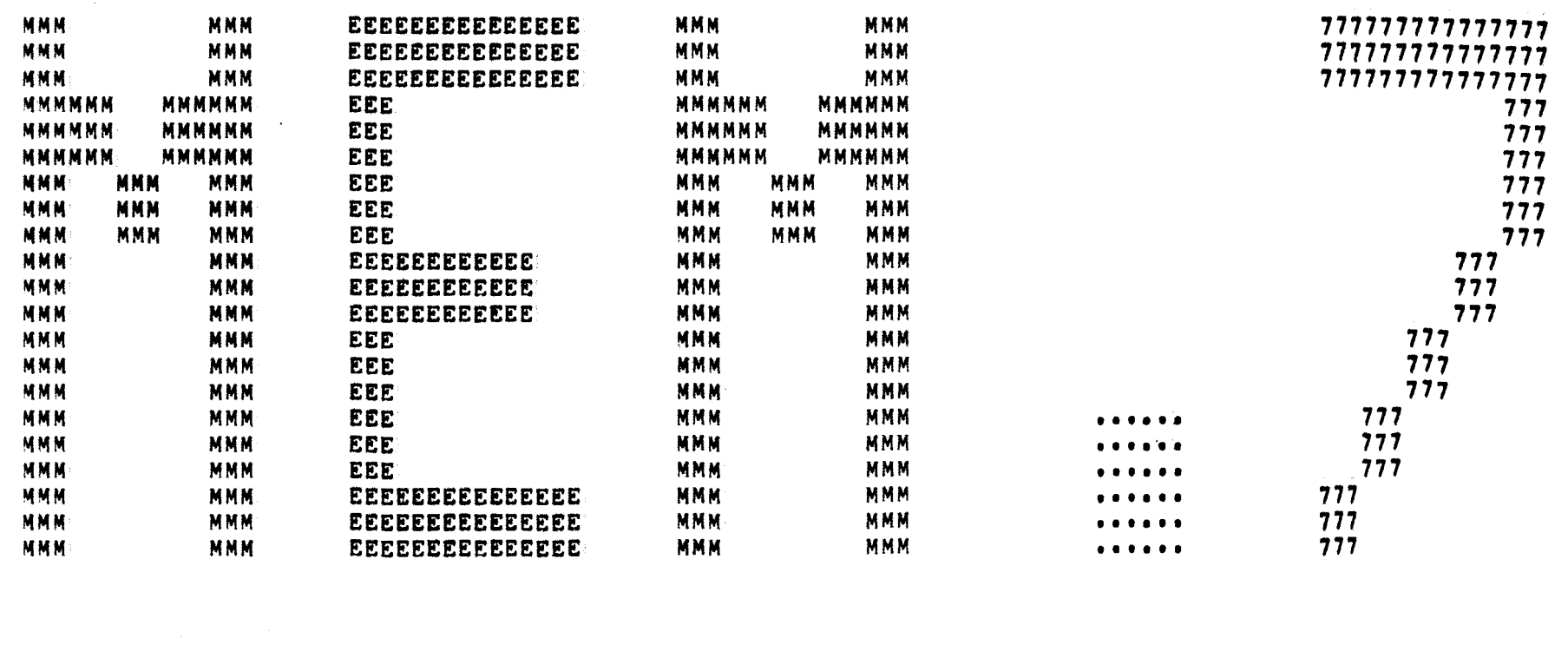
2.0 DATA CACHE ORGANIZATION

The cache is two way set associative. It is addressed by physical addresses only. There are 256 lines of four longwords in each cache for a total of 2048 longwords in the data cache. It is a writeback cache - a valid and a written bit is kept for each four longword line entry. Byte parity is stored with the data. A cache read operation will take 66.66 ns. if the data is in the cache. A cache Longword write operation will take 133.33 ns. A cache masked write operation will take 133.33 ns. The directory entry corresponding to each block of 4 longwords is called the "tag". The cache tag store is 256 entries high and includes the tag, the valid bit and the written bit.

2.1 Data Cache Parameters

1. Physically addressed cache - chosen since most cost effective.
2. Block size 16 bytes
3. Total size 8K bytes
4. Two way associative
5. Write back
6. LRU (least recently used) replacement
7. Write Allocate by block
8. The data cache stores byte parity.





START Job MBOXOV Req #495 for BRUCKERT Date 18-Feb-80 9:37:01 Monitor: 1031 Great Pumpkin, TOPS-20 Mor
File PS:<BRUCKERT>MBOXOV.MEM.7, created: 18-Feb-80 8:57:08, printed: 18-Feb-80 10:00:52
Job parameters: Request created:18-Feb-80 9:37:01 Page limit:810 Forms:NORMAL Account:1
File parameters: Copy: 21 of 30 Spacing:SINGLE File format:ASCII Print mode:ASCII

4.1.1 Alignment Of Processor References - All references from the Ibox to the Mbox must be longword aligned.

4.1.2 Mbox Error Registers - There are six error registers: Ibuf error register, Ebox MD error register, Ibox MD error register, I/O error register, Processor error address register, and I/O error address register. All error registers are read via the MD bus. The Ibuf and MD error registers store error bits for errors that are synchronous to the processor. The processor error registers (Ibuf, Ebox MD and Ibox MD) are automatically cleared when the error register is read. The I/O error register stores error bits for errors which are asynchronous to the processor microinstruction stream, including cache writebacks and errors on I/O transfers not involving the processor. The I/O error register is cleared by a command to the Mbox. No single error will be stored in more than one register: the Mbox will know whether the error is synchronous or asynchronous and record the error in the proper register. The microcode/software will know where to look by whether it received a fault response code or an interrupt. The processor error registers are cleared when read. The Ebox microcode should read the error address register first and then the appropriate error register. For Ibox errors, reading the error register allows Ibox requests again. The error registers contain error flags for the following types of errors:

1. TB parity error (processor only)

2. Cache data parity error (both I/O and processor errors)

3. Cache address (and V and W bits) parity error (both I/O and processor)

4. Memory correctable ECC error (both I/O and processor)

5. Memory uncorrectable ECC error (both I/O and processor)

6. No such location (both I/O and processor)

7. Memory address parity error from the ECC (both I/O and processor)

8. Abus parity error (both - the processor might be doing an I/O register read/write.) This Error is

4.2 I/O ADDRESS MAPPING

The Mbox hardware divides I/O address space into 1 Mb pieces. The assignment is programmable by the console. Each megabyte piece can be assigned to a specific Abus adapter. This must be performed at configuration time - dynamic run time changing of the I/O address space is not allowed.

4.2.1 Abus Device Addressing - The devices or adapters on the Abus will be addressed using I/O addresses (i.e. their control and error registers are in I/O space). Some of the address for the SBI adapter have been defined as processor registers. The micro-code will perform the conversion from processor register to I/O space.

4.2.2 Processor I/O References - A processor I/O reference is any reference to a physical address above 1FFF,FFFF.

4.2.3 ABUS MEMORY (EXTERNAL MEMORY) - Abus (SBI or BI) memory, WHICH IS UNCACHED, is assigned physical address space in 1 Megabyte sections. The configuration is performed by the console.

4.2.4 ABUS REQUEST DATA SOURCES -

Longword read	Data from cache	Data from memory
Quadword read	Data from cache	Data from memory
Octaword read	Data from cache	Data from memory
Longword write	Write in cache	Write memory
Octaword write	Invalidate cache, Write memory	Write memory
Masked (Byte) write	Write cache	Refill, then write cache

5.0 INTERRUPTS

The Mbox will not provide any special hardware for interrupts. This is deemed an Ebox/microcode problem.

6.0 INTERLOCKS

The Mbox implements a main memory interlock. When the Mbox receives a read lock request from I/O or Pbox it sets the internal memory interlock. For an exact description of the interlock see the ABUS specification.

7.0 PROCESSOR INITIATED READS AND WRITES

When the Mbox discovers that an address points to I/O space it will issue an Abus read or write request to the appropriate adapter. The request includes context information (mask bits,modify intent, and interlock). The selected adapter may not be able to process the requested operation from the processor immediately. Therefore the Mbox will be freed up and the processor held in a wait state. This allows I/O memory transfers to occur and prevents any deadly embrace.

8.0 MEMORY INTERFACE

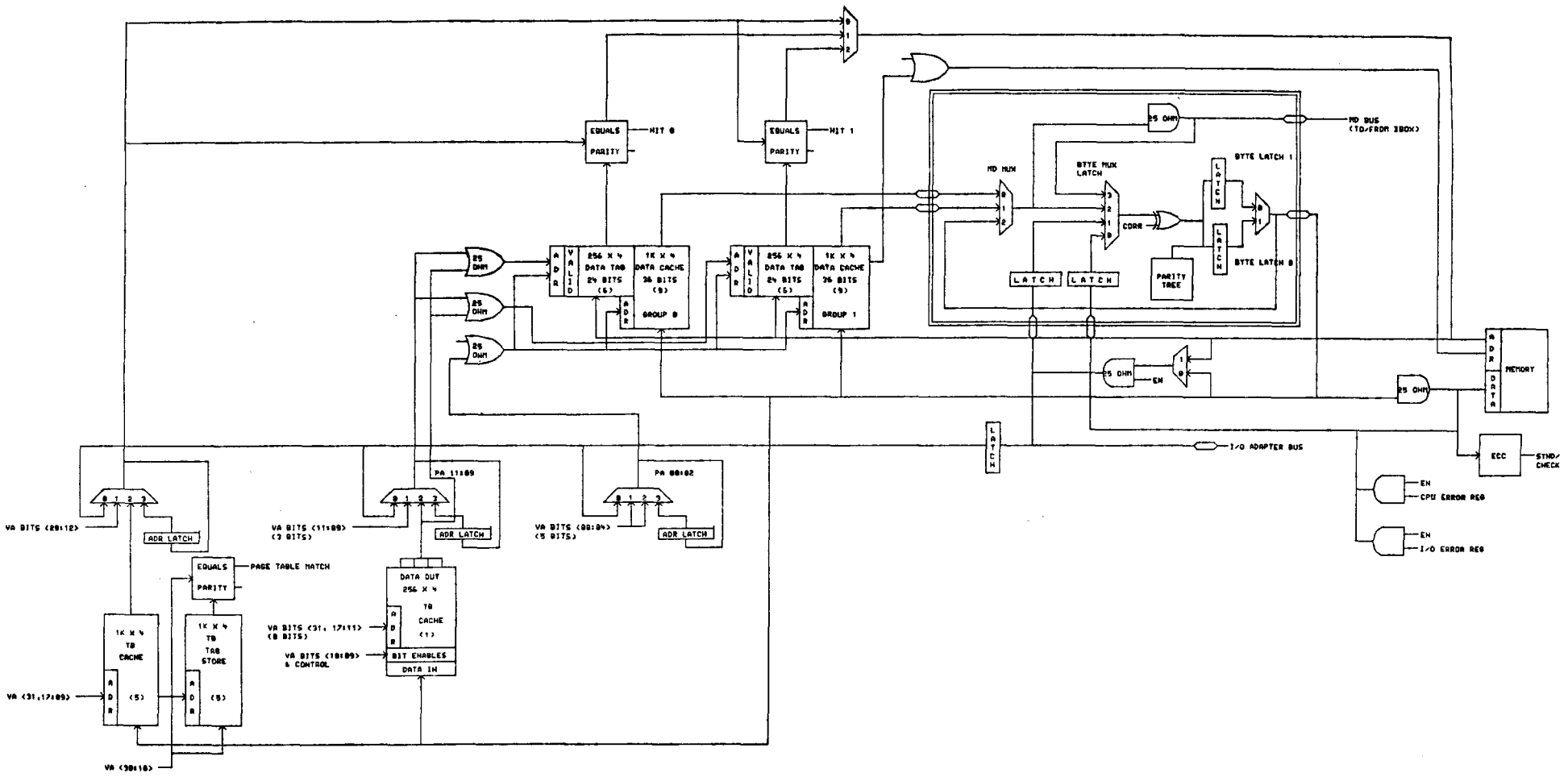
The MBOX is the only source of commands for the internal memory. In fact, the memory control logic is built on the same board as the Mbox. The MBOX is responsible the ECC generation and correction, and masked writes. The memory can only perform writes on 1, 2, 3, or 4 39 bit longwords. The data and address lines to the memory are separate. There is no parity checking in the storage modules. Address parity is included in the data ECC. This allows an address driver failure to be detected. The Venus Memory control is a subsystem within the MBOX. It receives its control over an interconnect with the MBOX and subsequently drives the storage arrays over the storage array bus. One or more <up to 8 max.> memory storage arrays may be used in the basic Venus configuration.

8.1 Memory Capacity

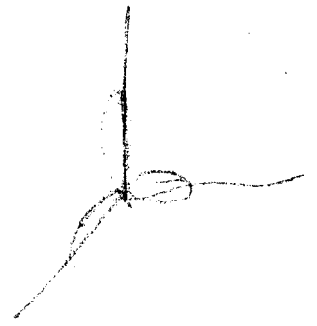
memory controller will be designed such as not to exclude the possibility of designing a bus repeater. Such a repeater will enable another 32 storage modules to be implemented in an expansion cabinet.

8.2 Refresh

Refresh on the arrays will be done in accordance with the Mos Ram specification. Array logic will establish refresh interval timing and the refresh address. Refresh contention logic is on the storage array making the refresh cycle transparent to the memory control.



	DRN.	DATE 18-FEB-88	ENG.	DATE	TITLE: VENUS MBOX BLOCK DIAGRAM
	CHK'D.	DATE	BOARD LOCATION: SHEET 1 OF 1		
VENUS:<MBOX>EM.DRW	22-JAN-88 19:07	NEXT HIGHER ASSEMBLY:		SIZE D	CODE CS
FIRST USED ON OPTION/MODEL:				NUMBER	RE



CHAPTER 1

EBOX DATA PATH SPECIFICATION

1.1 DATA PATH SPECIFICATION

The EBOX data path consists of several units. They are:

1. ALU
2. Shifter
3. AMUX
4. BMUX
5. Scratchpads
6. VMQ register
7. VMQ MUX
8. W register
9. ALU MUX
10. ALU Rotators
11. SHIFT Counter
12. SHF MUX

1.2 ARITHMETIC AND LOGIC UNIT

The arithmetic and logic unit does logic operations, binary arithmetic, and binary coded decimal arithmetic on 32 bit data quantities. Data arrives at the input to the ALU from the A and B mutiplexers. The ALU uses a fast carry lookahead carry technique to speed up the arithmetic functions.

All data that arrives at the A and B inputs of the ALU has odd longword parity. Logic in the ALU takes these parity bits and performs an appropriate transformation to predict the longword parity of the result of the ALU operation. Thus the ALU output is a 33 bit result, 32 data bits and an odd parity bit.

1.2.1 ALU Control

The ALU is controled by the UALU field of the MICROWORD.

MICROWORD ARITHMETIC AND LOGIC UNIT CONTROL FIELD = 5 BITS

HEX CODE	OPERATION
	BOOLEAN OPERATIONS
	A
	B
	A.OR.B
	A.AND.B
	A.AND.(.NOT.B)
	A.XOR.B

MICROWORD ARITHMETIC AND LOGIC UNIT CONTROL FIELD - 5 BITS

HEX CODE	OPERATION
	BINARY ARITHMETIC
	A+B
	A+B+(ALU CRY)
	A+B+(PSL CRY)
	A+4
	A-4
	A-B
	A-B-.NOT.(ALU CRY)
	A-B-(PSL CRY)
	B-A
	B-A-.NOT.(ALU CRY)
	B-A-(PSL CRY)
	DIVIDE.FNC

MICROWORD ARITHMETIC AND LOGIC UNIT CONTROL FIELD = 5 BITS

HEX CODE	OPERATION
	DECIMAL ARITHMETIC OPERATIONS
	A+B
	A+B+(ALU CRY)
	A+B+(PSL CRY)
	A-B
	A-B-.NOT.(ALU CRY)
	A-B-(PSL CRY)

A divide function is provided in the ALU to allow a 1 bit at a time non-restoring binary divide algorithm to be implemented. The divide function is specified in the MICROWORD ALU control field.

When the divide function is true the ALU operation is controlled by hardware. Note that the divide function will also control the source of the bit coming into the left end of the VMQ register. (See VMQ.REG and Shifter descriptions.)

If the clocked ALU.CRY bit is true the ALU will perform the function ALU.OUTPUT<--ALU.AMUX-ALU.BMUX.

If the clocked ALU.CRY bit is false the ALU will perform the function ALU.OUTPUT<--ALU.AMUX+ALU.BMUX.

1.3 SHIFTER

The Shifter performs shifts of up to 32 places, byte swaps for converting packed decimal data from memory byte order to arithmetic byte order and back, nibble swap for numeric string to packed decimal conversions, packed decimal to numeric string conversions, and unpacked floating to signed packed F-format conversions.

The input to the Shifter is a 64 bit quantity which comes from the ALU.ROT and BMUX outputs. The most significant half of this 64 bit quantity comes from the BMUX and the least significant half from the ALU.ROT. Parity bits are not included in the input to the Shifter because the Shifter does not predict the parity of

its output.

The output of the Shifter is a 33 bit result, 32 bits of data plus odd parity.

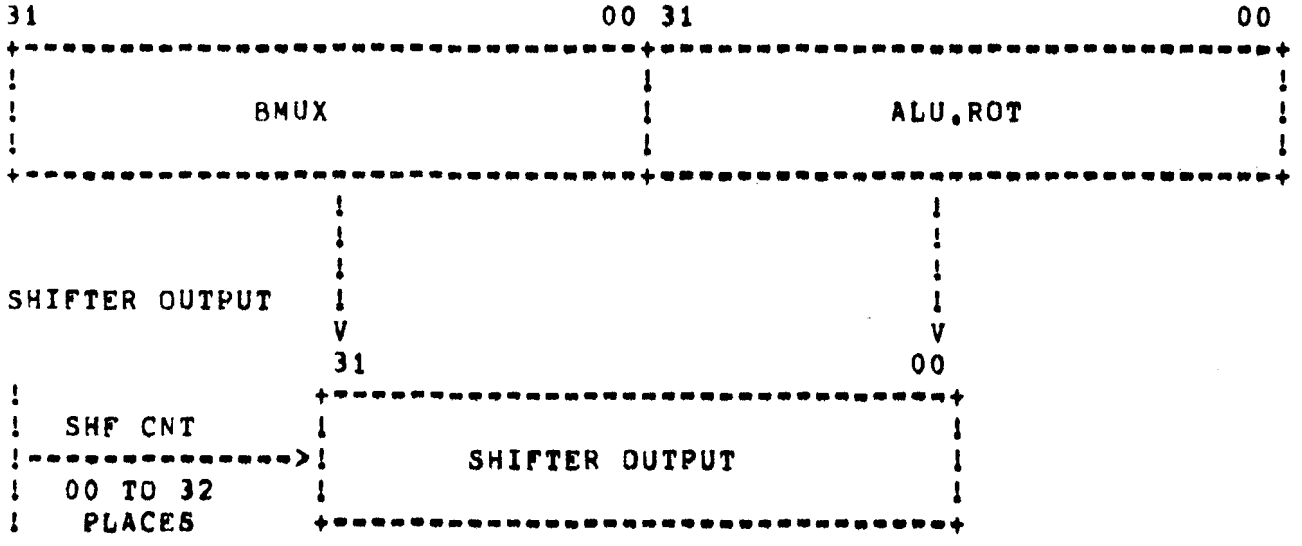
1.3.1 Shift FUNCTION

When shifting the Shifter is capable of selecting 32 bits of data from the 64 bit input to the Shifter. This is done by sliding a 32 bit window to the right across the 64 bit input the number of places specified in the shift count. The shift count field can specify a shift amount of 00 to 32 places.The shift count is a 6 bit control field that controls the operation of the shift matrix. This control field comes from the SHF MUX.

The selection of the shifter's data conversion functions is controlled by the shift count input to the shifter. Codes values greater than 32 decimal will be selected to specify these functions.

The operation of the Shifter when the shift count is non zero is shown below:

SHIFTER INPUT



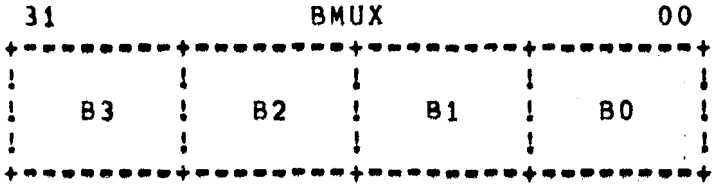
1.3.2 Shifter Byte Swap Operation

The Shifter is capable of performing a byte swap operation on a longword of data from the BMUX that converts memory format BCD data to a format suitable for use with the BCD ALU. This same byte swap operation can be used to convert the BCD output from the ALU to memory format BCD.

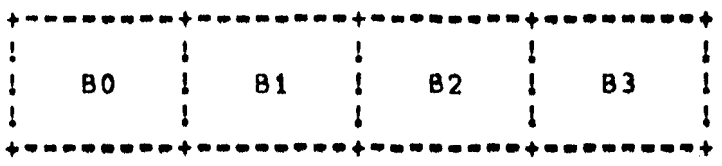
The order of the bits within a byte are not modified by the swap operation.

The byte swap that is performed is shown below:

SHIFTER INPUT



SHIFTER OUTPUT



SHIFTER<07:00><--BMUX<31:24>
SHIFTER<15:08><--BMUX<23:16>
SHIFTER<23:16><--BMUX<15:08>
SHIFTER(31:24><--BMUX<07:00>

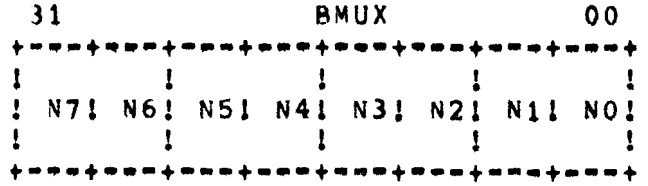
1.3.3 Shifter Nibble Swap For Numeric String To Packed Decimal Conversion

The Shifter provides a nibble swap operation to aid the conversion of numeric string data to packed decimal format.

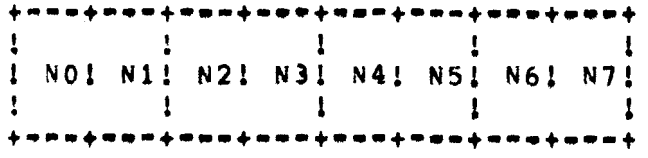
The conversion that is performed is shown below with each letter number pair representing a 4 bit nibble. The order of the bits within a nibble is not changed by the conversion operation.

The ALU.ROT inputs to the Shifter are ignored.

SHIFTER INPUT



SHIFTER OUTPUT NUMERIC STRING TO PACKED DECIMAL CONVERSION



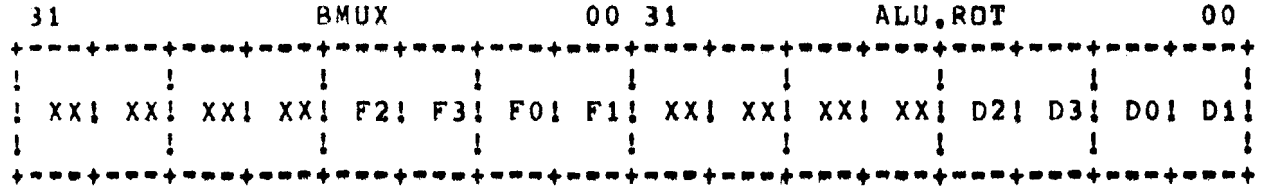
SHIFTER<03:00><--BMUX<31:28>
SHIFTER<07:04><--BMUX<27:24>
SHIFTER<11:08><--BMUX<23:20>
SHIFTER<15:12><--BMUX<19:16>
SHIFTER<19:16><--BMUX<15:12>
SHIFTER<23:20><--BMUX<11:08>
SHIFTER<27:24><--BMUX<07:04>
SHIFTER<31:28><--BMUX<03:00>

1.3.4 Shifter Packed Decimal To Numeric String Conversion Operations

The Shifter performs a conversion operation that is useful for converting packed decimal data to numeric string data.

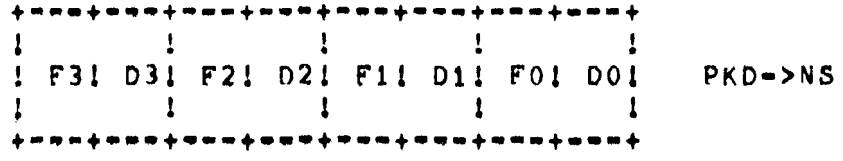
The conversion that is performed is shown below with each letter number pair representing a 4 bit nibble. The order of bits within a nibble is not changed by the conversion operation.

SHIFTER INPUT



XX= DON'T CARE

SHIFTER OUTPUT PACKED DECIMAL TO NUMERIC STRING CONVERSION



SHIFTER<07:00><--BMUX<07:04>!ALU,ROT<07:04>
SHIFTER<15:08><--BMUX<03:00>!ALU,ROT<03:00>
SHIFTER<23:16><--BMUX<15:12>!ALU,ROT<15:12>
SHIFTER<31:24><--BMUX<11:08>!ALU,ROT<11:08>

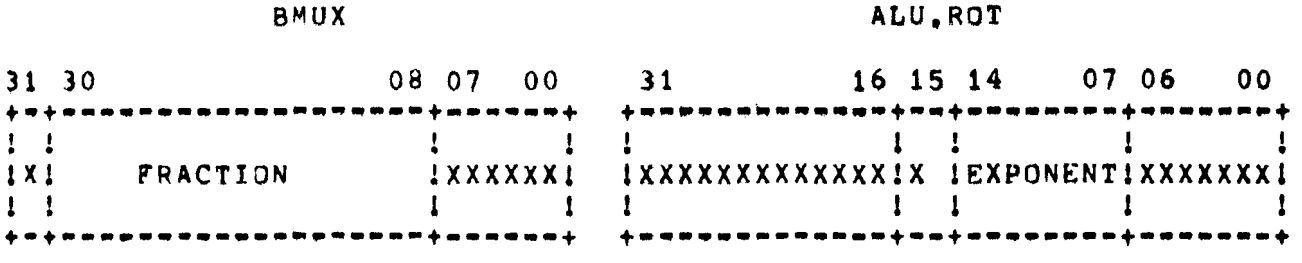
1.3.5 UNPACKED FLOATING TO PACKED F-FLOATING CONVERSION

The Shifter provides two operations which facilitates the conversion of unpacked floating point numbers to memory F-Format floating point number format.

The two conversions that are provided differ only whether the sign of the result gets set to a zero or a one.

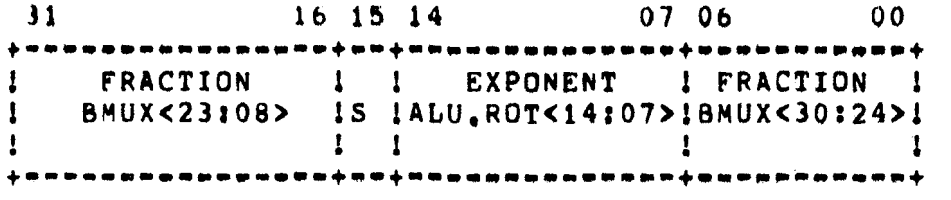
This conversion is shown below:

SHIFTER INPUT



X=DON'T CARE INPUTS

SHIFTER OUTPUT



SHIFTER<06:00><--BMUX<30:24>
SHIFTER<14:07><--ALU,ROT<14:07>
SHIFTER<15><-- 0 IF CONVERTING TO POSITIVE RESULT
1 IF CONVERTING TO NEGATIVE RESULT
SHIFTER<31:16><--BMUX<23:08>

1.3.6 Shift Right Two Function For Multiply

The shifter provides a shift right two function to support a two bit at a time multiply. This function is described below.

SHIFT RIGHT TWO

SHIFTER<31:00><--MUL.SGN<1:0>!ALU.ROT<31:02>

1.3.7 SHIFT LEFT ONE FOR DIVIDE

The shifter provides a shift left one place function to support a one bit at a time nonrestoring divide function. This shift function is show below.

SHIFT LEFT ONE

SHIFTER<31:00><--ALU.ROT<30:00>!VA<31>

1.3.8 Shifter Control

The Shifter is controled by a six bit control field. This field is encoded to minimize its width.

If the value in the control field is 00 to 32 (00 to 20 HEX) then the control field is interpreted to be the shift count.

If the value in the control field is 33 TO 47 (21 to 2F HEX) the shifter may be performing a data conversion function. The encodings that have been selected for each function are given below.

If the value of the control field is between 48 and 63 (30 to 3F HEX) the output of SHIFTER<03:00> will be the least significant HEX digit of of the control field. This function allows a 4 bit number to be read into the data path from the SHF.MUX.

HEX CODE	FUNCTION
00 to 20	SHIFT BY AMOUNT IN FIELD
2X	PACKED DECIMAL BYTE SWAP
2X	NIBBLE SWAP (NUMERIC TO PACKED DECIMAL CONVERSION)
2X	PACKED DECIMAL TO NUMERIC CONVERSION
2X	UNPACKED FLOATING POINT TO PACKED F-FLOATING CONVERSION POSITIVE SIGNED RESULT
2X	UNPACKED FLOATING POINT TO PACKED F-FLOATING CONVERSION NEGATIVE SIGNED RESULT
2X	SHIFT RIGHT TWO (MULTIPLY)
2X	SHIFT LEFT ONE (DIVIDE)
30 to 3F	SHIFTER<03:00><--HEX.CODE<3:0> SHIFTER<31:04><--ZERO

1.4 SHF MUX

The SHF MUX selects the source of the bits that control the Shifter. The SHF MUX is a four way multiplexer that has as its inputs SHF CNTR <05:00>, AMUX.LEADING.ZERO.CNT<3:0>, ALU<N,Z,C>,and ULIT<5:0>.

1.4.1 SHF MUX Control

The selection of the SHF MUX inputs is controlled by the USHF.MUX and UMSC fields of the MICROWORD. The UMSC field will override the USHF.MUX field when it selects a function that controls the SHF.MUX.

When UMSC=SHIFTER<03:00><--AMUX<31:16> LEADING ZERO COUNT function is enabled then SHF.MUX<5:4><--1's and SHF.MUX<3:0><--count equal to the number of leading zero's in AMUX<31:16>. This function is provided for to support the normalization of unpacked floating point numbers.

When UMSC=SHIFTER<03:00><--0|ALU.CC<N,Z,C> the SHIFTER output will be the ALU Condition Codes. This function is provided to allow the ALU condition to be saved. A corresponding Condition Code setting function allows the ALU condition codes to be restored.

If UMSC is not controlling the SHF.MUX then the USHF.MUX field controls the SHF.MUX.

1.4.2 USHF.MUX

USHF.MUX SELECT FIELD - 1 BIT

HEX CODE	FUNCTION
	SHF.MUX<05:00><--SHF.CNTR<05:00>
	SHF.MUX<05:00><--ULIT<05:00>

1.5 SHIFT COUNTER

The Shift Counter is a 8 bit register. It can be held, loaded, incremented by 1, or decremented by 1.

The output of the Shift Counter can be used to control the Shifter (See SHF.MUX description).

1.5.1 Shift Counter Control

The Shift Counter is controlled by the USHF.CNTR field of the MICROWORD.

1.5.2 USHF.CNTR - 2 BITS

HEX CODE	FUNCTION
	HOLD SHF.CNTR SHF.CNT<8:0><--SHF.CNTR<8:0>
	LOAD SHF.CNTR SHF.CNTR<8:0><--ALU.ROT<8:0>
	INCREMENT SHF.CNTR SHF.CNTR<8:0><--SHF.CNTR<8:0>+1
	DECREMENT SHF.CNTR SHF.CNTR<8:0><--SHF.CNTR<8:0>-1

1.5.3 State Register

The data path has an 8 bit state register that can be loaded from the output of ALU.ROT<07:00>. The microcode can branch on the bits in this state register.

1.6 AMUX

The AMUX provides the source of data for the A input of the ALU and the A input of the Shifter. The output of the AMUX is 33 bits wide, 32 data bits and an odd parity bit.

If scratchpad A (SPA) is selected as the source of data for the AMUX and a longword write to the same register in SPA is occurring on the WBUS, the hardware will automaticly select the copy of the data from the WBUS to supply to the output of the AMUX. This prevents the EBOX from using the stale copy of the data in SPA.

If the context of the above WBUS write is WORD then the data from the right half of the AMUX will come from the WBUS and the data from the left half of the AMUX will come from SPA.

If the context of the above WBUS write is BYTE then the data from AMUX byte 0 will come from the WBUS and the data from AMUX bytes 1 thru 3 will come from SPA bytes 1 thru 3.

Data coming from the operand bus does not always have an odd parity bit. When the data is supplied without odd parity the IBOX will supply the signal IBOX NO PARITY along with the data. When this occures the EBOX will generate parity for the data so that the AMUX output will always have an odd parity bit.

Data coming from the WBUS has byte parity. This is collapsed to longword parity before the data gets to the AMUX.

Data coming to all other AMUX inputs will always have an odd parity bit.

1.7 BMUX

The BMUX provides the source of data for the B input of the ALU and the B input of the Shifter. The output of the BMUX is 33 bits wide, 32 data bits and an odd parity bit.

If scratchpad B (SPB) is selected as the source of data for the BMUX and a write to the same register in SPB is occurring on the WBUS, the hardware will automaticly select the copy of the data from the WBUS to supply to the output of the BMUX. This prevents the EBOX from using the stale copy of the data in SPB.

If the context of the above WBUS write is WORD then the data from the right half of the BMUX will come from the WBUS and the data from the left half of the BMUX will come from SPB.

If the context of the above WBUS write is BYTE then the data from BMUX byte 0 will come from the WBUS and the data from BMUX bytes 1 thru 3 will come from SPB bytes 1 thru 3.

Data coming from the operand bus does not always have an odd parity bit. When the data is supplied without odd parity the IBOX will supply the signal IBOX NO PARITY along with the data. When this occures the EBOX will generate parity for the data so that the BMUX output will always have an odd parity bit.

Data coming from the WBUS has byte parity. This is collapsed to longword parity before the data gets to the BMUX.

Data coming to all other BMUX inputs will always have an odd parity bit.

1.7.1 AMUX And BMUX Control

The source of data that is directed to the outputs of the AMUX and BMUX is selected by the UABMUX field of the MICROWORD.

1.7.2 UABMUX

MICROWORD ABMUX SELECT FIELD = 2 BITS

HEX CODE	FUNCTION
	AMUX<31:00><--SPA<31:00> BMUX<31:00><--SPB<31:00>
	AMUX<31:00><--SPA<31:00> BMUX<31:00><--OP.BUS<31:00>
	AMUX<31:00><--VMQ<31:00> BMUX<31:00><--SPB<31:00>

1.8 A LATCH AND B LATCH

The A LATCH and B LATCH are data latches that are used to latch data that is read out of scratchpad A and scratchpad B during the first half of a microcycle. During the second half of a microcycle the scratchpads are written with data from the WBUS. These latches are controlled by the hardware and are not visible or controllable by the microcode.

1.9 SCRATCHPAD ADDRESSING

The EBOX contains two 256 word x 36 bit/word ramfiles that contain identical copies of the 16 General Purpose Registers (GPR's) and 240 scratch locations that are used by the microcode. The ramfiles are called scratchpad A (SPA) and scratchpad B (SPB) and their outputs are inputs to the A LATCH and B LATCH, respectively.

Data is read out of SPA and SPB during the first half of a microcycle and during the second half both scratchpads are written with data from the WBUS.

Separate control fields are provided in the MICROWORD for specifying the addresses of the data that are to be read from SPA and SPB. Reads from SPA are controlled by a 9 bit field and reads

from SPB are controlled by an 8 bit field in the MICROWORD.

Writes to the scratchpads may originate from IBOX or EBOX requests. The arbitration of who gets to write the scratchpads is controlled by the WBUS Arbitrator. EBOX write requests win out over IBOX write requests.

If a GPR is being written by the IBOX during a microcycle and the EBOX attempts to read that same GPR the control logic in the EBOX will take the data off the WBUS in place of the data in its scratchpads as the read data.

The location that the EBOX writes when it modifies the scratchpads is controlled by a 4 bit field in the MICROWORD. One of the encodings in this field specifies that no write is to take place.

Scratchpad writes from the EBOX take place on the cycle after the scratchpad write command occures in the microword. The data to be written is loaded into the W register on the cycle in which the scratchpad write command is issued. The presence of a scratchpad write command causes a request for the WBUS to be made to the WBUS Arbitrator. The EBOX will either receive a grant during the cycle the write command was issued or it will stall until it receives grant.

Once the grant is received and the clock ticks the W register will be enabled onto the WBUS.

If a read to the location that is being written occures while the data is on the WBUS the hardware will substitute the data from the WBUS for the read data from the scratchpad location. This makes the delayed write appear invisible to the microcode. It also means that there is no time penalty for using data from a scratchpad location that the EBOX has just written.

If the write context was less than a longword then only the rightmost bytes will come from the WBUS and the remaining bytes will come from the scratchpad.

The scratchpads store byte parity. The byte parity is converted to longword parity when data is read out of the scratchpads so that longword parity is presented to the AMUX and BMUX.

1.9.1 SPA Read Address

The read address of SPA is controlled by 9 bits from the MICROWORD.

Eight of the bits form the USPA,ADR field. This field can be used to address all 256 locations in SPA.

The nineth, USPA,SPADR,REG,SEL, determines whether the or not the SPADR,REG will be used to form the SPA address. SPADR,REG is a 4 bit register that can address 16 scratchpad locations.

If USPA,SPADR,REG is false the USPA,ADR field will be used to address SPA.

If USPA,SPADR,REG is true the high 4 bits of USPA concatenate with SPADR,REG<3:0> will be used to form SPA's address. In this mode of addressing USPA<7:4> specify the page number of a 16 word page within the scratchpads that can be addressed by SPADR,REG.

1.9.2 Microword Control Of Scratchpad A's Read Address - 9 BITS

MICROWORD CONTROL OF SCRATCHPAD A'S READ ADDRESS

USPA,SPADR,REG,SEL	FUNCTION
0	SPA,ADR<7:0><--USPA<7:0>
1	SPA,ADR<7:0><--USPA<7:4>|SPADR,REG<3:0>

USPA	FUNCTION
	8 BIT MICROWORD FIELD THAT IS USED TO ADDRESS THE 256 LOCATIONS IN SPA

Note that there is no time for decoding the USPA field to eliminate the USPA,ADR,REG in the MICROWORD.

1.9.3 SPB Read Address

Scratchpad B is always addressed by USPB.ADR field of the MICROWORD. The USPB.ADR field is an 8 bit field and is capable of addressing all 256 locations of SPB.

1.9.4 USPB - 8 BITS

SCRATCHPAD B READ ADDRESS

USPB	FUNCTION
	8 BIT MICROWORD FIELD THAT IS USED TO ADDRESS ALL 256 LOCATIONS OF SPB

1.10 WRITING SPA AND SPB

Anytime the scratchpads are written the same location in both is written with the identical data. The data that is written has byte parity.

Writes to the scratchpads can originate from writes the EBOX has initiated or from writes to the GPR's that were initiated by the IBOX.

If the EBOX initiated the write then the destination address is determined by the scratchpad destination (USP.WR) field of the MICROWORD.

1.10.1 EBOX Scratchpad And GPR Write

The writing of Scratchpad and GPR locations is controlled by the USP.DST field of the MICROWORD.

If USP.DST is equal to NOP then no scratchpad or GPR write will take place.

If the destination field (USP.DST) of the MICROWORD is not equal to NOP in microcycle 0 then a location in the scratchpads will be written at the end of microcycle 1. Whether or not the location is a GPR is determined by the the scratchpad address that is generated. The data that is being written during microcycle 1 will be available for use by the microcode during that cycle because of the WBUS feedback path in the data path.

The assertion of WBUS GPR WRITE(See WBUS SPEC), and WBUS ADR are determined by the scratchpad destination logic. The WBUS DATA CTX is controlled by the UDT field of the MICROWORD. The data context must be specified during the same cycle that USP.DST is asserted.

If a microtrap occures in microcycle 0 then the arbitrator will not enable the data onto the WBUS. On return from the microtrap routine the the microinstruction that specified the GPR or scratchpad write will be retried.

If a microtrap occures during microcycle 1 then, assuming that WBUS ABORT is false, the data will be written into the GPR or scratchpad location in all boxes during microcycle 1. This must be done because GPR and Scratchpad writes are pipelined operations.

If WBUS ABORT is asserted during microcycle 1 then the GPR or scratchpad write will not take place. The assertion of WBUS ABORT cause a microtrap in the EBOX. The data that was being written will be lost unless the box that was doing the writing makes provision to save the data. The microtrap routine will execute a machine check specifying whether the instruction in progress should be aborted or faulted.

1.10.2 Scratchpad Write Address And Write Control

The write address for EBOX scratchpad writes is controlled by the USP.WR field.

One coding of the USP.WR field specifies that the scratchpads will not be written.

1.10.3 USP.WR - 4 BITS

SCRATCHPAD WRITE ADDRESS FIELD

HEX CODE	LOCATION WRITTEN	
0	SP[10]	SCRATCHPAD TEMPORARY 0
1	SP[11]	SCRATCHPAD TEMPORARY 1
2	SP[12]	SCRATCHPAD TEMPORARY 2
.	.	
.	.	
.	.	
9	SP[19]	SCRATCHPAD TEMPORARY 9
A	SP[1A]	SCRATCHPAD MEM.WRITE.SAV LOCATION
B	SP[(0|SPADR.REG<3:0>)]	GPR ADDRESSED BY SPADR.REG
C	SP[(USPA<7:0>)]	SCRATCHPAD A'S ADDRESS
D	SP[(USPB<7:0>)]	SCRATCHPAD B'S ADDRESS
E	SP[(USPA<7:4>|SPADR.REG<3:0>)]	SCRATCHPAD PAGE MODE
F	NOP - NO SCRATCHPAD LOCATION WILL BE WRITTEN	

Note that scratchpad locations 10 thru 1A are temporaries that the microcode can use when doing 3 address operations such as C<--A+B where A, B, and C are different scratchpad locations.

Note that the above scratchpad addressing fields allow the the following types of scratchpad operations to be performed

SCRATCHPAD OPERATIONS

SP[(USPA<7:0>)]<--SP[(USPA)]+SP[(USPB)]

SP[(USPA<7:0>)]<--SP[(USPA<7:4>|SPADR,REG<3:0>)]+SP[(USPB)]

SP[(USPB<7:0>)]<--SP[(USPA)]+SP[(USPB)]

SP[(USPB<7:0>)]<--SP[(USPA<7:4>|SPADR,REG<3:0>)(USPA)]+SP[(USPB)]

SP[TEMP]<--SP[(USPA)]+SP[(USPB)]

SP[TEMP]<--SP[(USPA<7:4>|SPADR,REG<3:0>)]+SP[(USPB)]

SP["0"HEX|SPADR,REG<3:0>]<--SP[(USPA)]+SP[(USPB)]

SP["0"HEX|SPADR,REG<3:0>]<--SP[(USPA<7:4>|SPADR,REG<3:0>)]+SP[(USPB)]

SP[(USPA<7:4>|SPADR,REG<3:0>)]<--SP[(USPA)]+SP[(USPB)]

SP[(USPA<7:4>|SPADR,REG<3:0>)]<--SP[(USPA<7:4>|SPADR,REG<3:0>)]+SP[(USPB)]

+ denotes any ALU function

1.10.4 Scratchpad And GPR Write Context

The context of a scratchpad writes is controlled by the memory context field data type(UDT) field of the MICROWORD if the write is to GPR location.

If the write is to any other location in the scratchpad the write context is always longword.

1.11 WRITE DATA TYPE

The UDT field of the MICROWORD controls the context of EBOX writes to the GPR's and memory, and the setting of condition codes.

The data types that can be specified in this field are byte, word, longword, and instruction dependent.

If the field specifies instruction dependent data type the data type comes from a ram that is indexed by the opcode of the microinstruction and PSL<CMP>. The data type in this case can be byte, word, longword, quadword, or octaword.

1.11.1 UDT-2 BITS

HEX CODE	FUNCTION
0	BYTE
1	WORD
2	LONGWORD
3	INSTRUCTION DEPENDENT - USE DATA TYPE SUPPLIED BY IBOX IN COMPATABILITY MODE AND DATA TYPE FROM A RAM IN EBOX THAT IS INDEXED BY THE EXTENDED OPCODE IN NATIVE MODE.

1.11.2 SPADR.REG

The scratchpad address register (SPADR.REG) is an 4 bit register that can be used to address a bank of 16 scratchpad registers.

This register is loaded by the IBOX if a source, modify, or destination operand is a GPR. The register can only be loaded at the begining of a microcycle on which the EBOX is waiting for an operand.

There is no direct way to read SPADR.REG. However, the address can be be read indirectly if the constants 00 thru 15 decimal are stored in a block of scratchpad locations that starts at an address that is a multiple of 16. When this block of locations is read using SPADR.REG the constant that is read out of the location will be the value stored in SPADR.REG.

The EBOX can load this register from the output of the ALU by executing a LD.SPADR.REG MICROWORD function

The EBOX can increment the SPADR.REG by executing a INCR.SPADR.REG MICROWORD function and can decrement the register by executing a DECR.SPADR.REG MICROWORD function.

To support compatibility mode a function which inclusively OR's a one into least significant bit of SPADR.REG is provided. This function can be invoked by executing an OR.SPADR.REG MICROWORD function.

This register appears to the microcode to be a single register but due to the timing of operand passing between the EBOX and IBOX it will have to be implemented as two registers. One is loadable and controlled by the IBOX. The other is under control of the EBOX.

The two register implementation is required because optimizable instructions require that the GPR address be available at the start of operand passing microcycle so the EBOX can do:

GPR<--GPR.[ALU.OP].OPBUS

GPR<--SPA.[ALU.OP].OPBUS

To do this the GPR number has to be available at T0 of the microinstruction because only the delay of a flip flop, mixer, and output cell can be tolerated in the path that generates scratchpad addresses. This requires that the IBOX copy of the GPR address be in a register inside the scratchpad address MCA.

At the end of the microinstruction the contents of the IBOX copy of the GPR address is copied into the EBOX copy of this register so the scratchpad write can take place during microinstruction+1. In microinstruction+1 the ebox could be executing a another operand pass cycle for the next microinstruction which could also be optimized and would need to access a different GPR.

Writes will always use the EBOX copy of this register for the write address.

Note also that the IBOX loads its copy of this register at the same T0 on which it asserts operand valid and the EBOX may not have stored the result of the previous instruction by this time.

1.11.3 SPADR,REG Control Functions

The UMSC and ULIT fields of the microword are used to control the scratchpad address register. If UMSC CONTROL SPADR,REG is not selected then the contents of SPADR,REG can only change when an operand is passed to the EBOX from the IBOX.

UMSC = CONTROL SPADR,REG

ULIT	FUNCTION	
	LD,SPADR,REG	SPADR,REG<03:00><--ALU<03:00>
	INCR,SPADR,REG	SPADR,REG<03:00><--SPADR,REG<03:00>+1
	DECR,SPADR,REG	SPADR,REG<03:00><--SPADR,REG<03:00>-1
	OR,SPADR,REG	SPADR,REG<03:00><--SPADR,REG<03:00>,OR,^X01

1.12 ALU,MUX

The ALU,MUX is two way multiplexer that is used to select the AMUX or ALU as the source of data to the shifter.

1.12.1 ALU,MUX Control

The ALU,MUX is controlled by the UALU,MUX field of the MICROWORD.

UALU,MUX

HEX CODE	FUNCTION
	ALU,MUX<31:00><--ALU<31:00>
	ALU,MUX<31:00><--AMUX<31:00>

1.12.2 ALU Rotators

The ALU Rotators are two four way multiplexers on the output ALU MCA that rotate the bits within a nibble from 0 to 3 places to the left before sending ALU,MUX and BMUX data to the shifter, shift counter, state register, and and BMUX Conditions MCA.

The rotators are controlled by the least significant two bits of the SHF,MUX.

1.13 VMQ REGISTER

The VMQ register is used as the Virtual Address register for addressing memory, Multiplier register when doing a two bit at a time multiply, and the Quotient register for a 1 bit at a time divide.

The VMQ register can be loaded from the ALU, shifted left one place, shifted right two places, and held.

1.13.1 VMQ Register Control

The VMQ register is controlled by the UVMQ field of the MICROWORD.

1.13.2 UVMQ

UVMQ

HEX CODE	FUNCTION
	HOLD,VMQ VMQ<31:00><--VMQ<31:00>
	VMQ,GETS,ALU VMQ<31:00><--ALU<31:00>
	VMQ,GETS,VMQ,LEFT1 VMQ<31:00><--VMQ<30:00>|ALU,CRY,OUT,UNCLK'D

	VMQ,GETS,VMQ,RIGHT2
	VMQ<31:00><--VMQ,IN<31:30>|VMQ<31:02>
	IF (SHIFTER,FUNC=RIGHT,TWO) THEN
	VMQ,IN<31:30><--ALU<01:00>
	ELSE
	VMQ<31:30><--00

22-Nov-82 15:39:31,905;000000000001
Date: 22 Nov 1982 1537-EST
From: PETTY at KL2263
To: kotok at KL1031, bell at KL1031, derosa at ISHTAR, Aubut at DEMILO,
Ball at KL2263, Beaven at KL2263, Cook at KL2263, Dale at DEMILO,
Flanagan at KL2263, Gallagher at DEMILO, Gibson at DEMILO,
Gilby at DEMILO, Glackemey at DEMILO, Leveille at DEMILO,
Moore at DEMILO, Pasquantonio at KL2263, Petty at DEMILO,
Roch at KL2263, Shively at DEMILO, Shoop at KL2263, Somers at ISHTAR,
Stevens at KL2263, Tibbetts at DEMILO
cc: anton at ESTA, balboni at ESTA, glorioso at KL1031
Subject: ***** IT LIVES *****
Message-ID: <"MS11(2333)+GLXLIB1(1056)" 11874037772.49.114.10509 at KL2263>

This message is being typed to you on the Venus Prototype 0
console terminal. Every character passing through the Venus console
T11 on its wayto tnis timesharing system.

Yes Virginia, there is a Santa Claus!!

Bob
cc: BJ, Ulf, Kotok.
a most positive, impressive report. Keep it going!

NOV 29 1982

```
**********
*        *
* d!i!g!i!t!a!l *
*        *
**********
```

INTEROFFICE MEMORANDUM

TO: Ulf Fagerquist
CC: Gordon Bell
Bill Johnson

DATE: 22 November 1982
FROM: Bob Glorioso
DEPT: LVE
EXT: 5915
LOC/MAIL STOP: MR01-2/E47

SUBJECT: MONTHLY REPORT - NOVEMBER

VENUS

- Venus is still progressing well despite rumors to the contrary.
- VAX instructions currently running on the Venus CPU Sage model are: ADDB3, BEQ, HALT, NOP, MOVL, MOVQ, MOVC3. Note, however, that each instruction has only been tested using selected register modes and memory reference modes.
- VSS has been working well for most of November.
- Ball! Great. The first network message has been sent from Venus Proto One via the console - see attached.
- M Box Sage is off the dime and they have run several more tests despite unreliability of Emily (KL10).
- The Operations group has gotten Emily under control this past week - they had it rebuilt over a weekend and now have it under Field Service contract.
- Hatchet (Small AXE) has been run in TUMS. The first two tests uncovered three microcode problems.
- The E Box database has been built for Autodly and we remain on schedule for timing analysis - E Box starts this week.
- Both MCA sockets (one from each vendor) pass the elctrical tests and the pin grid array is progressing. A decision on the pin grid array will be made in the next two weeks.
- Ten layer boards from Fujitsu still looking good but we may not make the first proto with them. We will use multiwire as an alternative.

- Board submissions are progressing on schedule. We have been helped by the change in Jupiter submissions by taking advantage of extra multiwire capability for first prototype boards. Current prototype completion date is March 10 with some expectation to pull it in to February.
- MCA submissions and turnarounds still going well. Crunch is still expected in January.

- We have started a cooperative effort with the languages group to optimize compilers (FORTRAN in particular) to take advantage of the Venus pipeline.

- Continuous work on performance model still showing modest improvements in performance as the model gets more refined. Initial transaction processing (HP/Kawpas) results show 3.7 second response time at 510 users.

- The Quarterly Review with ABI (Bell) went very well. Carl Gibson is working directly with Bart Donahue of ABI to provide them with technical help and performance modeling assistance. We should be able to provide them with the prototypes they need.

- Ken Waine will do Venus DVT - plan due by 1 December.

- I have begun to push for patent submissions on Venus and have initiated a series of papers on Venus for publication at and after announcement. I want to give the engineers a chance to grow professionally, have good records of the design and help sell machines.

KEY RISKS (Action Person)

- Our loss of John Golenbieski from F Box may impact its completion. Attrition is still a problem and can really hurt if it continues. (Bob Glorioso/Greg Gaines)

- There is still some risk in the processing of boards - through the PC Factory. Continued support from Andover on IDEA is critical. (Sultan Zia/Jeff Singer/Dave Copeland)

- Implementation of the Data Base coordination for simulation and protos. (Bill Walton/ John Bloem)

- Keeping all our computers and networks going. (Bob Glorioso/Dave Copeland)

OTHER

- Morningstar System concept (Venus with CI and NI only on A-Bus) and product requirements were presented to and accepted by the Venus Marketing Task Force.

- Jeff Singer, Joe Zeh and I have initiated a focused effort to get Supervax I defined, staffed and under way. Supervax I will be a "cleaned-up" Venus with improved technology.

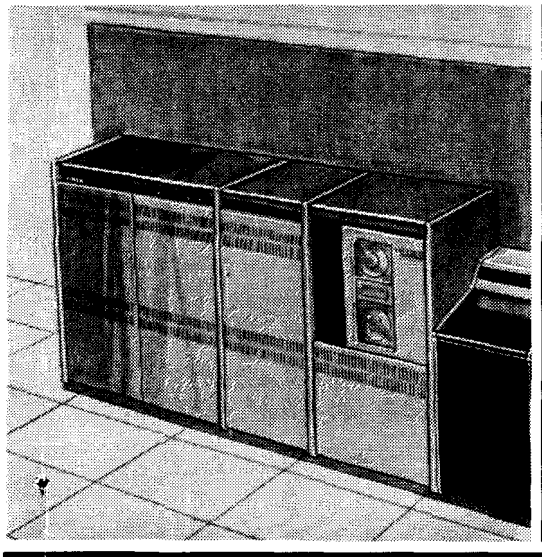
- Dave Orbits is completing the current phase of Hypervax work. A report and several seminars on the Short Tick model will be completed by the end of January.

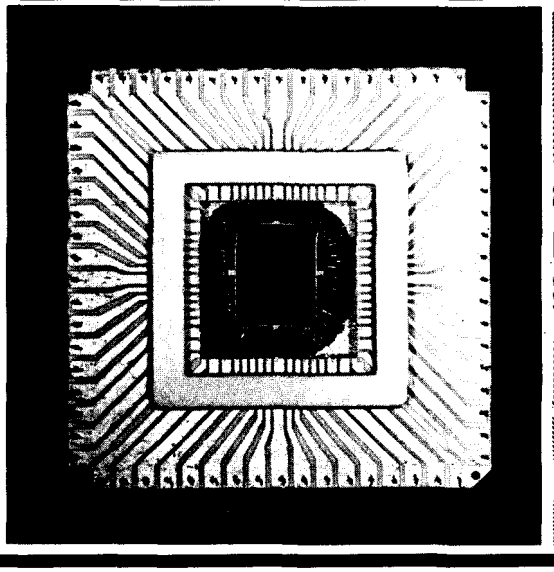
PHASE 1

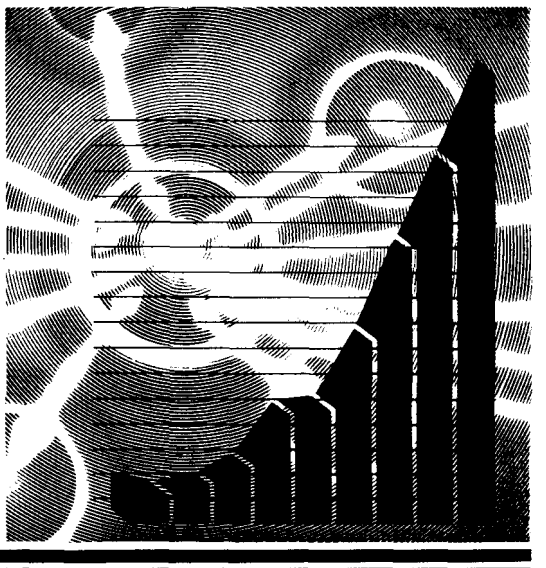
SYSTEM BUSINESS

PLAN

VOLUME I







PHASE 1

SYSTEM BUSINESS

PLAN

VOLUME I

"It is a bad plan that admits of no modification."

Publilius Syrus
Maxim 469

Phase 1 System Business Plan
Rev 2.0 September 1982

This plan is organized into four major sections, packaged in three volumes:

Volume I

Section 1 Executive Summary

Volume II

Section 2 System Description, Forecasts and Assumptions

Section 3 Financial and Sensitivity Analysis

Volume III

Section 4 Appendices containing relevant and supporting detail information

Appendix A: Product Line Detail
B: Service Detail
C: Audit
D: Software Contribution

To reduce distribution costs, Volumes II and III will be distributed only upon request.

Contact Ruth Jobin, DTN 231-6732, for copies.

System Project Team

Engineering Group Manager	Bob Glorioso	231-5915
Product Manager: System	Carl Gibson	231-6779
Product Manager: Software	Peter Ross	231-4471
Product Manager: Hardware	Peter Schay	231-5784
Program Manager	Sultan Zia	231-6277
CPU Engineering Managers	Alan Kotok	231-7381
	Bill Walton	231-6274
Manufacturing Manager	Chas. Bradshaw	231-6115
Customer Services Manager	Walter Manter	231-6503
Diagnostics Manager	Carl Gibson	231-6779
MEC Manager	Peter Emery	231-4401

SECTION I. EXECUTIVE SUMMARY VENUS BUSINESS PLAN

1.1 System Definition Follow-on to VAX-11/780. Provides >4 x 11/780 system throughput at comparable costs for systems above $250K MLP. Uses new I/O architectures, but is backward compatible to 11/780 at extra cost. Absolutely VMS compatible to application programs.

1.2 Prioritized Goals

1.2.1 Marketing Goals

#1. Customer satisfaction superior to comparable IBM system.

#2. Maximum life cycle profit as measured by overall program IRR.

#3. Growth (investment protection) for entry and mid-range VAX customers.

1.2.2 Development Goals

#1. Quality

#2. Time To Market (6/84)

#3. Life-Cycle Cost of Ownership less than comparable 11/780

#4. Minimal practical system under $50K T/C with 4x 11/780 performance

#5. New I/O architecture CI, NI, HSC

#6. 11/780 compatibility for migration

#7. Minimal development expense

1.2.3 Specific Financial Expectations

- After tax internal rate of return of 43% (Corp. Minimum Guideline = 40%)
- Accrual breakeven date: FCS + 5 quarters
- Cash breakeven date: FCS +3.25 years
- MLP Sales over life (System) >$6.9B (13,000+ system)
- Gross Margin: >55%
- Operating profit as % of total NOR= >30%

1.3 Relationship to Corporate Product Strategy

VENUS is the high-end VAX system called for by the strategy.

1.7. Risk Assessment

		1	2	3	4	5	6	7	8	9	10	
Technology												
For Industry	well established, know how to do						X					never done before
For Digital	well established, know how to do							X				never done before
Simulation									X			
Cycle time									X			
Individual box designs						X						
Computer resources							X					
Manufacturing												
Test Procedures	well established						X					new techniques
Processes	well established						X					new
Service	similar to many products, high ease of diagnosis				X							new techniques, new service procedures, much training, sparing
Marketing												
Distribution Channels	traditional end-user direct sales, or OEM/reseller, standard terms and conditions		X									new distribution channels, new terms and conditions, new order processing procedures
Customer Base	current Digital customers			X								new Digital customers
Customer Capability	highly technical					X						computer naive
Applications	well understood, done before				X							new applications

PRINTED IN U.S.A. EJ 19340 26/80 060 03 01

JAN 14 1983

```
**********
*        *
* digital*
*        *
**********
```

TO: Distribution

DATE: January 10, 1983
FROM: Sultan M. Zia
DEPT: L.V.E.
EXT: 231-6277
LOC/MAIL STOP: MR01-2/E47

SUBJECT: MINUTES OF VENUS QUARTERLY REVIW MEETING JANUARY 6, 1983

ATTENDEES: Sam Appleton, Dileep Bhandarkar, Ken Brabitz, Charlie Bradshaw, Richard Candor, Dave Copeland, Dick Cygan, Bill Demmer, Peter Emmery, Tom Eggers, John Ewalt, Ulf Fagerquist, Bob Glorioso, John Grose, Len Kreidermacher, Alan Kotok, Vic Ku, Janice LeBlanc, Walter Manter, Jeff Mariano, Trudy Matthews, Don McInnis, Bill Munson, Bob Murphy, Matt Nolan, Moshe Roditi, Peter Ross, Peter Schay, Don Simon, Vehbi Tasar, Ken Waine, Bill Walton, Joe Zeh, and Sultan Zia.

See attachments for the material presented.

Feedback from reviewers:

Bill Demmer

- Manufacturing should have a contingency plan to meet the Ramp rate even if engineering fails to provide 3 chip level isolation.
- He is very pleased with the progress in our simulation effort. Very pleased with CAD groups' effort.
- He is concerned that the minimum level of testing in the hardware debug is at the box level.

Ulf Fagerquist

- He is concerned that both M and I boxes are on the critical path for hardware power on.
- He recommends that we do not commit to ship any prototypes to any customer beyond ABI and Schlumberger.

- He agreed to work the resource conflict in the board layout area between Venus and Jupiter.

- He would like to see at the next Quarterly Review a detailed discussion on the revision control process.

Charlie Bradshaw

- Charlie feels good about the progress the Venus program has made to date.

- Although he is concerned with the Ramp rate he believes that Marlboro manufacturing is equipped to meet it.

Walter Manter

- Impressed with the results so far on the Venus program.

- He would have liked to have seen a presentation by Reg Burgess on CSSE accomplishments.

Patrick Courtin

- Restated the fact that the demmand for Venus in the marketplace is huge in particular ABI as a potential large customer.

Bill Munson

- Bill congratulated the Venus Product Management Group for a excellent job done in their dealings with the customers.

SZ/jrl
Attachemnts

PROGRAM OVERVIEW AND STATUS - SULTAN ZIA

VENUS PROGRAM STATUS

RELATIVE TO PLAN

FUNCTIONAL VERIFICATION

THE 5 MACRO INSTRUCTIONS RUNNING ON THE ENTIRE CPU	DEC 7, 82
DIRECTED AXE TESTS RUNNING (HATCHET)	FEB 22, 83
EXCEPTION TESTS RUNNING	MARCH 7, 83
MEMORY MENAGEMENT TEST RUNNING	MARCH 21, 83
FAULT TOLERANT TESTS RUNNING	APRIL 18, 83
ALL HARDWARE AVAILABLE	APRIL 7, 83/ FEB 15, 83
MACRO HARDCORE TESTS RUNNING ON THE PROTOTYPES	MAY 15, 83/ MARCH 15, 83
100K AXE TEST CASES RUNNING (WITH EMMULATORS)	JUNE 30, 83/ MAY 15, 83
START VMS DEBUG	JUNE 30, 83/ MAY 15, 83
UETP RUNNING ON PROTOTYPE	OCT 30, 83/ SEPT 15, 83

TIMING VERIFICATION

AUTODLY TOOL VERIFIED	FEB 1, 83
START TIMING VERIFICATION ON E/F BOXES	FEB 1, 83
START TIMING VERIFICATION ON M/I BOXES	MARCH 30, 83
TIMING VERIFICATION OF ALL BOXES DONE	MAY 30, 83
DEFINE THE SPEED OF THE MACHINE	JUNE 15, 83
START SECOND PASS SUBMISSION OF MODULES AND MCA'S	JUNE 30, 83

FUNCTIONAL VERIFICATION

SUBMIT FINAL PASS MCA'S	OCT 30, 83/ SEPT, 83
PROTO RUNNING VMS AT F.C.S. SPEED	DEC 30, 83/ NOV 83
START DMT AND FIELD TEST	FEBRUARY, 84/ JAN 84
F.C.S.	JULY, 84/JUNE 84

PROGRAM RISKS

1. SUCCESS IN OUR SIMULATION EFFORT	(ROBERT GLORIOSO)
2. MANUFACTURING'S ABILITY TO MEET THE RAMP RATE	(BOB MURPHY)
3. MODULAR POWER SUPPLIES	(PETE EMERY)
4. RELIABILITY OF THE H7140 POWER SUPPLY IN THE F.E. CABINET	(PHILIP TAYS)
5. SEVEN MODULES ARE LIKELY CANDIDATES FOR THE 10 LAYER BOARDS	(WARREN PELUSO/ BILL WALTON)

CPU FUNCTIONAL VERIFICATION/STATUS - ALAN KOTOK

VENUS DESIGN VERIFICATION STATUS

OUTLINE

- PROCESSES
  - SAGE (CPU)
  - TUMS
  - TIMING SPEC VERIFIER
  - AUTODLY
- MACHINE SUBSYSTEMS
  - IBOX
  - EBOX
  - MBOX
  - FBOX
  - I/O
  - CLOCK
  - CONSOLE

ALAN KOTOK
6 JAN 83

SAGE

- VSS PROBLEMS FIXED
- CPU SAGE NOW RUNNING SEVERAL MACROINSTRUCTION SEQUENCES
- SEMI-AUTOMATIC CPU MODEL-BUILD PROCESS HAS BEEN CREATED, DOCUMENTED AND TESTED
- PARALLEL TEST EFFORTS NOW UNDER WAY, IN ATTEMPT TO INCREASE THE NUMBER OF ERRORS FOUND PER DAY
- ERRORS FOUND THRU CPU SAGE:

EBOX	IBOX	MBOX	FBOX
2	24	5	2

- ERROR DISCOVERY RATE NOW ABOUT 7 TO 10 PER WEEK
- PROBLEMS
  - NETWORK LINKS TO "SAGE 1" AND "SAGE 2"
  - ~~"PROCESS" MANPOWER~~ – SOLVED
  - NEED A "BREAK" FEATURE IN USS/SAGE
  - NEED TO PULL TOGETHER MBOX WITH SIGNIFICANT SIZE MEMORY ARRAY INTO CPU MODEL

ALAN KOTOK
6 JAN 83

TUMS DEBUG STATUS AS OF

12/31/82

BOX and/or BOX INTERFACE	TESTED	TESTED W/ PROBLEMS	UNTESTED
1. IBOX			
o ucode	98 %		2 %
o logic DP	Data Not Avail.		
o logic CTL	Data Not Avail.		
2. EBOX			
o ucode	cime only [1]		
o logic DP	Data Not Avail.		
o logic CTL	Data Not Avail.		
3. MBOX			
o ucode	See MBOX table		
o mmgmnt logic	Data Not Avail.		
o cache/ cache ctl	Data Not Avail.		
o sequencer	Data Not Avail.		
4. FBOX	NA [2]		
5. IBOX/EBOX INTERFACE			
o stalls	WBUS,DBUS lmted		
o fork dispatch	All but few cases		
o data buses	All but few cases		
o error logic	0 %		100 %
o misc	Data Not Avail.		
6. IBOX/MBOX INTERFACE			
o stalls			
o memory rqs (CTL)	Limited Cases		
o memory rqs (data)	Limited Cases		
o errors	0 %		100 %
7. IBOX/FBOX INTERFACE	NA [2]		
8. IBOX/CONSOLE INTERFACE	0 %		100 %

notes:

[1] CIME stands for the "Cursory" I/E/M box TUMS model. Major algorithms of the EBOX ucode have been debugged in this model.

[2] FBOX TUMS model is currently not available. John Mcallen of the LSCAD group is responsible for its implementation. Delays due to loss of resources.

TUMS IBOX DEBUG STATUS AS OF

12/31/82

IBOX Functionality (implemented in TUMS)	TESTED	TESTED W/ PROBLEMS	UNTESTED
1. IBOX ucode [1]			
o specifier procsing (s#L,i[Rx],...,L^D)	*		
o access modes (RMWA * BWLQO)	*		
o Unaligned refs	*		
o RLOG Unwind	*		
o RAF			*
LOGIC:			
2. ACU - DP	All except as noted		o Unpacker o Opbus Sn extend o Output Rotator
3. Resource Stalls	70 %	5 %	25 %
4. Control Logic	Data Not Avail.		
5. Errors			o CPC,ISA, ESA Log o RLOG Unwind o Hardware Error Logic

notes:

[1] All Ibox uword location (except for Reserved Addressing Mode faults, R.A.F.) have been executed at least once. This, however, does not imply path analysis (i.e., all possible paths due to branching.

TUMS MBOX DEBUG STATUS AS OF

12/31/82

MBOX Functionality (visible to ucode)	TESTED	TESTED W/ PROBLEMS	UNTESTED
MBOX UCODE FUNCTIONS:			
o CP Read/Writes [1]	*		
o Write Back	*		
o Write Back with error		*	
o Refill	*		
o Write Cache	*		
o Write B Cache Off	*		
o Cache Data Correction		*	
o Non Fatal Cache Tag Parity Error		*	
o Write TB PTE			*
o READ PTE (DIAG)			*
o READ PTE Tag (DIAG)			*
o Write PTE			*
o Write EMD			*
o Probe Write			*
o Clear TB Entry			*
o CP I/O REQ	NOT AVAILABLE ON TUMS MODEL		
o Sweep Cache			
o Invalidate Cache			
o Write/Read MBOX REG			
o Read/Write IOA			

notes:

[1] All references from IBUF-Port, OP-Port, and Ebox are considered CP requests from the point of view of the MBOX.

TUMS DEBUG STATUS AS OF 12/31/82

ERROR DISCOVERY ANALYSIS FOR Q2 FY'83

	CLASS OF PROBLEM	# OF PROBLEMS	% OF TOTAL
1.	TUMS Modelling Problems	30	42.25 %
2.	AXE Interface Problems	11	15.5
3.	VMS 3.1 Upgrade	1	1.4
4.	UCODE (E,M,I BOX)	9	12.6
5.	DESIG (HWARE)	20	28.25
	Totals	71 [1]	100 %

notes:

[1] Actual number was 70, but 1 of the eeors was both designed and modelled incorrectly.

TIMING SPEC VERIFIER

- HAS BEEN RUN OVER COMPLETE SET OF INTERBOX SPECS
- 10 VIOLATIONS REMAIN, ALL UNDER 2 NS
- WE WILL NOT PUSH HARDER, SINCE SPEC DATA NOT SUFFICIENTLY PRECISE
- NOW ABLE TO FEED SIGNAL TIMING CONSTRAINTS TO AUTODLY WITH VARIABLE CYCLE TIME

ALAN KOTOK
6 JAN 83

AUTODLY

- HAVE SUCCESSFULLY LOADED FULL EBOX INTO VERSION 3.1 OF AUTODLY
- INCLUDES ALL SIGNAL PROPAGATION DELAYS FROM PLACED BOARDS AND MCA'S, BUT NOT BACKPLANE
- EBOX IS THE TESTBED FOR RINGING OUT MORE PROBLEMS IN AUTODLY SYSTEM
- VERSIONS 3.2 AND 4.0 COMING IN NEXT 2 MONTHS TO CORRECT DEFICIENCIES AND ENHANCE CAPABILITIES
- AUTODLY INPUT FILE BUILD PROCESS WORKING FROM EXISTING DATA BASE
- PROBLEMS
  - RESOURCE CONFLICTS IN CAD GROUP TO COMPLETE VERSION 4.0 AND NEW DATA BASE MECHANISMS ON SCHEDULE

ALAN KOTOK
6 JAN 83

IBOX

- STANDALONE SAGE PASSED "5 MACROINSTRUCTION" TEST
- IN CPU SAGE THE FOLLOWING FUNCTIONS HAVE RUN SUCCESSFULLY:
  - FETCHED INSTRUCTIONS INTO IB FROM BOTH MEMORY PORTS
  - DECODED SPECIFIERS AND OPCODES TO START IBOX MICROCODE; UPDATED THE PC AND SHIFTED IB
  - CALCULATED EFFECTIVE ADDRESS OF OPERANDS AND FETCHED THEM FROM CACHE. SEVERAL ADDRESSING MODES USED.
  - PASSED IMMEDIATE OPERANDS TO EBOX
  - PASSED ALL DATA NECESSARY TO START EBOX
  - TAKE RESULTS FROM EBOX AND SEND TO MEMORY
  - EXECUTED CONDITIONAL AND COMPUTED BRANCHES, BOTH TAKEN AND NOT TAKEN
  - PIPELINE SUCCESSFULLY FLUSHED AND REFILLED
  - IBOX TRAPS FROM EBOX EXECUTED
  - USE OF IB FOR STRING INSTRUCTION DATA
  - LOAD AND DRAIN OF INTERSTAGE BUFFERS
  - ETC., ETC., ETC.

ALAN KOTOK

EBOX

- ALL BOX LEVEL SAGE TESTS COMPLETED
- NO OUTSTANDING DESIGN PROBLEMS: SOLUTIONS TO ALL PROBLEMS FOUND IN SAGE AND TUMS HAVE BEEN IMPLEMENTED
- EXPERIMENTAL AUTODLY ANALYSIS IN PROGRESS
- 4 OUT OF 10 MCS EMULATORS BUILT AND TESTED

ALAN KOTOK
6 JAN 83

MBOX

- BOX SAGE DEBUG STILL IN PROGRESS
- THINGS WHICH RUN:
  - CACHE READS AND WRITES, VARIOUS ALIGNMENTS
  - READ AND WRITE ALL MBOX REGISTERS
  - ECC GENERATION AND CHECKING FOR CACHE AND ARRAY
  - CORRECTION CYCLES
  - SDB LOADING OF MICROCODE
  - LONG, QUAD AND OCTAWORD DMA READS, CACHE HIT
  - CACHE SWEEPS
  - MEMORY MANAGEMENT FAULTS
  - BRANCH OPTIMIZATION SEQUENCE
  - EBOX FIRST AND SECOND REFERENCE
  - MULTIPLE PORT REQUESTS

ALAN KOTOK
6 JAN 83

MBOX, CONTINUED

- BELIEVED TO WORK (TESTS NOT DEFINITIVE YET)
  - DMA READS, CACHE MISS
- KNOWN PROBLEMS
  - DMA READ/WRITE WITH PARITY ERRORS
  - DMA MASKED OPERATIONS (NOW WORKS)
  - PROCESSOR LOCK REQUESTS
  - READ AND WRITE OF PAMM
- UNKNOWN STATUS (NOT TESTED YET)
  - DMA WRITES CACHE MISS
  - VARIOUS SEQUENCES OF OPERATIONS, IN COMBINATION
  - SOME ERROR HANDLING
- SUMMARY
  - I BELIEVE WE HAVE A WORKABLE DESIGN, BUT PROGRESS IS SLOWER THAN DESIRED

ALAN KOTOK
6 JAN 83

FBOX

- ALL FBOX INSTRUCTIONS HAVE BEEN TESTED IN STAND-ALONE SIMULATION
- ABOUT 4,000 CASES HAVE BEEN RUN SUCCESSFULLY
- NO KNOWN OUTSTANIDNG PROBLEMS
- SDB AND ERROR DETECTION AND REPORTING TESTED WITH SAGE
- A FEW FBOX INSTRUCTIONS HAVE BEEN RUN SUCCESSFULLY ON CPU SAGE MODEL, INCLUDING SOME SEQUENCES
- FBOX MICROCODE UNDERGOING OPTIMIZATION TO MAKE MORE ROOM FOR SELF-TEST CODE

ALAN KOTOK
C JAN 83

I/O SYSTEM

- I/O BACKPANEL DEBUGGED ON P1
- FRONT END CAB CHECKED OUT
- A LARGE ASSORTMENT OF PERIPHERALS WHICH WILL BE CONNECTED TO PROTOTYPES HAVE BEEN CHECKED OUT, USING THE VENUS INTERNAL CONTROLLERS WHEN APPROPRIATE
- SBI ADAPTER
  - TIMING SPECS MESHED WITH REST OF SYSTEM, AND NO SIGNIFICANT PROBLEMS UNCOVERED
  - SAGE SIMULATIONS HAVE BEEN RUN SUCCESSFULLY, BUT
    - DC022 MODEL NEEDS REBUILDING TO MATCH TIMING REQUIREMENTS
    - MORE TESTS MAY BE NEEDED TO COVER ALL SEQUENCES

ALAN KOTOK
6 JAN 83

CLOCK MODULE

- REV "B" BOARD NOW AVAILABLE
- HARDWARE DEBUG WILL BEGIN SOON USING CONSOLE SDB LOGIC
- VCO WILL BE CHECKED OUT
- SAGE SIMULATION ALSO BEING DONE, SOMEWHAT BELATEDLY

CLOCK RULES

- SOLIDLY ESTABLISHED
- MOST MODULES HAVE HAD LITTLE DIFFICULTY CONFORMING TO THE RULES
- A FEW DISCREPANCIES HAVE BEEN FOUND BETWEEN RULES AND SEIN/REFLEX ANALYSIS
- CLOCK RULE CHECKING NOW IN A PROGRAM

ALAN KOTOK
6 JAN 83

CONSOLE

- 2 CONSOLE BOARDS BUILT AND IN DEBUG
- HAVE SUCCESSFULLY DOWN-LINE LOADED SOFTWARE TO CONSOLE RAM FROM A VAX 11/780
- HAVE SUCCESSFULLY LOADED AND EXECUTED RT11 PRIMARY BOOTSTRAP FROM RL02 ON QBUS
- CURRENTLY DEBUGGING RT11 SECONDARY BOOTSTRAP
- THINGS NEEDING ADDITIONAL VERIFICATION
    - INTERRUPTS
    - QBUS DMA READS FROM T11 RAM TO THE RL02
    - TIME OF YEAR CLOCK
- STAND ALONE TESTER NEAR COMPLETION

ALAN KOTOK
6 JAN 83

STATUS OF VERIFICATION TOOLS - JEFF SINGER

VENUS CAD TOOLS

SAGE SIMULATION

- SIMULATOR & VSS STABLE
- PROBLEM/FEATURE BACKLOG BEING WORKED
- CPU BUILD PROCESS IN-PLACE & DEBUGGED

AUTODLY

- RELEASES: 1.0, 2.0, 3.0, 3.1 ARE OUT
- RELEASES 3.2 (JAN), 4.0 (MAR) SCHEDULED
- DATA BASE PROCESS STILL FRAGILE/INCOMPLETE

TUMS

- RELEASE 2.2
- ENHANCEMENTS FOR SPEED/FBOX MODEL SCHEDULED FOR JANUARY

JEFF SINGER
1/6/83

CPU BUILD SCHEDULE - BILL WALTON

VENUS HARDWARE STATUS

		[SEPT QUARTERLY REVIEW DATES]
CONSOLE		
SOFTWARE DEBUG PROGRESSING WELL		
CLOCK		
- MODULE AVAILABLE	4 JANUARY	[1 NOV 82]
- DEBUG IN PROTO STARTS	10 JANUARY	
EBOX		
3 CONTROL MODULES AVAILABLE	14 JANUARY	
1 DATA PATH	21 JANUARY	
2 CONTROL STORE BDS AVAILABLE	26 JANUARY	
B.P.	AVAILABLE NOW	
P.O.	26 JANUARY	[28 JAN 83]
FBOX		
BOTH BOARDS AVAILABLE	14 FEBRUARY	
B.P. AVAILABLE	18 FEBRUARY	
P.O.	18 FEBRUARY	[25 FEB 83]
MBOX		
1ST BOARD AVAILABLE	28 FEBRUARY	
2ND & 3RD BOARD	8 APRIL	
POWER ON	8 APRIL	[11 FEB 83]
IBOX		
1ST BOARD DUE	21 MARCH	
POWER ON	8 APRIL	[18 FEB 83]

W. WALTON
6 JAN 83

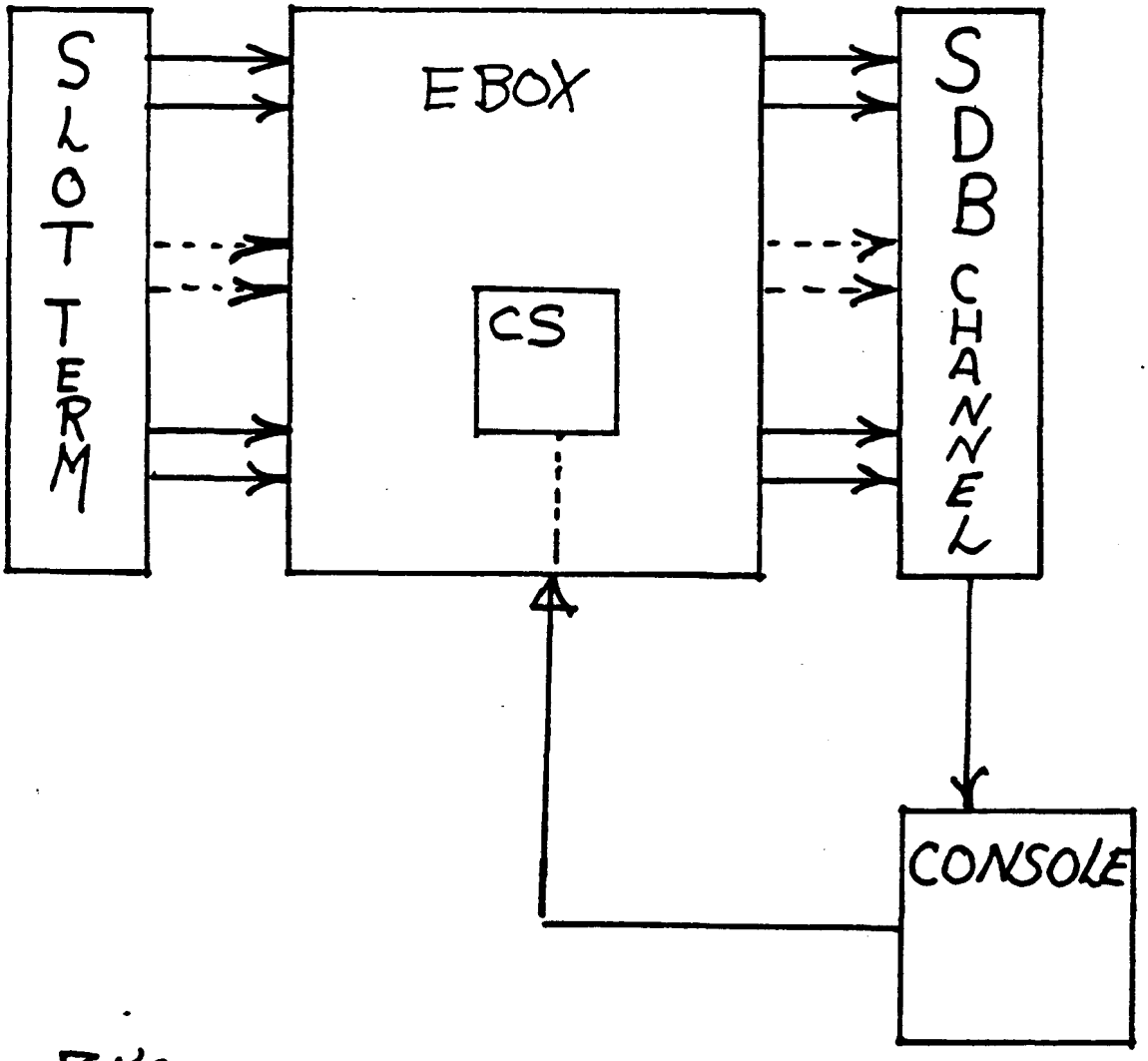
VENUS HARDWARE STATUS

EMULATORS

- 12 EMULATORS IN PROCESS
- 4 CHECKED OUT
- TURN AROUND TIME FASTER THAN MCAs AND IMPROVING

W. WALTON
6 JAN 83

EBOX STANDALONE TESTING



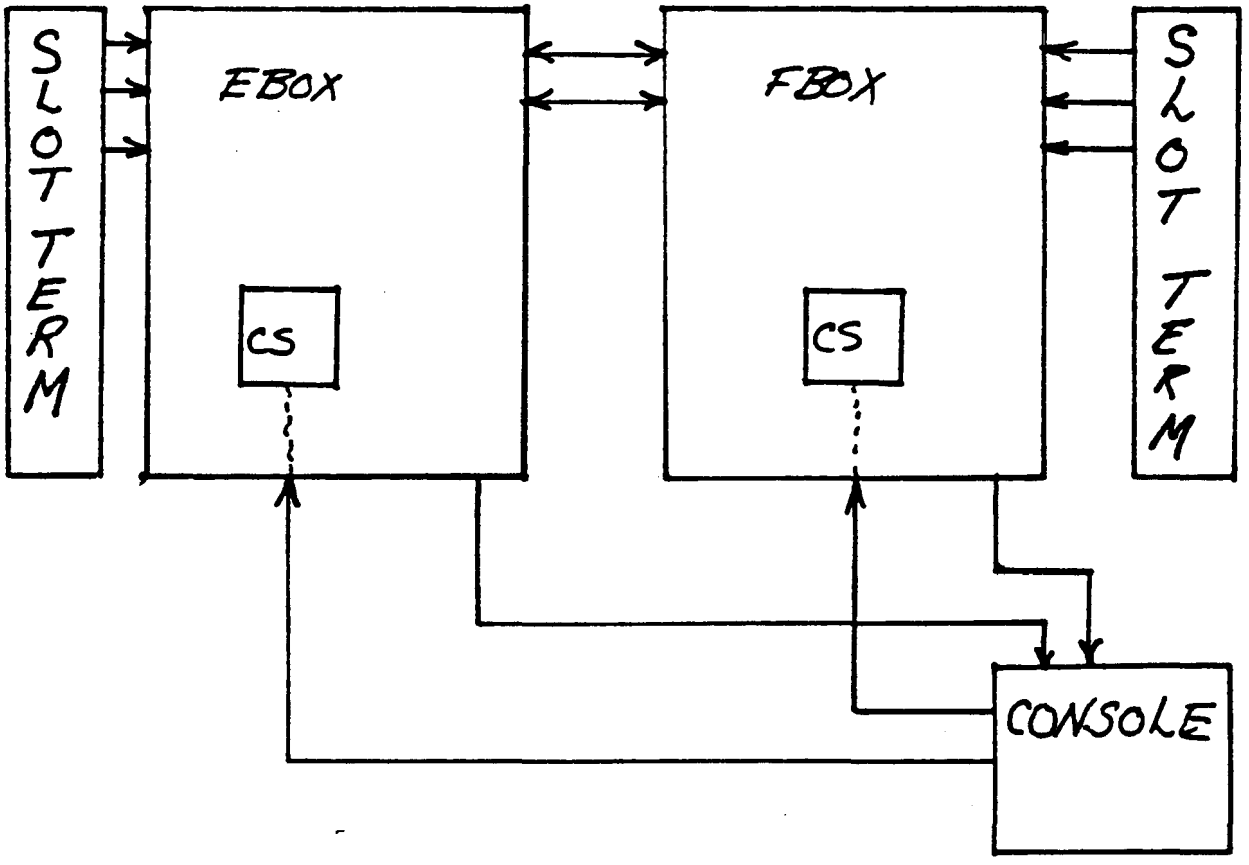
750 TESTS

> 80% DATA PATH COVERAGE

> 25% CONTROL COVERAGE

B WALTON
6 JAN 83

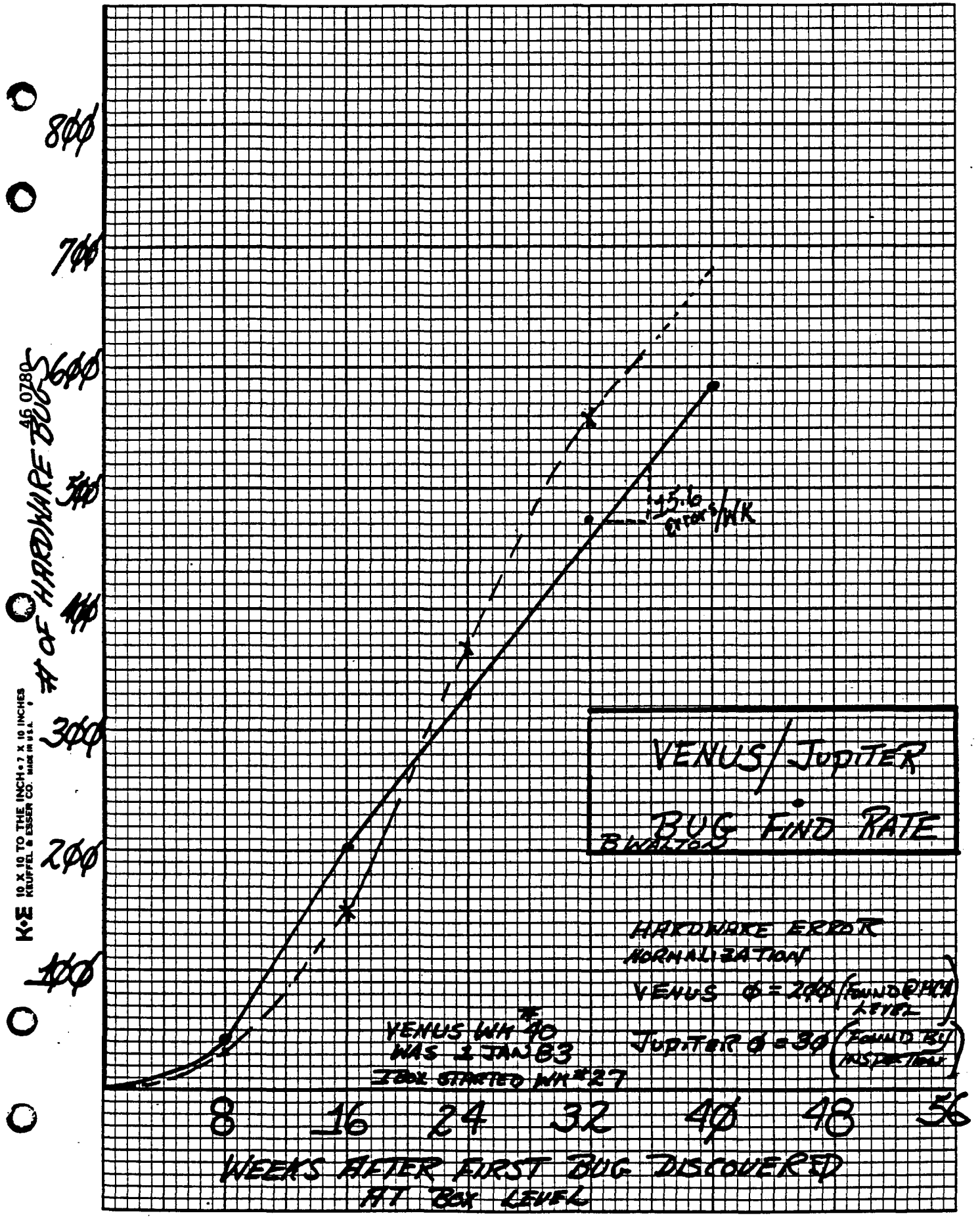
FBOX / EBOX CHECKOUT



SELF TEST
FP INSTR

> 90% COVERAGE

B WALTON
6 JAN 83


800
700
600
500
400
300
200
100
OF HARDWARE BUGS 46 0780
K&E 10 X 10 TO THE INCH • 7 X 10 INCHES KEUFFEL & ESSER CO. MADE IN U.S.A.
VENUS/JUPITER
BUG FIND RATE
BI WARTOG
HARDWARE ERROR NORMALIZATION
VENUS Φ = 200 (FOUND BY HW ATTEL)
JUPITER Φ = 30 (FOUND BY INSPECTION)
VENUS WK #40 WAS 1 JAN 83
IBOL STARTED WK #27
15.6 ERRORS/WK
8
16
24
32
40
48
56
WEEKS AFTER FIRST BUG DISCOVERED AT IBOL LEVEL

COMPUTER SUPPORT FOR VENUS - DAVE COPELAND

ACCOMPLISHMENTS

- 3 ADDITIONAL SYSTEMS INSTALLED
- DIAL-UP SWITCH INSTALLATION
- NETWORK RE-CONFIGURATION AND HOOK UP TO 782
- CONSISTENT STABLE MONITOR RELEASES ON ALL KL'S
- HELP LINE PROCEDURE IMPROVED

PROBLEMS

- CONTINUED PROBLEMS WITH KL RELIABILITY (EMILY) (Appears to be solved)
- NETWORKS NOT UP ON SAGE MACHINES
- DIFFICULTY IN GETTING SOFTWARE SUPPORT STAFF

RISKS

- UNKNOWN VAX CAPACITY REQUIREMENTS
- POTENTIAL RESOURCES DRAIN TO INSTALL AND SUPPORT JUPITER SIMULATION MACHINES AND SYSTEMS INTEGRATION LAB MACHINES

STATUS OF MANUFACTURING BUILD PROCESS - BOB MURPHY

MANUFACTUREING PROCESS :

ACCOMPLISHMENTS :-

* S/O TESTER - OPERATIONAL.
* UNIVERSAL FIXTURE - OPERATIONAL.
* DYNAPERT EQUIPMENT - ACHEIVED GOAL
* UNIBUS EXPANSION CABINETS : 15 PER DAY
  90% F.L.Y
* ALL CABLES FOR T8D AT 98% GOODNESS
* MMAIS PROGRAM OPERATIONAL
* HIGH VOLUME PROCESS : DEFINED. IN SIGN OFF
* AIDED ENGINEERING BUILD P1 & P2
* TRAINING : PERSONNEL ASSIGNED
* ALL CUSTOM TESTER IN DESIGN
* P.C.O PROCESS IN-PLACE : HDWR
* INITIAL DOCUMENT SIGN OFF HDWR & CABLES
* CRITICAL PARTS PLAN PUBLISHED
* INCOMING ESTABLISHING : ZERO BASED TESTING
  : PURCHASE HI-REL

MANUFACTURING PROCESS

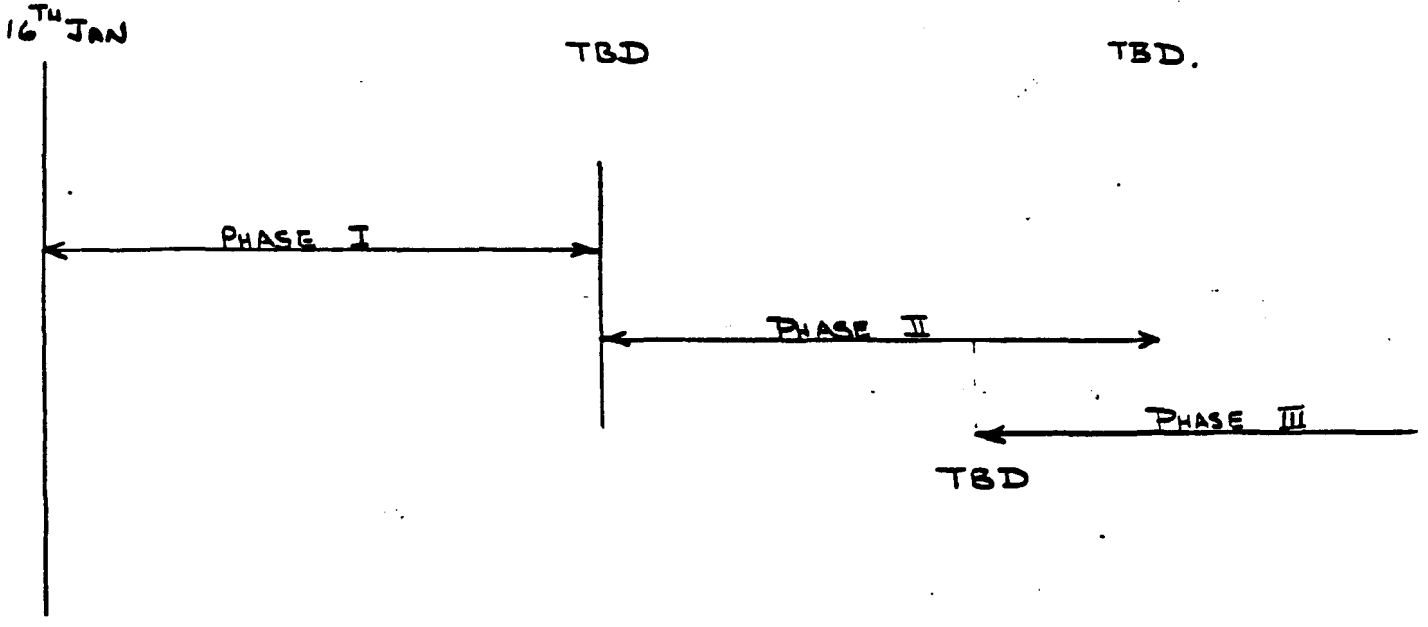
NEXT QUARTER :

* TEST PLAN SIGN OFF -
* PROTOTYPE BUILD AREA COMPLETE 16TH JAN 83
* START VENUS BUILD 17TH JAN 83
* COMPLETE DYNAPERT ACCEPTANCE
* INTRODUCE "LOG POINT" SEMI AUTOMATIC INSERTOR.
* ESTABLISH REV CONTROL THROUGHOUT PROCESS
* MONITOR "DHSFT" PROGRAM.
* INTRODUCE "MRP" SYSTEM
* FINALIZE DOCUMENTATION RELEASE PROCEDURE
* IMPLEMENT CRITICAL PARTS PLAN

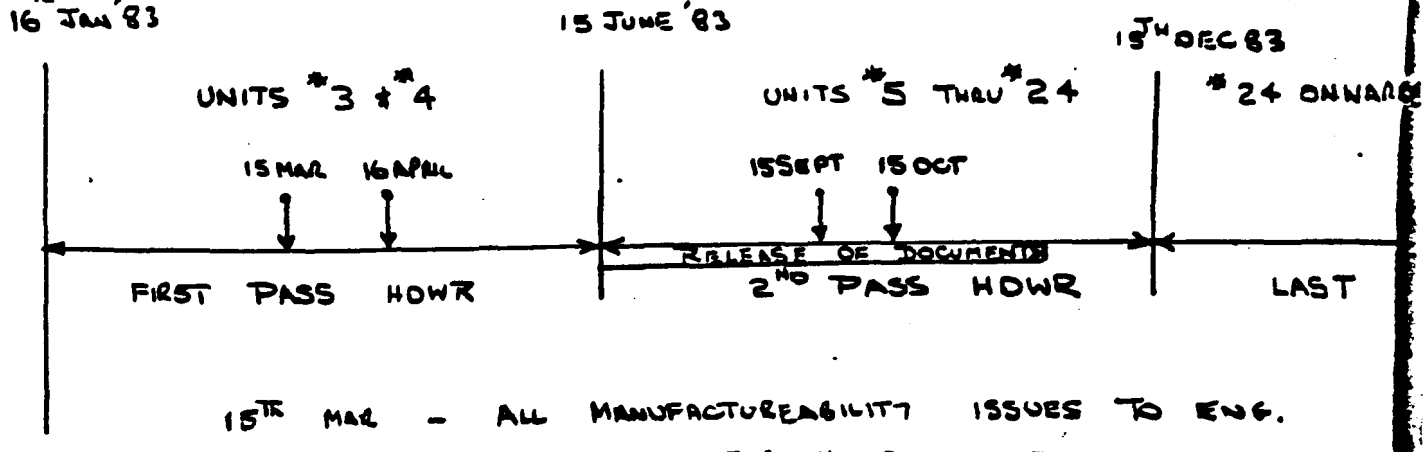
GOALS :-

* INTRODUCE I.C.T. BY JUNE.
* TEST ALL EQUIPMENT BEFORE RAMP
* CONFIGURE TO CUSTOMER ORDER (STD COST) JUNE '83
* BUILD & TEST MACHINES # 3 & 4

HARDWARE SCHEDULE



* PHASE I : BUILD & TEST MACHINES *3 & *4
* PHASE II : BUILD & TEST MACHINES *3A & *5 THRU *24
* PHASE III : PASS OFF AND TRANSITION : PROTO TO PRODUCTION



15TH MAR - ALL MANUFACTUREABILITY ISSUES TO ENG.
15TH APRIL - ALL NEW FAB HDWR IN PROCUREMENT.
15TH SEPT - 2ND PASS MANUFACTURABILITY ISSUES TO ENG.
15TH OCT - 2ND PASS NEW FAB HDWR IN PROCUREMENT.

digital

INTEROFFICE MEMORANDUM

TO: Gordon Bell ✓

DATE: November 26, 1979
FROM: Dan Hamel
DEPT: Materials
EXT: 4554
LOC/MAIL STOP: HL

NOV 28 1979

SUBJECT: 100K ECL Vendor Base

If we are seriously reconsidering the use of the Motorola Macro Cell Array in the Venus Program, it is important to note that one alternative, namely 100K ECL, is inadequately sourced to support a program of the anticipated magnitude of this one.

At present, only Fairchild plans to introduce a complete family of 100K MSI devices while RTC (France) has announced introduction schedules for only 75% of the Fairchild devices. Additionally, 100K Memory sourcing looks even more dismal with only Fairchild currently in the sampling mode while RTC's and Hitachi's target for sample introduction is June 1981 and January 1981 respectively.

The projected demand for 100K ECL to be used in our system 2080 will position us in the neighborhood of 10-20% of the industry demand. If we add to this volume the anticipated Venus demand, we could find ourselves consuming greater than 50% of forecasted industry output. This demand which would be of necessity primarily put on an historically inconsistent volume supplier, Fairchild, would be at best, very risky! This fact coupled with the unclear long term posture and direction as result of the Schlumberger acquisition should make the sourcing dilemma a major consideration to be weighed heavily in any design reconsideration.

DH/eom: 78

c.c. ULF. Fagerquist
George Hoff
Rod Schmidt
Jim Cudmore
Henry Crouse
Bill Green

++++++++++++++++
!D I G I T A L!
++++++++++++++++

INTEROFFICE MEMORANDUM

NOV 26 1979

TO: Ulf Fagerquist

CC: OOD
Venus Staff
Bob Armstrong
Carl Gibson
Per Hjerppe
Alan Kotok
Bill McBride
Pat Sullivan
Sultan Zia

DATE: 15 November 1979
FROM: George Uhff
DEPT: LSG Technology/VAX Eng.
LOC: MR1-2/F47
EXT: 231-6524

SUBJ: The Venus Technology Issue and Gordon's Response to Pat Sullivan's Memo

The reason the MCA technology is proposed for Venus is that it provides the only available solution to meet the Venus product goals. The data presented to you and Gordon on November 5, indicated that the MCA technology provides a 30% functionality versus loaded cost advantage versus the next best alternative which is 100K. The loaded cost includes PCB terminators, module power, assembly and test. In addition, we estimate that reduction in interconnect delays will improve CPU speed by 10-20%. This totals a 40-50% performance versus cost advantage versus 100K. The 40-50% figure applies to the CPU and FPA module set with power.

This data is based on test designs of Venus structures in both technologies and the latest cost information for both technologies. Pat Sullivan's packaging proposal does not present any new data. The true density achievable with this approach, is no greater than that used in the tradeoff we made.

I do not agree with Gordon's assessment that 100K will yield a much lower cost product. The time to market risk is a real problem that is a result of the gate array design process.

This nets out to the question of whether a 40-50% performance versus cost advantage, is worth the risk of six months to one year slip in time to market. I believe, with the learning available from Comet, the risks can be controlled. Another factor, is that the restart time for 100K partitioning may result in no real savings in time to market.

I also believe we must pursue gate array technology to survive in the long run. In the medium and large system markets, we must compete with IBM technology. The 4331 and 4341 are a clear indication of where they are heading. I believe the Venus follow on product must provide a 3:1 gain versus Venus just as Venus is targeted to provide 3:1 plus versus the 11/780.

I do not believe we can accomplish this with microprocessors, PLAs, bit slices or similar off the shelf approaches. We will require specialized parts (probably higher density arrays), with multiple dies on carriers. The MCA technology offers the opportunity to take a big step in advancing our design process and manufacturing process in the direction we must go to survive. If we regress to a MSI Venus, we will be faced with a double jump in technology in the future which will not be viable. Gordon's memo concerning our posture versus TI, IBM and the Japanese, noted our weakness in the IC technology area. The Motorola agreement offers the opportunity to take a significant step in the Bi Polar area. Future realities may dictate more self sufficiency in the IC area and we should not casually discard this opportunity.

I understand the ball is in my court! I will be prepared in December, to clearly lay out the alternatives and what the implications are. In addition, I intend to present a list of things we learned from discussions with the Comet designers and how we would minimize risks in Venus. The alternatives presented will include a design approach which supports an FCS Q4/82 at the lowest risk.

To: Rollins, Sparrow, Iss, Fell

digital — INTEROFFICE MEMO

Cc This page?

From: Tryggve Fossum
Loc: MR1-2/E47
Phone:231-6285
Date: 10-Dec-82

Tryggve

To: RAD I came away frustated with this because

Subject: VENUS Summary

Tnis memo tries to give the information about VENUS requested by the RAD committee. More material will be provided during the presentation.

1.0 INTRODUCTORY.

1. The targeted system performance for Venus is 4 X 780. There is reason to believe it will be somewhat better.

2. Venus will have an ABUS, with one or two SBI adaptors.The operating system will be VMS, version 3b.

3. Transfer cost for a minimal CPU with one SBI adaptor will be 33K.

4. Transfer cost for a CI based system with 1 HSC50, 3 RA81, 1 TA78, 4MB memory, etc., will be 88K.

5. Transfer cost for FPA will be 3.3K.

6. FRS June 1984.

I wanted more info. What can we do toward ~~doing a~~ getting a much better understanding of design complexity so that we can estimate how long the next design will take?

Can we talk about this?

Gordon

This understanding is the key to bounding design time + cost!!

2.0 DEVELOPMENT ISSUES.

2.1 Physical Configuration.

Double width cabinet for CPU, containing 17 CPU modules, 8 memory modules of 4MB each. All modules are L type, with 8 layers, and 4 signal layers. There is room for two ABUS adaptors of two modules each. In addition there is an I/O backpanel with 21 module slots.

There is also a single width front-end cabinet with console load device (RL02), and Unibus communication and unit record controllers.

2.2 Logical Configuration.

Ebox, Mbox, Ibox, Fbox are four microcoded processing units, providing some execution overlap. The base machine runs at a 72 ns cycle time, while the Fbox runs at 36 ns per cycle. There is also a T11-based Console with RL02 load device.

2.2.1 Mbox -

The Mbox contains 512 locations TB, 16K bytes Cache, interface to ABUS, interface to Main Memory, and interfaces to Ibox and Ebox. Reads and Writes take one cycle. There is an Address Module, a Data Module, and a Control Module.

2.2.2 Ibox -

The Ibox consists of the Ibuffer unit for instruction prefetching and the OPFETCH unit for operand fetching and storing. There is a 8 byte prefetch buffer used for I-stream and string fetching. There is a Decode RAM for instruction decoding of up to 512 opcodes, with up to 8 operands each. The Ibox has its own copy of the GPRs for address generation and operand fetching. The Ibox decodes instructions and specifiers, and delivers operands to the Ebox and Fbox over the OPBUS. It receives result data from these boxes over the WBUS. Register conflicts are detected and scoreboarded in the Ibox. There is a register log for tracking changes to the GPRs due to specifier evaluations.

The Ibox keeps track of past, present, and future values of the PC, so that instructions can be backed up and continued appropriately.

2.2.3 Ebox -

The Ebox is a heavily microcoded unit, which executes the instructions, and controls operations of the whole system. It has a dual ported RAM for storing GPRs, constants, temporaries, and architectural data. During each cycle, two operands can be read, operated upon, and stored back into the RAM. The Ebox runs the memory management functions, and assists the Fbox with floating point instructions. It processes hardware errors, and handles interrupts and exceptions.

2.2.4 Fbox -

The Fbox has a 32 bit data path, and runs at half the cycle time of the base machine. It has a 32 X 8 multiplier. It receives operands from the Ibox over the OPBUS, and returns results over the WBUS. It completes ADDF in 2 base machine cycles. When not executing floating point instructions, it runs self testing diagnostics.

2.3 Complexity Measures.

The basic design unit is the MCA, an ECL gate array with 60 signal pins, about 70 cells, 5 W max power. Each MCA contains about 1000 gates. Each module can contain 20-30 MCAs, depending on number of 10K parts. Some modules have only 10K parts. Each module has about 230 signal pins. The ECL RAMS used are 256x4 and 1Kx4.

A good complexity measure is number of MCA types. Designing and processing MCAs seems to take up most of the time. There are 16 types in the Ibox, 13 in the Ebox, 18 in the Mbox, and 18 in the Fbox.

2.4 Documentation.

Documentation is done at the box level. In the Fbox there is a functional spec for the whole box, and specs for each individual MCA. There are also block diagrams for datapath and for control, at for the box and for individual MCAs.

There is also a spec for the Ebox microcode, describing structure and algorithms.

2.5 Design Tools.

The fundamental CAD tool is SUDS. It provides inputs to CALDEC, IDEA, MCACUT, MODCUT, and SAGE. SAGE is used for gate level simulation, which is being done at chip, module, box, and CPU levels. To fit it all in, the MCAs go tthough a functional compression program. For timing verification, the AUTODLY program is being developed. There are also automatic test pattern generators, like AXE and LASAR. Functional TUMS models are used for initial verification.

3.0 SPECIAL ISSUES.

3.1 Difficult Problems.

Most difficult problem: Designing with Gate Arrays.

In the Fbox, we seem to always run into limitations of space and power on the modules, and power, pins, and cells for the MCAs. If we had been less aggressive in goals for cost and/or performance, this might have been simpler. Inadequate CAD software made the task of debugging and layout more difficult. Turnaround time in modules and gate arrays is going to hurt us once we start hardware debug.

The fact that the Venus project is taking so long, creates problems with good engineers leaving the project (and often the company). The growthpaths have been inadequate. It is hard to create and sustain enthusiasm.

The complexity of the VAX architecture is a problem in the way that it aggravates pin, cell, and power problems unless you plan for it. Some of us did not always plan well.

I wish we had handled branches better. That may be our biggest performance problem. There is room for improvement without changing the architecture or introducing excessive design complexity.

The Venus design has a lot of special cases. These may increase the performance, but create problems in verification, debugging, and design time.

3.2 What Increased The Performance.

1. Cycle time is almost 1/3 of 780 cycle time.
2. Main Memory access time is almost 1/3 of 780.
3. Processing Overlap between the Ibox and the Ebox.
4. Optimization, i.e. processing two operands in one cycle.
5. Fewer micro-cycles per instruction in the Ebox and Fbox.
6. Pipelined memory references, make it possible to make a Cache reference every cycle.

7. Write Back Cache speeds up back-to-back writes.

8. Cache valid bit per longword.

3.3 What Decreased The Performance.

1. The instruction overlap is not efficiently used when the instruction sequence is interrupted.

2. Not all instructions are optimized that should have been.

3. The Mbox does not do as much parallel processing as it might have.

4. There is no logic to detect memory interlocks, creating an Ibox stall for every memory write.

5. The Bus structure may not be optimal.

6. Compatibility Mode is neither in nor out, resulting in a costly (in microcode) low performance implementation.

NOV 27 1979

digital

INTEROFFICE MEMORANDUM

TO: Gordon Bell

cc: George Hoff
Dan Hamel
Jim Cudmore
Ulf Fagerquist

DATE: November 26, 1979
FROM: Bill Green
DEPT: LSI
EXT: 4482
LOC/MAIL STOP: ML1-4/B34

SUBJECT:

In regard to the attached memo of yours, I have asked Dan to author a memo on the supply situation for 100K ECL. We believe that there does not exist now, nor will there exist in the future, a source of supply adequate to support Venus. Widespread industry interest has not arisen to stimulate availability. The 2080 is of sufficiently low volume as to be no problem, apparently.

If Venus does not go MCA some other solution than 100K should be sought.

WBG:cg
Attachment

00 BURT DECGRAM ACCEPTED S 3182 0 03 08-DEC-79 11:33:00

****
* d i g i t a l *
****

TO: HENRY CROUSE @CLEM

cc: ULF FAGERQUIST
JIM CUDMORE @CLEM

DATE: SAT 8 DEC 1979 11:28 AM EST
FROM: GORDON BELL
DEPT: OOD
EXT: 223-2236
LOC/MAIL STOP: ML12-1 A51

SUBJECT: MOTOROLA MCA VERSUS FAIRCHILD 100K

Henry, I don't like what I am getting from the collective DEC organizations in Manufacturing and Engineering.

Intuitively I am skeptical that we can pull off the deal with Motorola in any reasonable time scale and any reasonable cost, and development cost. Also, I am less positive that gate arrays are the way to go in the future for us in all but a few cases. We just may have missed our window in time. This is based on turn-around, our equipment to design with, the interface with manufacturing, the fact that we will be stretching an already thin mfg organization into ecl gate arrays, when the really big pressures will be on better mos, mos capacity, bipolar capacity,better bipolar gate arrays, etc. I think IBM is the only one capable of really pulling gate arrays off. Also, I think there is a narrow time window we missed on the technology that will pass when we get denser chips and the gate array concept may cave in (Craig Mudge is adamant about this and pointed this out to me).

Your organzation strongly supports not doing business with Fairchild, which I understand, due to their poor record...and there is no second source. On the other hand, I think we could get a second sorurce more easily then we can make the MCA work... as it hasn't met any of its goals so far. Also, we have to bring mca in. No one has been able to make me understand why the 2nd source can't be found. I really worry about doing business because it feels good with the people (ISS felt good too), but alternatively doing business with incompetents isn't good either (Pertec, fifairchild).

We are going full blast ahead with mca, but I sure couldn't feel worse.
PS

The engineering groups in lsi (green, cudmore) and under ulf are absolutely determined do to the technical challenge.

Can you offer any way to help? to convince me? to get a second source?

digital

INTEROFFICE MEMORANDUM

cc → Demmer

NOV 12 1979

TO: George Hoff - MR1-2/E47
Bill McBride- MR1-2/E85

DATE: 7 NOV 79
FROM: Pat Sullivan PS
DEPT: L.S.E.G.
EXT: 6234
LOC/MAIL STOP: MR1-2/E85

NOV 14 1979

Given the MCA review in VENUS, I support using 100K (for VENUS) as giving us a much lower cost, quicker time to market product. The MCA decision doesn't feel right. Gordon

SUBJ: 100K PACKAGING

I have a new proposal for an extended hex PC board arrangement that will accomodate 180-24 pin 100K logic IC's as well as 60-24 pin terminator dip packages, all of which are machine insertable. I believe this proposal will reduce 2080 manufacturing costs and development time and therefore is worth considering for the VENUS project which could substantially benefit by reducing time to FCS.

Basically, the proposal is to have 240-24 pin IC positions which are fixed in 8 rows of 30 positions each. Each of the 240 IC positions on the module accepts either a 100K IC or a terminator package. The terminator package is a 24 pin dip which contains 18 terminator resistor, a-5V bypass capacitor and -2V capacitor(s). The advantages of this system are:

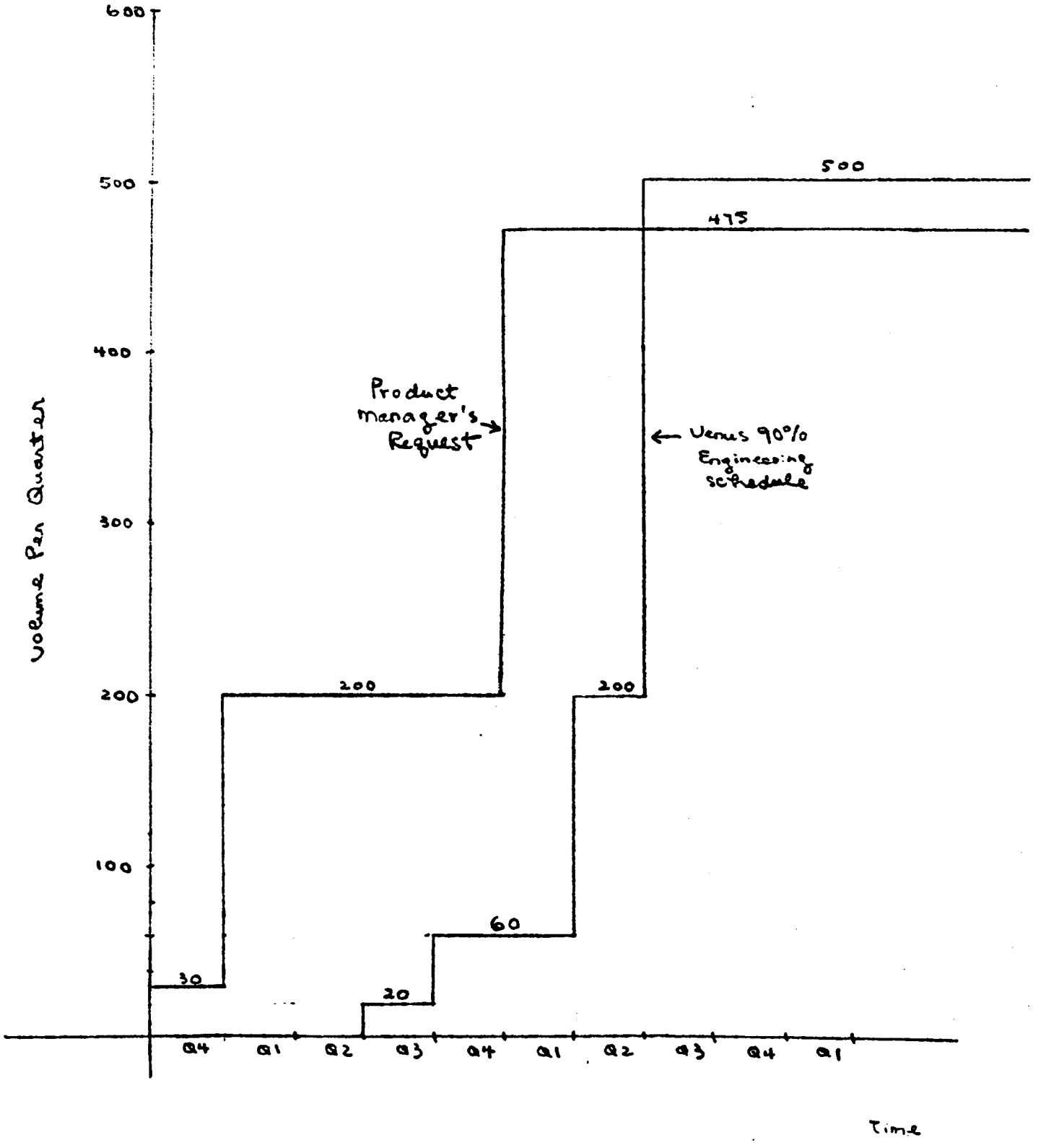
1. All parts on board are machine inserted except for 8-10 storage capacitors.
2. Use standard inner layers for all PC boards.
3. One test pattern for PC board bed of nails tester.
4. All component locations are interchangeable which should make the auto PC placement program (pincut) work more efficiently.
5. Any unused location are predrilled powered which greatly simplifies ECO installation.

In addition to the above, I believe there are other major advantages in designing with 100K. Foremost is the elimination of all of the MCA design constraints, design aids processing, and MCA manufacturing. Secondly 100K implementation is more forgiving in respect to correcting errors (etch cut and wire add as opposed to several month wait for new MCA).

The only rationale for Venus/MCA is we think we need the technology (Cadmore) and layout for subsequent technologies ... and will pay for it. Gordon

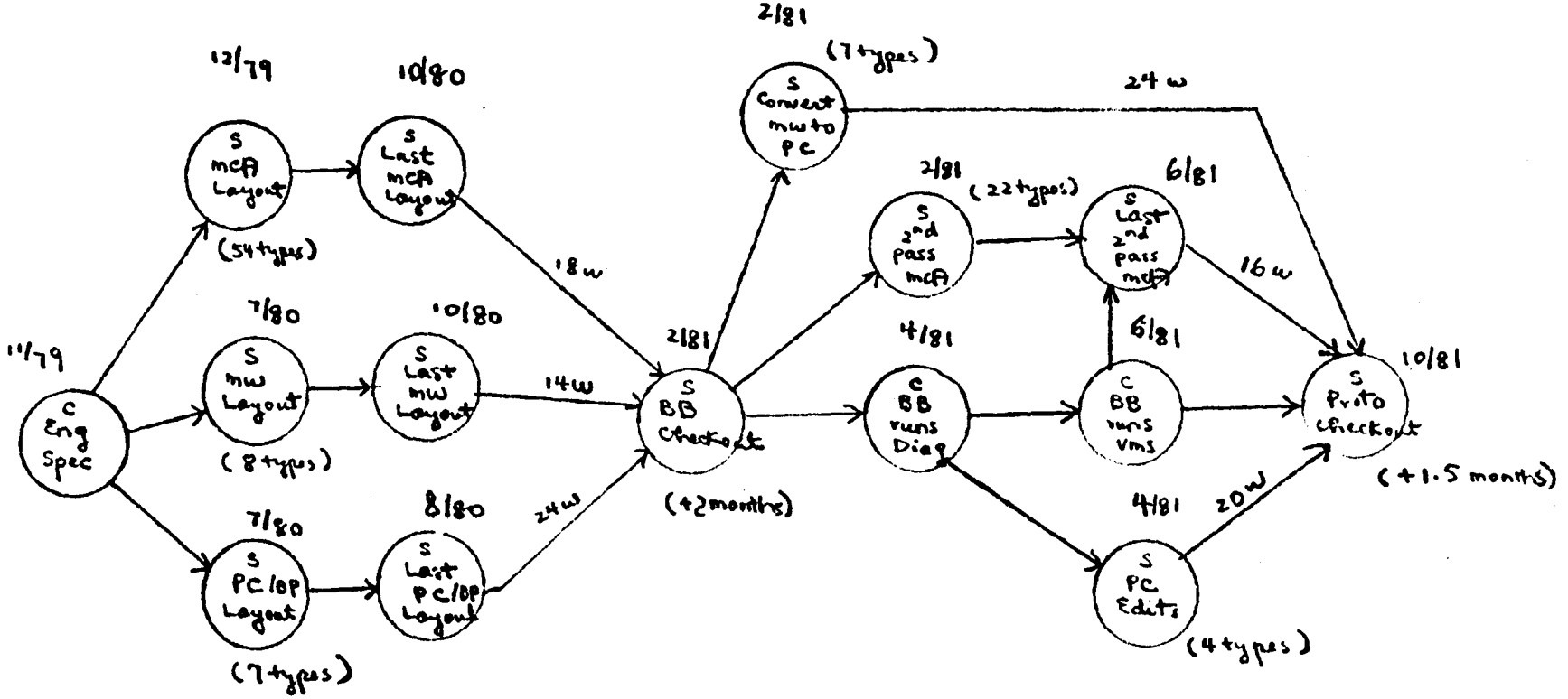
bjb

CC: Gordon Bell
Nat Kerllenevich
Jud Leonard



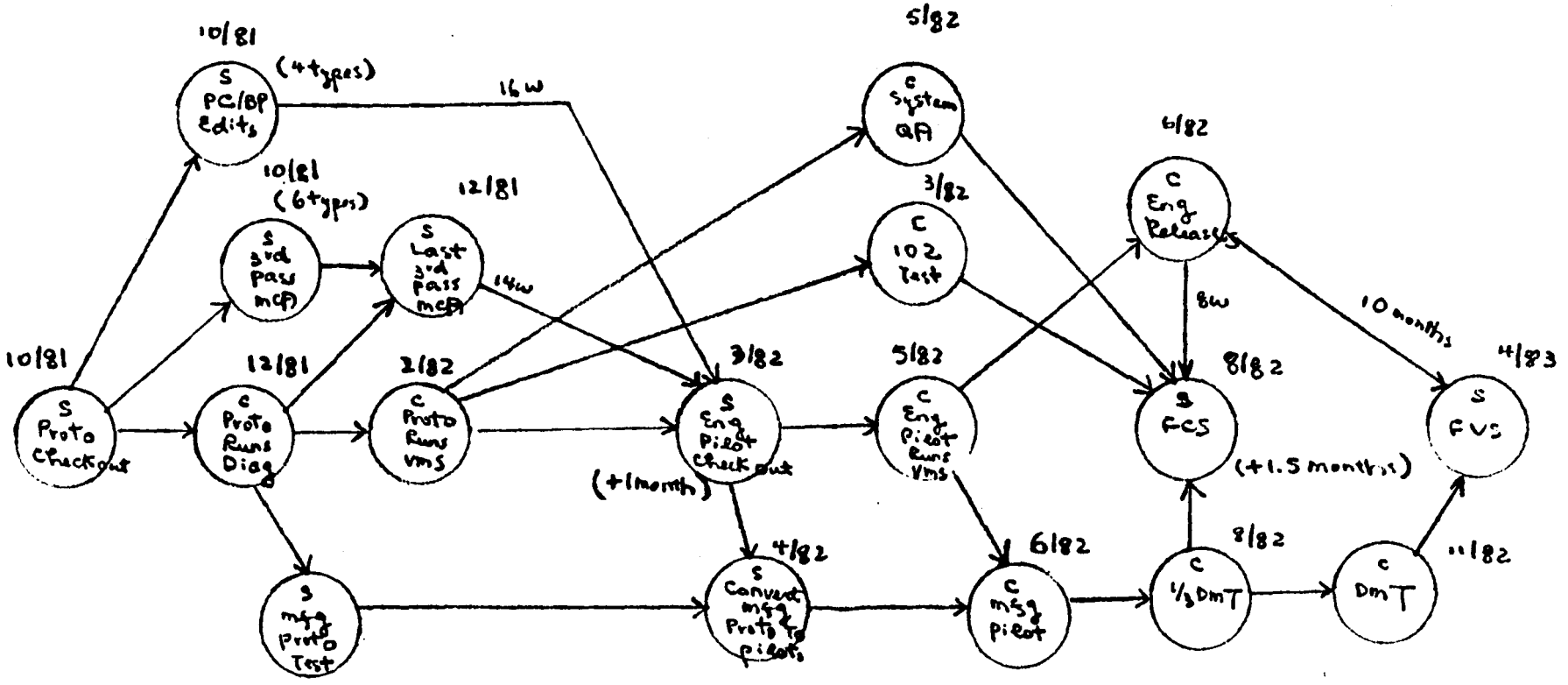
Venus System Shipping Rate

Vic Ku
1/31/80



Venus Overview Part 1

Vic Kyu
1/3/79



Venus Overview Part 2

Vic Ku
1/3/79

DISTRIBUTION LIST

GEORGE HOFF	MR1-2/E47
SULTAN ZIA	MR1-2/E47
VIC KU	MR1-2/E47
JUD LEONARD	MR1-2/E47
BILL WALTON	MR1-2/E47
PETE LAWRENCE	MR1-2/E47
ROGER LAWSON	MR1-2/E18
CARL GIBSON	MR1-2/E73
DALE COOK	MR1-2/E63
JOHN GROSS	MR1-3/174
MIKE ROBEY	MR1-1/905
CHUCK SAMUELSON	FW/D03
GORDON BELL	ML12-1/A51
SAM FULLER	ML3-5/H33
ALAN KOTOK	ML3-5/H33
JOHN HOLZ	ML5-2/E57
DUANE DICKHUT	ML1-2/E55
AL RYDER	MK1-1/D29
DAVE RODGERS	FW/C04
BOB STEWART	FW/C04
KEN OKIN	FW/C03
BILL STRECKER	FW/A03
STEVE ROTHMAN	FW/D06
DAVE CANE	FW/D06
TOM EGGERS	FW/B05
DICK HUBVEDT	FW/D03
DON HOOPER	MR1-2/E85

****
* d i g i t a l *
****

TO: GORDON BELL*
ULF FAGERQUIST
cc: HENRY CROUSE @CLEM
JIM CUDMORE @CLEM

DATE: WED 12 DEC 1979 1:41 PM EST
FROM: GEORGE HOFF
DEPT: LS ENG OPR
EXT: 231-6524
LOC/MAIL STOP: MR1-2 E78

SUBJECT: YOUR MEMO ON MCA VS. 100K

I believe we can do an MCA Venus at reasonable cost in a reasonable time frame. We have "not to exceed" pricing in the contract that is now significantly lower in cost than 100K which is going up at every quote. Motorola has the volume facilities in place to back up their volume commitments. We presently, have more qualification data collected on MCA than 100K. The part is living up to expectations in terms of design return. We have auto placement running and have characterized layout schedules with several DOLPHIN parts. The layout risk in VENUS is PCBs not MCAs.

The MCA has not met all of its goals to date, but many have been met. All macros functional, IIL very close, qualification parts delivered within one day of schedule, top priority in wafer processing, qualified resources on board to resolve outstanding issues. Daily communication on status and data, capacity commitment being fulfilled, outside second secong source (National) agreement is signed up.

It is true that the MCA VENUS will put an additional burden on Manufacturing. I believe they need this product to drive the DL content of our products down, if we are to survive. We must reduce parts counts, reduce module counts, improve the sophistication of our test methods at the chip level/module level and system level. The IIL approach offers an opportunity to advance our test methods in several areas.

I believe that the MCA offers an opportunity to advance our BiPolar process. The technology transfer and the opportunity to build production parts, is a solid opportunity to move ahead in the BiPolar area. MOS will not solve the VENUS follow on requirement and will probably not satisfy the COMET follow on either.

Many of our competitors in addition to IBM, are either in production with gate arrays or in the process of qualification. Motorola currently, has 13 customers somewhere in the start up process with MCAs.

I do not agree with your assessment about the narrow window for gate arrays, particularly, in the VENUS and up range of products. Craig has a lot of challenging concepts, however, I believe the custom design process must be developed and matured on simple systems before it can be moved up into more complex designs. I will write a separate mamo on this subject.

I do support your request for a thorough understanding of the ECL capacity alternatives. The primary goal I am dealing with on the VENUS program, is "time to volume". The turn around of vendor

Pr[illegible]ions on volume commitments over the past year has been dramatic - I feel a strong shift to a seller's market.

I believe we can get solid commitments from Motorola if we close the contract in January. We will explore what commitments we can get from Fairchild with bigger numbers. Behind Motorola, I could have Hudson and National. All we have behind Fairchild is RTC.

****
* d i g i t a l *
****

TO: GORDON BELL*

cc: ULF FAGERQUIST

DATE: WED 12 DEC 1979 4:47 PM EST
FROM: GEORGE HOFF
DEPT: LS ENG OPR
EXT: 231-6524
LOC/MAIL STOP: MR1-2 E78

SUBJECT: VENUS STATUS

Schedule

Our PERT is presently five months over our 50% date - Q4/FY82. We are presently focusing on changes to plan to pull this back in. The major risk element is people and computers in the design process - I think we can significantly reduce schedule and risks with additional

ULF FAGERQUIST PER HJERPPE GEORGE HOFF

"CC" DISTRIBUTION:

LARRY PORTNER BERNIE LACROUTE ANDY KNOWLES
BILL DEMMER SI LYLE

****
* d i g i t a l *
****

TO: see "TO" DISTRIBUTION

cc: see "CC" DISTRIBUTION

[redacted] DATE: SUN 2 DEC 1979 10:58 AM EST
[redacted] FROM: GORDON BELL
[redacted] DEPT: OOD
[redacted] EXT: 223-2236
[redacted] LOC/MAIL STOP: ML12-1 A51

SUBJECT: GETTING THE RIGHT DEVELOPMENT PRIORITIES SET FOR VENUS

Like the 2080, I am concerned that the development group from top to bottom have the same set of development goals. In talking with Jud, it is clear that this is not the case. In many respects, I think the priorities should be about the same as the 2080, but possibly interchanging entry cost and performance.

To begin with, we need to write down a bunch of definitions as to just what the parameters are we have any control over. I assume these: t.development; perf., availability, t.repair, reliability, $.dev correlates with t.development; $.entry; $.ownership.
After defining these, we need to define the interrelationships among them and get them crisp.

Next, we have to get a standard set of assumptions about the market projection. Mine are roughly: 780 fcs may 78, 780 fvs sept 78, rate at about 100/month and 200K each until venus. I assume a linear ramp up in 6 mos from nov 82 till may 83, and then a 48 month run and then a 12 month gradual ramp down when venus replacement is announced in nov. 86...giving a life of roughly that of 780. I also assume any delay in this will reduce 780 revenues and the venus plan will be shorter by the appropriate amount. This set of numbers, whatever (including the size distributions) should be used in the following analysis.

Now, we enter the plan into BURP and begin to analyze some partial derivatives: d($.product cost)/d(t.development)---is my favorite. Namely using some really gross numbers of say $200K/machine, and 100 machines/month for 50 months, we get 20m (or 10m of cost/month) for sales (nor), and $1 b over machine's life. I assume if the machine slips a month, we lose 20m and are spared 10m in cost. Note Note the product contribution is the same, but the roi has gone way down becaause the 50m to bring to market is constant and probably increases as we slip, giving us a double whammy. In order to hold roi (by not slipping, I believe we can spe spend up to 10m for all the 5000 machines we will build for every month we might slip...or a hit of 2K/machine. It isn't intuitively obvious how these all inter act, but the issue is the analysis of these critical partial derivatives to give a constant roi.

The second partial of interest is d($.product cost)/d($.ownership). Namely how much can we add to product cost to reduce cost of ownership, while having const. or increasing roi to the corp. Here, we have to get the service content model i into burp (which is there if we use it). Issues like ecc on the cache, and various busses should pop out of the work.

Somehow I haven't seen any of the taskforce work on venus, and I want to. We have to get these critical issues resolved so our developers know which way to go. I would note that the straightforward 780 implementation at a relatively high product cost really paid off in time to market and reliability. I don't want another comet where we optimize the hell out fo cost and lose all.

"TO" DISTRIBUTION:

ALL listed

digital

INTEROFFICE MEMORANDUM

DEC 3 1979

TO: Gordon Bell

DATE: November 26, 1979
FROM: Dan Hamel
DEPT: Materials
EXT: 4554
LOC/MAIL STOP: HL

SUBJECT: 100K ECL Vendor Base

If we are seriously reconsidering the use of the Motorola Macro Cell Array in the Venus Program, it is important to note that one alternative, namely 100K ECL, is inadequately sourced to support a program of the anticipated magnitude of this one.

At present, only Fairchild plans to introduce a complete family of 100K MSI devices while RTC (France) has announced introduction schedules for only 75% of the Fairchild devices. Additionally, 100K Memory sourcing looks even more dismal with only Fairchild currently in the sampling mode while RTC's and Hitachi's target for sample introduction is June 1981 and January 1981 respectively.

The projected demand for 100K ECL to be used in our system 2080 will position us in the neighborhood of 10-20% of the industry demand. If we add to this volume the anticipated Venus demand, we could find ourselves consuming greater than 50% of forecasted industry output. This demand which would be of necessity primarily put on an historically inconsistent volume supplier, Fairchild, would be at best, very risky! This fact coupled with the unclear long term posture and direction as result of the Schlumberger acquisition should make the sourcing dilemma a major consideration to be weighed heavily in any design reconsideration.

DH/eom: 78

c.c. ULF. Fagerquist
George Hoff
Rod Schmidt
Jim Cudmore
Henry Crouse
Bill Green

```
  AAA          BBBB    U   U   SSS
 A   A         B   B   U   U  S   S
 A   A         B   B   U   U  S
 AAAAA         BBBB    U   U   SSS
 A   A         B   B   U   U      S
 A   A         B   B   U   U  S   S
 A   A         BBBB     UUU    SSS

DDDD    EEEEE   SSS    CCC   RRRR   III PPPP   TTTTT III   OOO   N   N
D   D   E      S   S  C   C  R   R   I  P   P    T    I   O   O  N   N
D   D   E      S      C      R   R   I  P   P    T    I   O   O  NN  N
D   D   EEE     SSS   C      RRRR    I  PPPP     T    I   O   O  N N N
D   D   E          S  C      R R     I  P        T    I   O   O  N  NN
D   D   E      S   S  C   C  R  R    I  P        T    I   O   O  N   N
DDDD    EEEEE   SSS    CCC   R   R III P        T   III   OOO   N   N
```

****

1.0 INTPODUCTION

The I/O Adapter Bus (A Bus) provides an information path between the Venus MBOX and its I/O adapters and supports a communication protocol which allows data transfers between the Venus CPU and adapters and between the adapters and Venus memory. The A Bus allows the attachment of up to 4 I/O Adapters to a Venus CPU. Thus far, the need for adapters to the SBI, BI, and perhaps CI or DR32 interface has been defined, and other adapters or devices may be connected to the A Bus in the future. The interconnections between the I/O Adapters and the Venus CPU are shown in figure 1.

1.1 A Bus Signals and Basic Protocol

The A Bus consists of 70 ECL level signals which interconnect the I/O Adapters, MBOX, and E/IBOX. There are 40 bi-directional parallel signals between the MBOX and I/O Adapters over which command, address, and data are passed under control of 7 other parallel signals and 16 radially distributed signals. Also associated with the A Bus are 7 parallel signals between the EBOX and I/O Adapters which are used to convey information concerning the levels of pending interrupt requests from the I/O Adapters.

The information transport mechanism involves the use of

register files in each I/O Adapter which receive and drive the bi-directional signal lines, namely ADDRESS/DATA <31:00>, COMMAND/MASK <3:0>, DATA LENGTH/STATUS <1:0>, ADDRESS/DATA PARITY, and CONTROL PARITY. Basically, the transfer of information to the MBOX from I/O Adapter n is effected by the adapter loading the information into locations in the register file and signalling the presence of such information over one of the radially distributed request lines (IOA REQUEST n) to which the MBOX responds by asserting a grant signal (IOA SELECT n) which causes the I/O Adapter to present the contents of the appropriate register file locations over the bi-directional signal lines. The MBOX transfers information to an I/O Adapter by loading it into register file locations over the bi-directional signal lines and signalling the completion of the transfer with the assertion of the DONE n line, which induces the I/O Adapter to access the register file to obtain the information.

Interrupt service requests are communicated to the EBOX through a scanning sequence in which each I/O Adapter is polled for the highest pending interrupt request which is presented to the EBOX over the IPR RETURN lines. The adapter being polled during an A Bus cycle is encoded on the IPR ADDRESS lines. The EBOX honors an interrupt request when priorities permit by reading an I/O register internal to the I/O Adapter via the A Bus and MBOX data paths in order to obtain an interrupt vector which is used to dispatch to the interrupt service routine.

The E/IBOX provides clocks to the I/O Adapters which drive logic that is synchronous with respect to the MBOX.

1.2 Transaction Buffers

The register file locations are organized into logical groups called transaction buffers. One possible buffering scheme (that proposed for the BI Adapter) is shown in figure 2. The 3 sets of register file locations are used to provide independent buffering of CPU initiated bus operations via the "P" designated buffers and I/O Adapter initiated DMA operations via the "A" and "B" designated buffers. Double buffering is provided for DMA operations to increase throughput. Write operations require a command address (CA) longword followed by 1 to 4 longwords of write data (WD1 - WD4). 4 bits of mask information are provided per longword to identify the bytes to be written. Read operations require a command address (CA) longword and a buffer in the reverse direction to hold the 1 to 4 longwords of read data (RD1 - RD4).

Other buffering arrangements than that described above

are permitted. Depending on the requirements of the I/O Adapter, DMA buffers may support 1, 2, or 4 longwords of read or write data. The A Bus protocol does not limit the number of DMA buffers.

1.3 Synchronization

The register files and the bus signals used to control them provide a well defined point of synchronization between the MBOX and the I/O Adapters. The devices used to implement the register file have independent A Bus and Internal IOA Bus ports so that an adapter can be performing operations on locations associated with one transaction buffer while the MBOX is simultaneously operating on other locations. The signalling scheme for passing control of a transaction buffer from the MBOX to I/O Adapter n and vice versa is accomplished with the IOA REQUEST, IOA SELECT, CLEAR REQUEST, and DONE signals which are synchronous with respect to the MBOX. All A Bus signals sourced by the I/O Adapters must be synchronized by the adapters. Similarly, interrupt requests are synchronized with respect to the EBOX by the I/O Adapters before being passed over the A Bus.

1.4 Arbitration

The MBOX controls use of the bi-directional signal lines as outlined above, thus acting as the A Bus master. Arbitration of the bus is effected by the MBOX determining the sequence in which it services I/O Adapters. Highest priority is assigned to address Ø, with each consecutive address having lower priority than its predecessor. SBI and BI Adapters are assigned addresses Ø and 1 to minimize read access latency and maximize bandwidth to SBI and BI devices, and the remaining two addresses are allocated for other devices.

The order in which transaction buffers within an I/O Adapter are serviced is determined by the adapter.

2.Ø PHYSICAL ADDRESS SPACE PARTITIONING

Only physical addressing is supported by the A Bus; it has no virtual addressing capability. DMA references by I/O Adapters to Venus main memory specify physical addresses, and processor references to I/O Adapters are passed over the A Bus as physical addresses. (This does not preclude adapters such as the CI Adapter from internally manipulating virtual memory addresses and using them to reference memory after performing virtual

to physical address translations.)

2.1 Physical Address Allocation

The allocation of the physical address space for processor references is a function of the MBOX. Each 1 megabyte block of physical address space (both memory and I/O space) can be independently assigned to either Venus main memory or any one of the I/O adapters. This will be accomplished by an MBOX configuration PAM with 1024 locations, one for each block, which is loaded at configuration time and thereafter directs processor requests to the appropriate destination. Further translation of processor accesses to I/O address space is required in BI and SBI adapters to allow their co-existence on the same A Bus. Any registers in the I/O Adapters (control, status, error, etc.) which must be made available to the Venus processor will appear in I/O space.

2.2 Physical Memory Address Space

The memory space may be occupied by main memory and memory on the SBI and/or BI which co-exist in the same machine. Adapters which support memory on the buses to which they interface must distinguish requests intended for local memory from those intended for Venus main memory. The allocation scheme is shown in figure 3.

Venus main memory is assigned memory addresses starting at location 0 and is contiguously allocated in 1 megabyte blocks to its upper bound. Memory on the SBI or BI starts at an address specified by a register in the adapter referred to as the Memory Separator Register. The value in the Memory Separator Pegister is constrained to a multiple of 1 megabyte and a range of 0 through 2000 0000 hexadecimal.

Each SBI or BI Adapter contains a Memory Separator Register so that it can determine the destination of DMA operations initiated by devices on the SBI or BI. References to addresses below that specified by the Memory Separator Pegister must be routed through the adapter to the MBOX and ultimately Venus memory, and references to all other locations must be permitted to be accepted by the appropriate device on the SBI or BI. The Memory Separator Registers in all adapters assume the value of the first location in the first 1 megabyte block of memory address space not occupied by Venus memory.

If more than one I/O Adapter interfaces to memory, each memory in the Venus system is assigned unique, non-overlapping, contiguous segments of memory. It is

expected that physical memory will be configured to minimize the size of gaps of unused memory address space between memories. This will allow VMS to map the memory address space as efficiently as possible by minimizing the size of the PFN data base, a VMS data structure.

2.3 I/O Address Space

Venus I/O space is partitioned into 16 segments of 32 megabytes each, as shown in figure 4. Segments Ø through 3 are assigned to I/O Adapters Ø through 3, respectively. The remaining segments do not carry fixed assignments to adapters. This scheme is compatible with the current 11/78Ø SBI implementation and BI specification.

Each BI or SBI Adapter requires only one I/O segment. Bits <28:25> of an I/O address are redundant since their value was already used by the MBOX to direct the CPU access to a particular I/O Adapter. If the address does not correspond to a register internal to the adapter, the transaction is repeated on the BI or SBI with bits <28:25> forced to zero.

Other types of I/O Adapters which require more than 32 megabytes of I/O space are not precluded from Venus. The MBOX configuration RAM allows segments 4 through F to be assigned independently to any of the adapters.

3.Ø INTERNAL I/O ADAPTER REGISTERS

I/O Adapters will contain some internal registers for the passage of configuration, control, status, error, and interrupt vector information between the adapters and EBOX. Locations with byte offsets <8 ØØØØ:F FFFF> hexadecimal from the beginning of the first 4 32-megabyte I/O segments are reserved for internal I/O Adapter registers.

Some of these registers are common to all adapters and are specified with respect to address, format, and content. These locations are reserved in the I/O address spaces of the BI and SBI, so there is no conflict with assigning the locations to registers internal to the adapters. CPU accesses to locations assigned to internal adapter registers are processed internal to the adapter; in the case of the BI and SBI Adapters, other accesses are repeated on the BI or SBI. The register assignments are listed below and shown in figure 5.

<8 0000:8 0003> Configuration Register (Read Only)
<8 0004:8 0007> Control/Status Register (Read/Write)
<8 0008:8 000B> Error Register 1 (Read/Write)
<8 000C:8 007F> (Adapter Specific Registers)
<8 0080:8 00BB> Interrupt Vector Registers (Read Only)
<8 00BC:F FFFF> (Adapter Specific Registers)

4.0 TRANSACTION BUFFERS

All information passed over the A Bus on the 40 bi-directional signal lines is communicated between the MBOX and register file locations in the I/O Adapters. The register file locations are grouped into transaction buffers as shown in figure 2.

4.1 CPU Buffer

One buffer is allocated for CPU initiated read and write operations which are limited to a maximum of one longword of data per transaction. It is designated the "P" buffer and is often referred to as the "CPU" buffer. It contains 3 locations: (1) a "CAp" designated location for storing command/address information from the MBOX, (2) a "WDp" designated location for storing a longword of write data from the MBOX for write operations, and (3) a "RDp" designated location for storing a longword of read data to be returned to the MBOX for read operations.

4.2 DMA Buffers

The other two buffers are allocated for I/O Adapter initiated read and write operations which involve 1, 2, or 4 longwords of data per transaction. These buffers are designated the "A" and "B" buffers and are often referred to as the "DMA" buffers. They contain 9 locations: (1) A "CAa" or "CAb" designated location for storing command/address information to be passed to the MBOX, (2) four "WDa" or "WDb" designated locations for storing up to 4 longwords of write data to be passed to the MBOX for write operations, and (3) four "RDa" or "RDb" designated locations for storing up to 4 longwords of read data returned to the adapter by the MBOX for read operations.

4.3 Buffer Formats

The information passed over the bi-directional data lines is grouped into 5 fields:

1. Address/Data <31:00> field. This field is used to pass 28-bit physical longword addresses in bits <27:00> for "CA" designated locations and 4 bytes of longword aligned data for "RD" and "WR" designated locations.

2. Command/Mask <3:0> field. This field specifies the basic command (read, read lock, read modify, write, write mask, or write mask unlock) for "CA" designated locations and provides a byte mask with the data contained in "WD" designated locations which identifies which bytes are valid and should be written into the destination location.

3. Data Length/Status <1:0> field. This field augments the basic command in "CA" designated locations by specifying the size of a transaction (even word, odd word, longword, quadword, or octaword) and indicates whether good or bad data is being supplied in the "WD" or "RD" designated locations.

4. Address/Data Parity bit. This bit is used to maintain odd parity across the Address/Data field. It is carried through the register files and MBOX A Bus interface and passed over the A Bus.

5. Control Parity bit. This bit is used to maintain odd parity across the Command/Mask and Data Length/Status fields. It is carried through the register files and MBOX A Bus interface and passed over the A Bus.

4.4 Physical Implementation

The transaction buffers will be contained in register files which permit independent access from either the A Bus or I/O Adapter side. A custom chip, the DC022 16X4 Register File, is being developed by the Microproducts Group for use on Venus. The DC022 is a 16 word by 4 bit register file with two independent read/write ports which allow simultaneous, asynchronous access to the register file from both ports. Data is written into and read out of the register file in parallel via 2 sets of 4 bi-directional data lines, one set for each port. Each port has its own address inputs, read enable, and write enable. One port has 10K/MCA compatible inputs and outputs with data output drivers capable of driving 25-ohm loads. The other port has TTL compatible inputs and outputs with data output drivers capable of sinking 20 milliamperes in the low state. (The DC022 is also being used in the Console interface to the E/IBOX.)

The physical implementation of the transaction buffers depicted in figure 2 is shown in figure 6. The

simultaneous read/write capability from both ports of the DCØ22 allows the "RD" and "WP" locations within each buffer to be shared since they are not used simultaneously within a buffer. Consequently, only 12 locations are needed for the transaction buffers.

4.5 Buffer Control

Access to the register file port on the A Bus side is controlled by both the MBOX and the I/O Adapters through manipulation of a counter which drives the A Bus port address lines. The MBOX has the ability to load, increment, and decrement the counter, but the adapter provides the value to be loaded into the counter. The MBOX, therefore, does not need to know the allocation of transaction buffers within a register file (hence register assignments can be adapter specific). When the MBOX honors a request for service from an I/O Adapter, the adapter determines which transaction buffer is to be serviced by providing the appropriate address to the counter load inputs. This value is loaded into the counter under control of A Bus signals from the MBOX and the information is read across the A Bus data lines. Additional locations in the buffer are accessed by the MBOX at its own rate by incrementing the counter when it is prepared to read or write the next location. In the case of a CPU initiated transaction, the MBOX can force an I/O Adapter to load the counter with the address of the "CAp" designated location in the CPU buffer and effect a write to that location. The control of access to the register file port on the adapter side is specific to the type of adapter.

5.Ø BUS CYCLES

Information is passed between the Venus processor and I/O Adapters and between the I/O Adapters and Venus memory over the A Bus utilizing sequences of operations called transactions. An A Bus transaction uses several A Bus cycles (67 nanoseconds per cycle) to pass the command, address, and write or read data between the MBOX and an I/O Adapter. There are basically six types of bus cycles: CPU initiated command/address cycles, DMA initiated command/address cycles, write data cycles, read data return cycles, null cycles, and arbitration cycles.

5.1 Information Transfer Cycles

The CPU command/address, DMA command/address, write data, and read data return cycles allow the transport of

information between the MBOX and register files in the I/O Adapters. Each type of bus cycle involves access to specific locations in a buffer. In particular, command/address cycles involve access to "CA" designated locations, write data cycles access "WD" locations, and read data return cycles access "RD" locations. The formats of these 4 types of cycles are shown in figures 7 through 10.

5.2 Null Cycle

The null cycle occurs when the A Bus is not being driven by the MBOX or any I/O Adapters. All the bi-directional signal lines are in their unasserted state; therefore, the parity is even. Null cycles occur during and between bus transactions and are inserted at will by the MBOX to regulate the rate at which information is passed over the A Bus.

5.3 Arbitration Cycle

The arbitration cycle is in some sense of the word a phantom cycle since it is performed in parallel with the other bus cycles. During an arbitration cycle, the MBOX arbitrates among the requests from the I/O Adapters which are conveyed to the MBOX over the IOA REQUEST <3:0> lines and services the highest priority request by asserting an IOA SELECT line is the same cycle.

6.0 INTERRUPTS

The A Bus protocol provides the mechanism for the passage of interrupt request information from the I/O Adapters to the EBOX. Interrupt vectors are passed back to the EBOX over the A Bus and through the MBOX using the same data paths and protocols employed for normal CPU initiated read and write operations issued to I/O Adapters.

6.1 Interrupt Request Acquisition and Prioritization

The Venus interrupt system gives I/O Adapters access to 15 of the hardware Interrupt Priority Request (IPR) levels, namely 10 through 1E hexadecimal. The EBOX employs a hardware polling mechanism which scans all the I/O Adapters (and potentially other interrupt sources within the CPU) to ascertain the highest IPR pending for each interrupt source in order to determine which source has the highest pending IPR and the level of that IPR.

Basically, at the beginning of each A Bus cycle, the IPR Address Lines <1:0> assume the next value in the scan sequence. An adapter detecting its address on the A Bus drives the IPR Return Lines <4:0> with the highest pending IPR during the same cycle. The EBOX latches the information at the end of the cycle. While polling another interrupt source during the next A Bus cycle, the EBOX compares the latest IPR against the previous highest IPR. If the latest IPR is higher, it saves the higher IPR and its source address. If the latest IPR is lower than or equal to the previous highest IPR, it is discarded unless it is from the same source as the previous highest IPR (indicating that an interrupt request was dropped) and the EBOX saves the latest IPR for that source. Consequently, the EBOX retains knowledge of the highest IPR pending and its source.

6.2 Interrupt Vector Acquisition

When processor conditions permit (i.e. processor Interrupt Priority Level (IPL) is lower than the highest pending IPR), the highest IPR will be serviced. If it originated from an I/O Adapter, the EBOX will invoke an I/O read through the MBOX to the appropriate adapter in order to obtain an interrupt vector which is used by the EBOX to index into the System Control Block (SCB) and dispatch to the appropriate interrupt service routine. There is a group of 15 consecutive longword I/O locations associated with each I/O Adapter. Each location within a group corresponds to one of the hardware interrupt request levels (IPR <1E:10>). An I/O read to one of these locations will cause the adapter to create or obtain an interrupt vector corresponding to the level associated with the accessed location.

For IPR <17:14> the SBI and BI Adapters must perform operations on the adapted buses to obtain the information needed to complete the I/O read. In the case of the SBI, an Interrupt Summary Read (ISR) is issued on the SBI for the interrupt level corresponding to the I/O register accessed by the EBOX which identifies all SBI devices interrupting on that level; the SBI Adapter constructs a vector from this information and returns it as read data. In the case of the BI, an Identify (IDENT) command is issued on the BI for the level corresponding to the I/O location accessed by the EBOX which causes the device with highest bus priority interrupting on that level to return its interrupt vector which is passed back to the EBOX as read data.

6.3 Interrupt Release

In general, interrupt requests are dropped by a device or

I/O adapter when it passes an interrupt vector back to the EBOX or when a register in the device is written explicitly to clear the interrupt. A race condition exists in the latter case between (1) the propagation of the negation of an interrupt from the source to the microcode interface in the EBOX and (2) the completion of the interrupt service routine and the potential lowering of the processor IPL. If the IPL is lowered prematurely, the EBOX will attempt to service the same interrupt again. A mechanism will be provided to prevent the EBOX from lowering its IPL until the negation of an interrupt can propagate back to the EBOX.

APPENDIX

A.1 List of A Bus Signals

The A Bus signal mnemonics and their sources and destinations are listed below.

SIGNAL NAME <BITS>	IOA	MBOX	EBOX	E/IBOX
IOA CLOCK <3:0>	D			S*
ADDRESS/DATA <31:00>	S/D	S/D		
COMMAND/MASK <3:0>	S/D	S/D		
DATA LENGTH/STATUS <1:0>	S/D	S/D		
ADDRESS/DATA PARITY	S/D	S/D		
CONTROL PARITY	S/D	S/D		
IOA REQUEST <3:0>	S	D*		
CLEAR REQUEST	D	S		
CPU BUFFER	S	D		
CPU WAIT	S	D		
IOA SELECT <3:0>	D	S*		
ADDRESS CONTROL <1:0>	D	S		
MBOX OUT	D	S		
MEMORY INTERLOCK	S/D	S/D		
DONE <3:0>	D	S*		
IPR RETURN <4:0>	S		D	
IPR ADDRESS <1:0>	D		S	

S = Source
D = Destination
* = Radially distributed, one per IOA

Total Signals To/From IOA: 58
Total Signals To/From MBOX: 59
Total Signals To/From EBOX: 7
Total Signals From CONSOLE: 4
Total A Bus Signals: 70

A.2 BI Adapter Bandwidth

Figures 11 and 12 are timing diagrams for DMA read and write data transport between the BI and Venus memory. The diagrams represent the maximum bandwidth which can be achieved by a BI Adapter which occurs under the following conditions:

1. All requests are for octawords (16 bytes).
2. More than one device is contending for the BI such that the imbedded arbitration is always won by some device other than the current master.
3. There is no contention for the MBOX or Venus memory by other I/O Adapters or the Venus CPU.
4. There is no interference from memory refresh cycles.
5. The memory being accessed is not on a repeater.
6. The BI bus cycle is 200 nanoseconds.
7. The A Bus cycle is 66.7 nanoseconds.

The octaword read bandwidth is 7.27 megabytes/second and the octaword write bandwidth is 13.3 megabytes/second.

J LACY
25-FEB-80

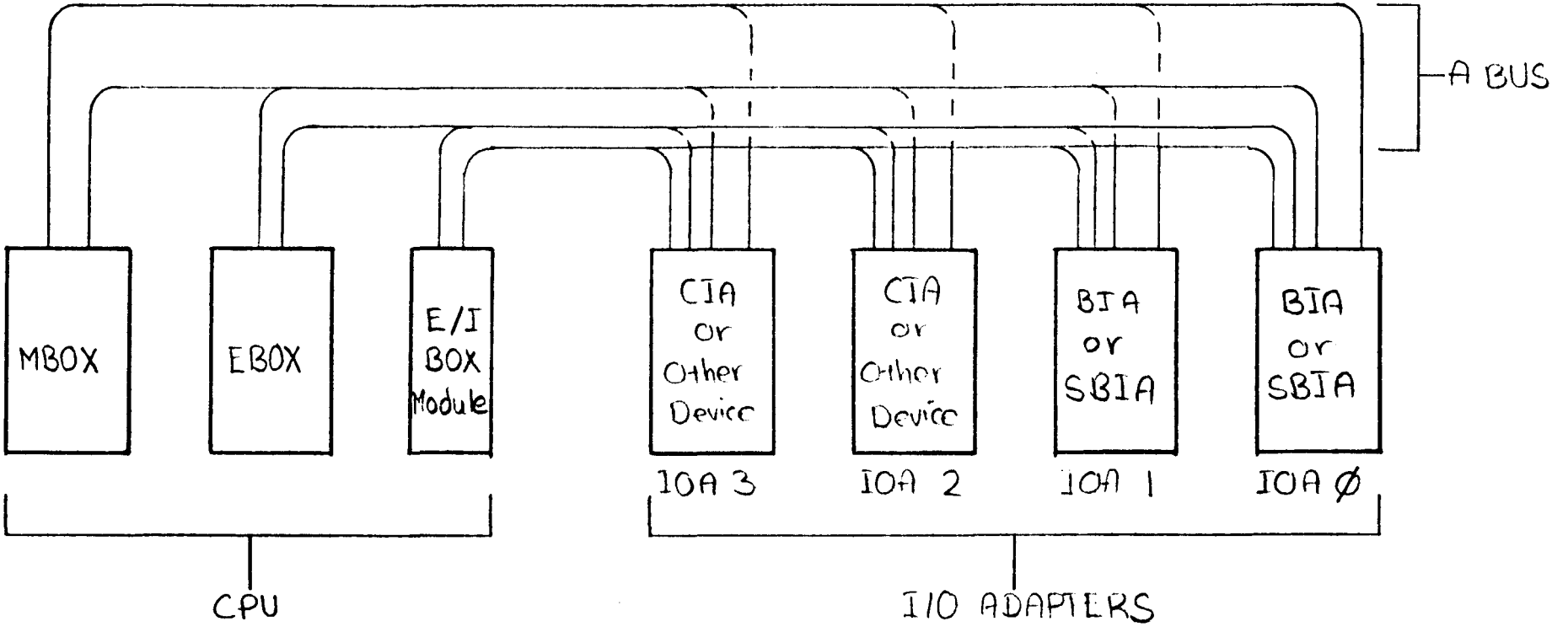


FIGURE 1 : SIMPLIFIED A BUS BLOCK DIAGRAM

J. LACY

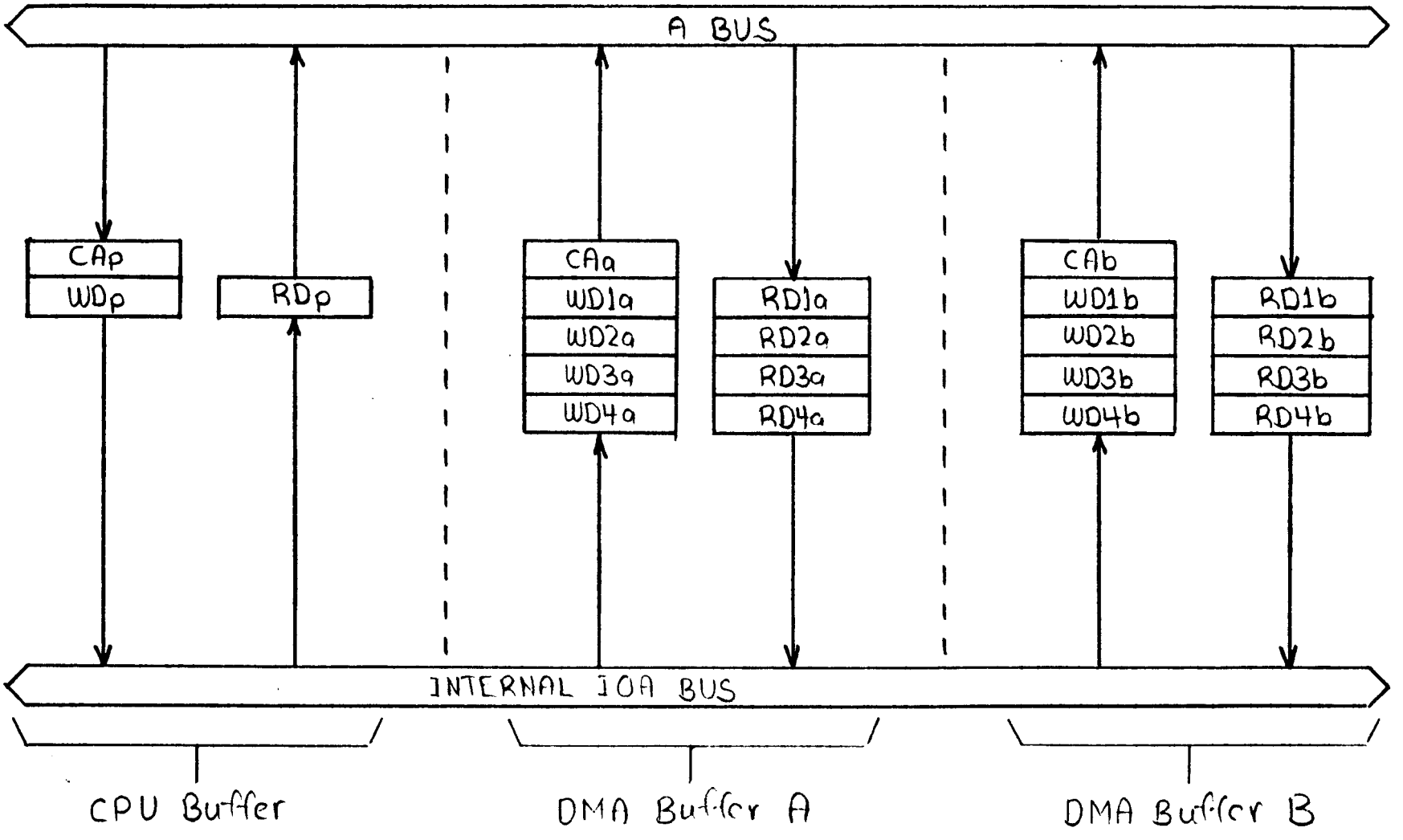


FIGURE 2: A BUS TRANSACTION BUFFERING

J. LACY
25-FEB-82

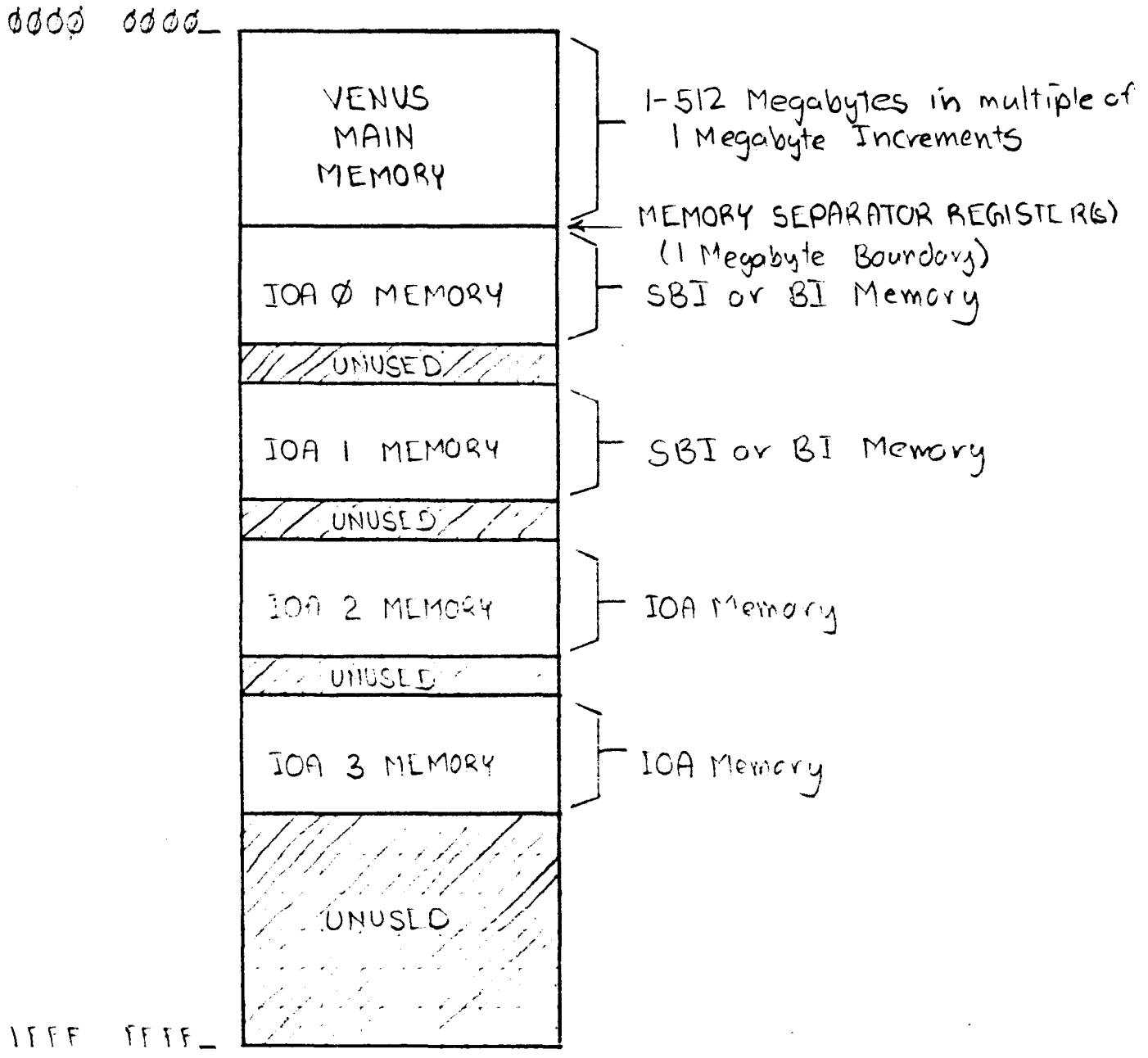


FIGURE 3:

PHYSICAL MEMORY ADDRESS SPACE ALLOCATION

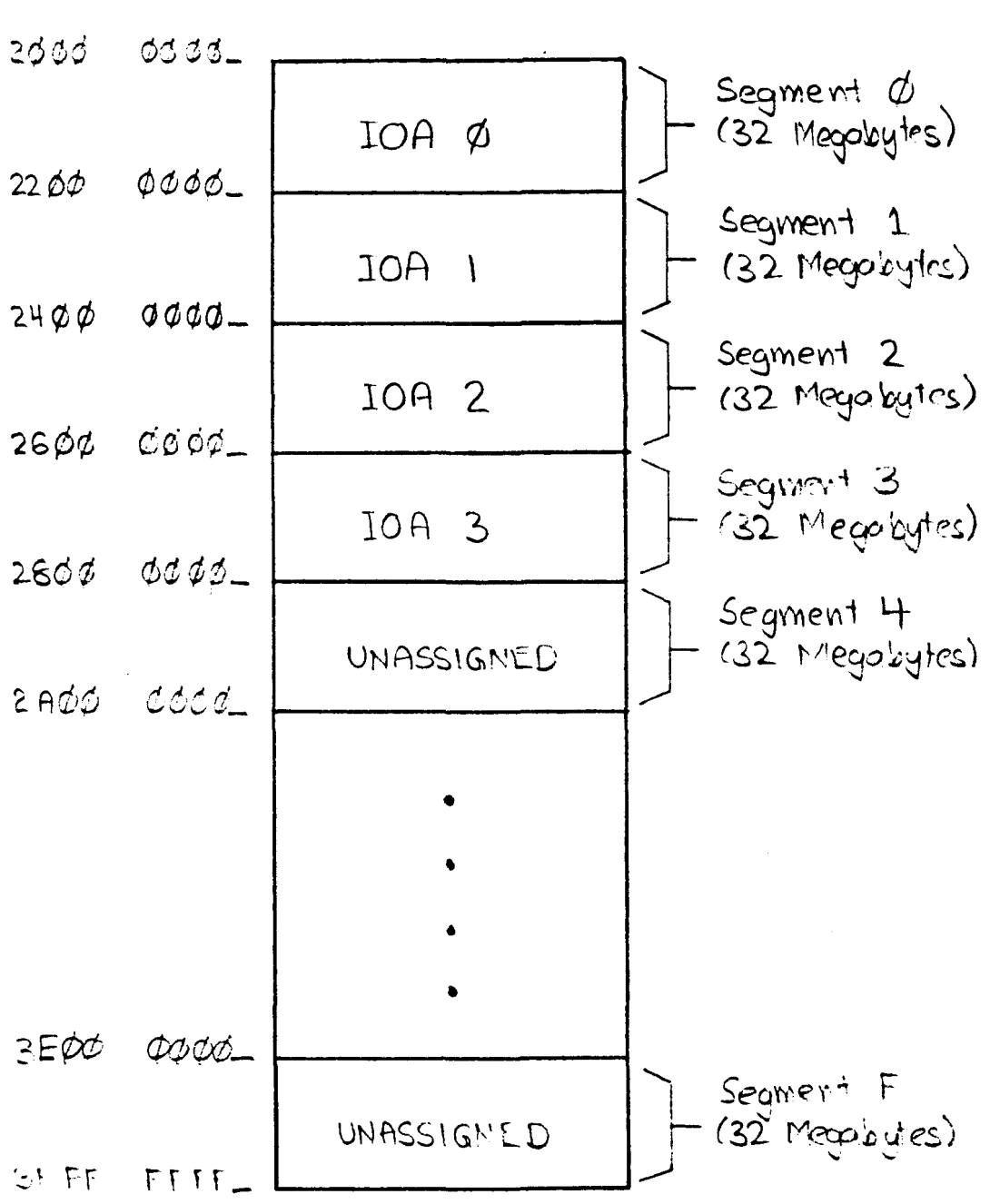


FIGURE 4:

I/O ADDRESS SPACE ALLOCATION

J. LACY

Hex Byte Adr

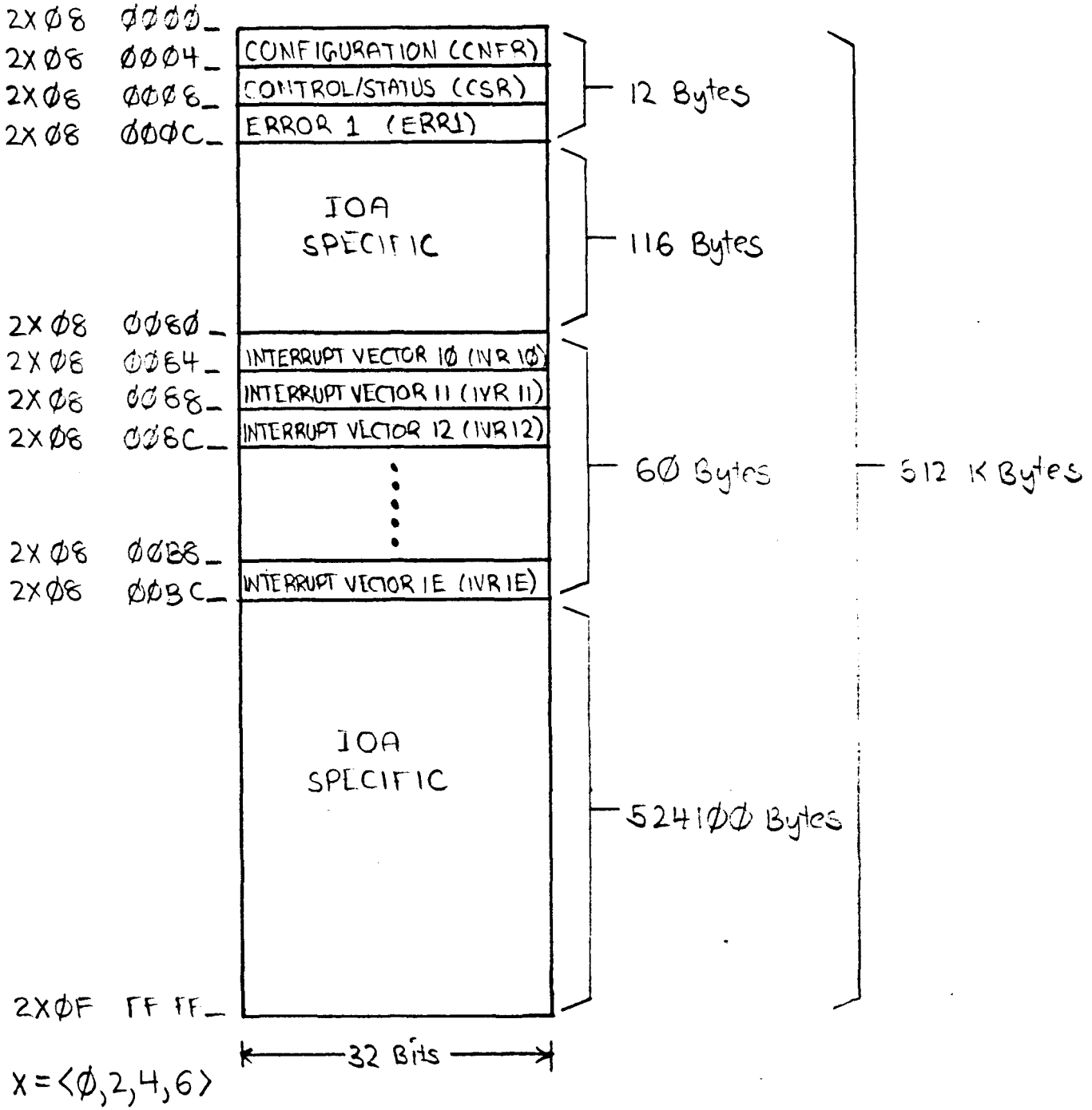


FIGURE 5:

INTERNAL I/O ADAPTER REGISTER ASSIGNMENTS

J. LACY

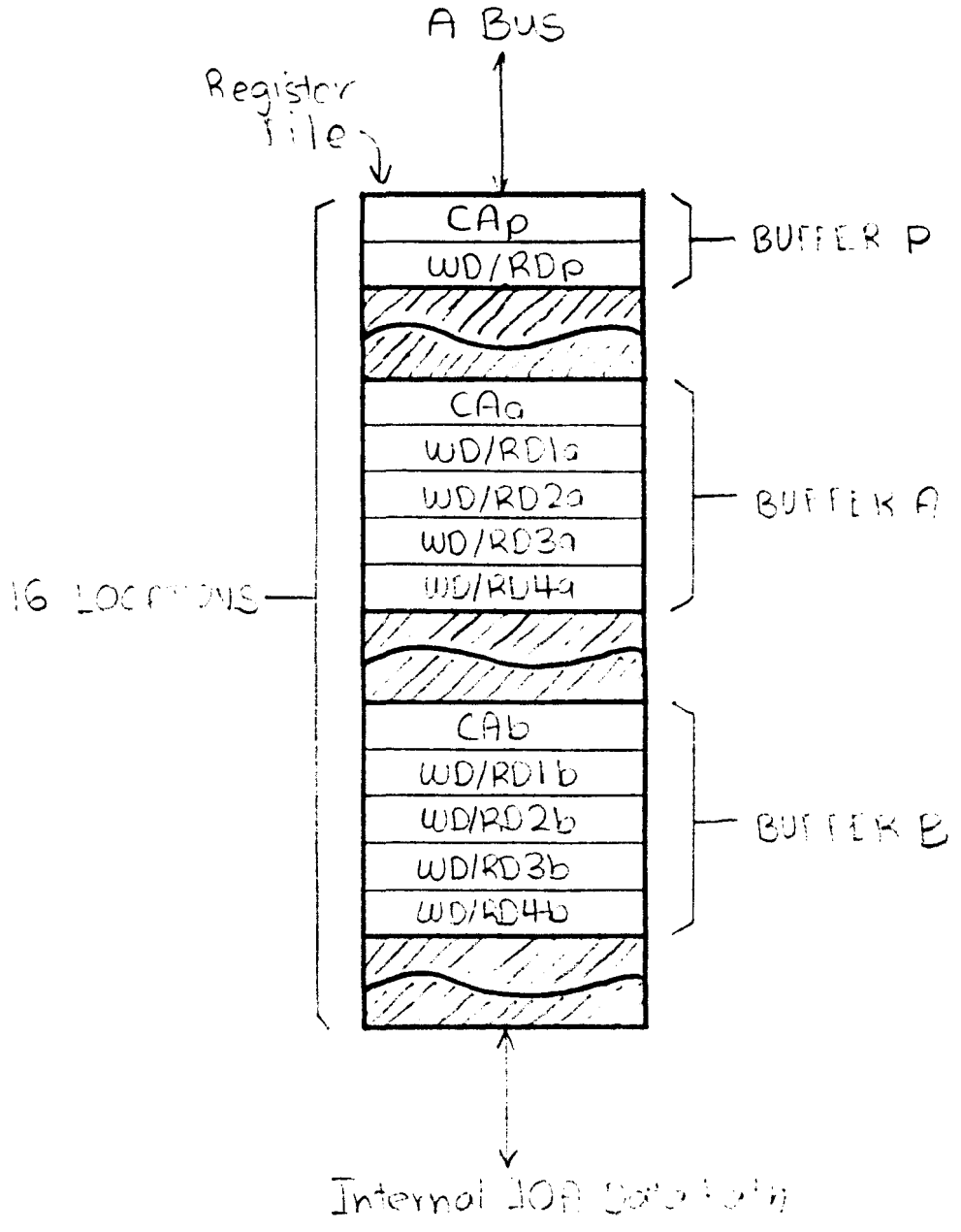


FIGURE 6:

PHYSICAL IMPLEMENTATION OF TRANSACTION BUFFERS

J LACY
25-FEB-83

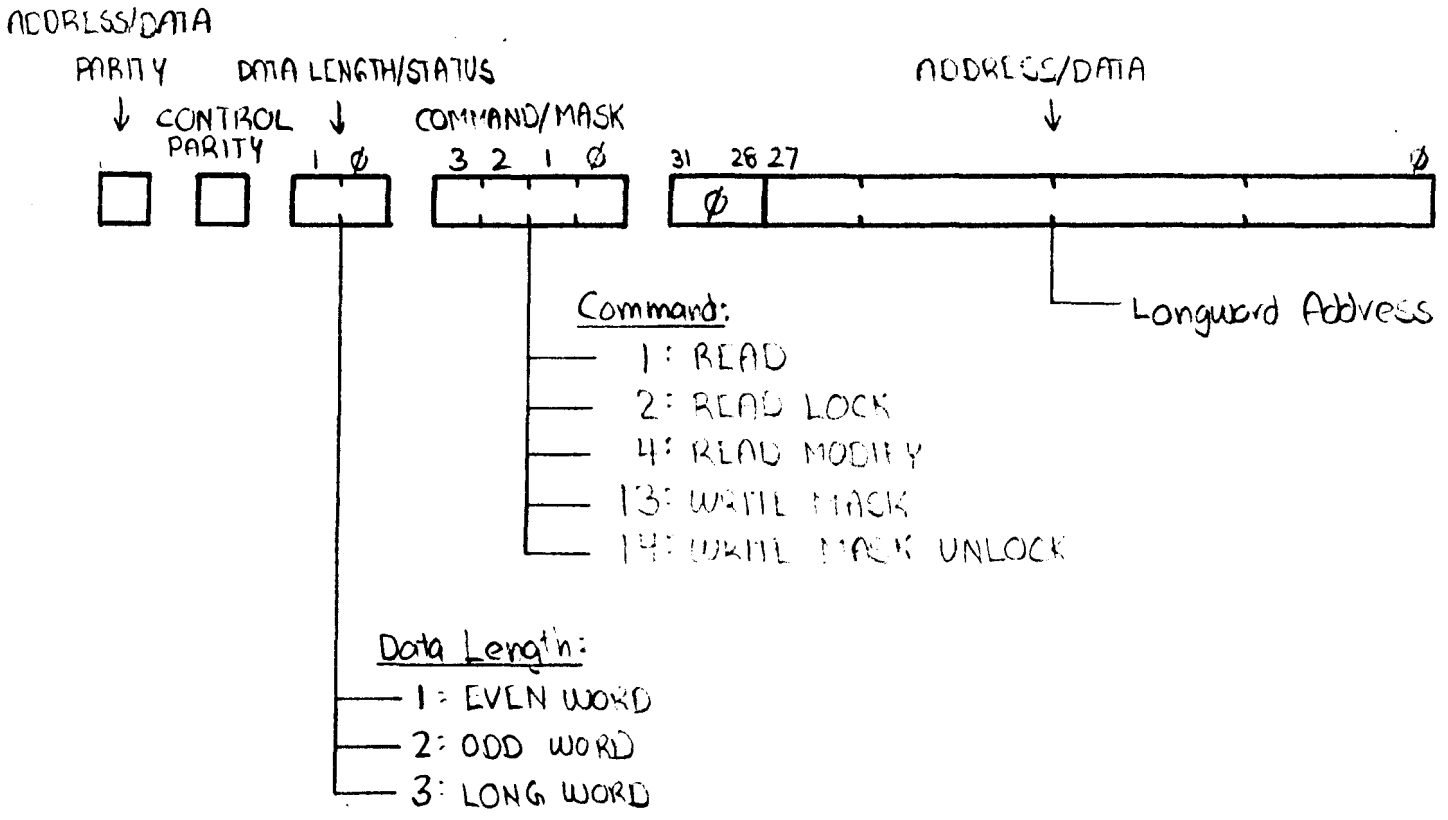


FIGURE 7 : CPU COMMAND/ADDRESS CYCLE FORMAT

J. LACY
25-FEB-80

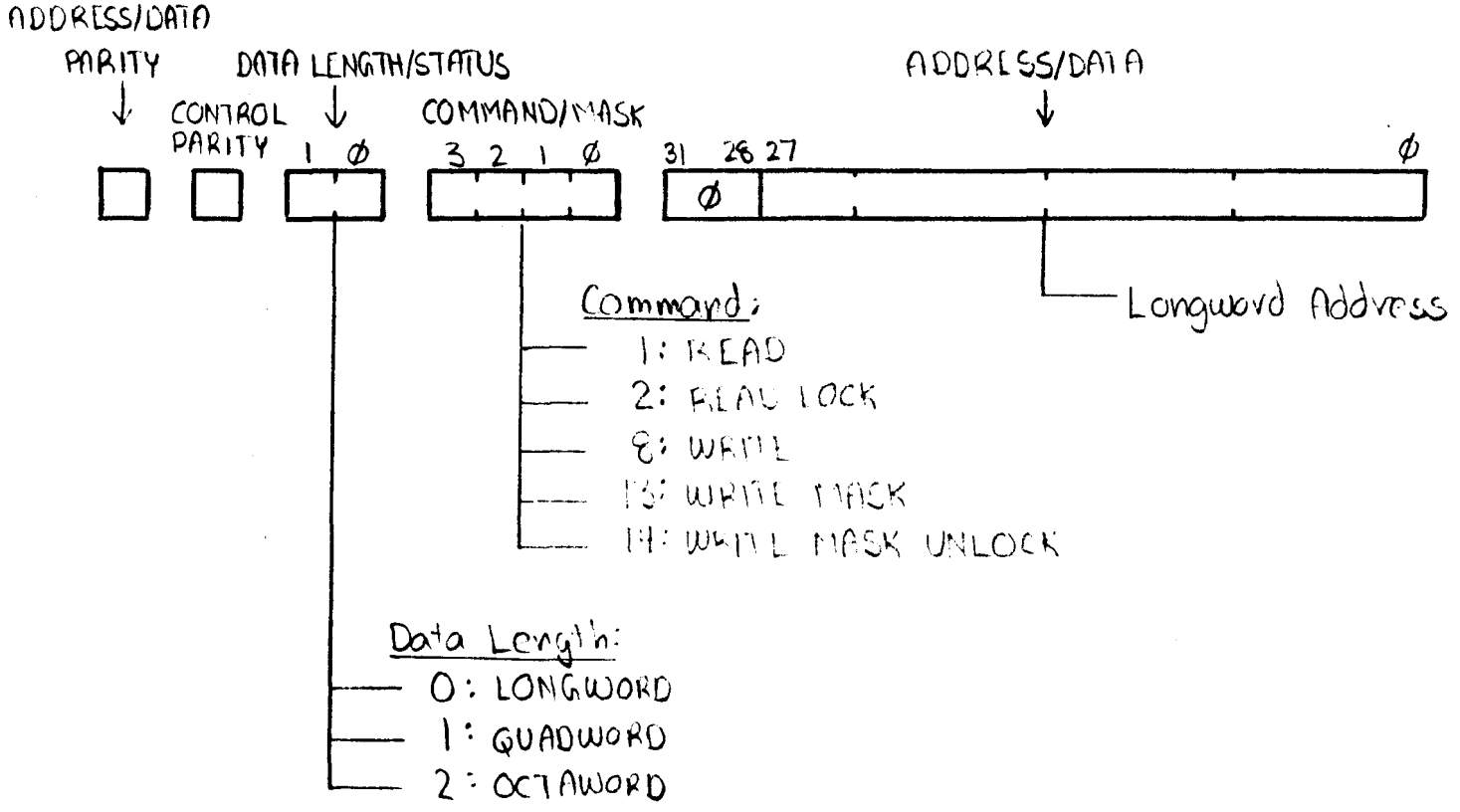


FIGURE 8: DMA COMMAND/ADDRESS CYCLE FORMAT

J. LACY
25-ILB-80

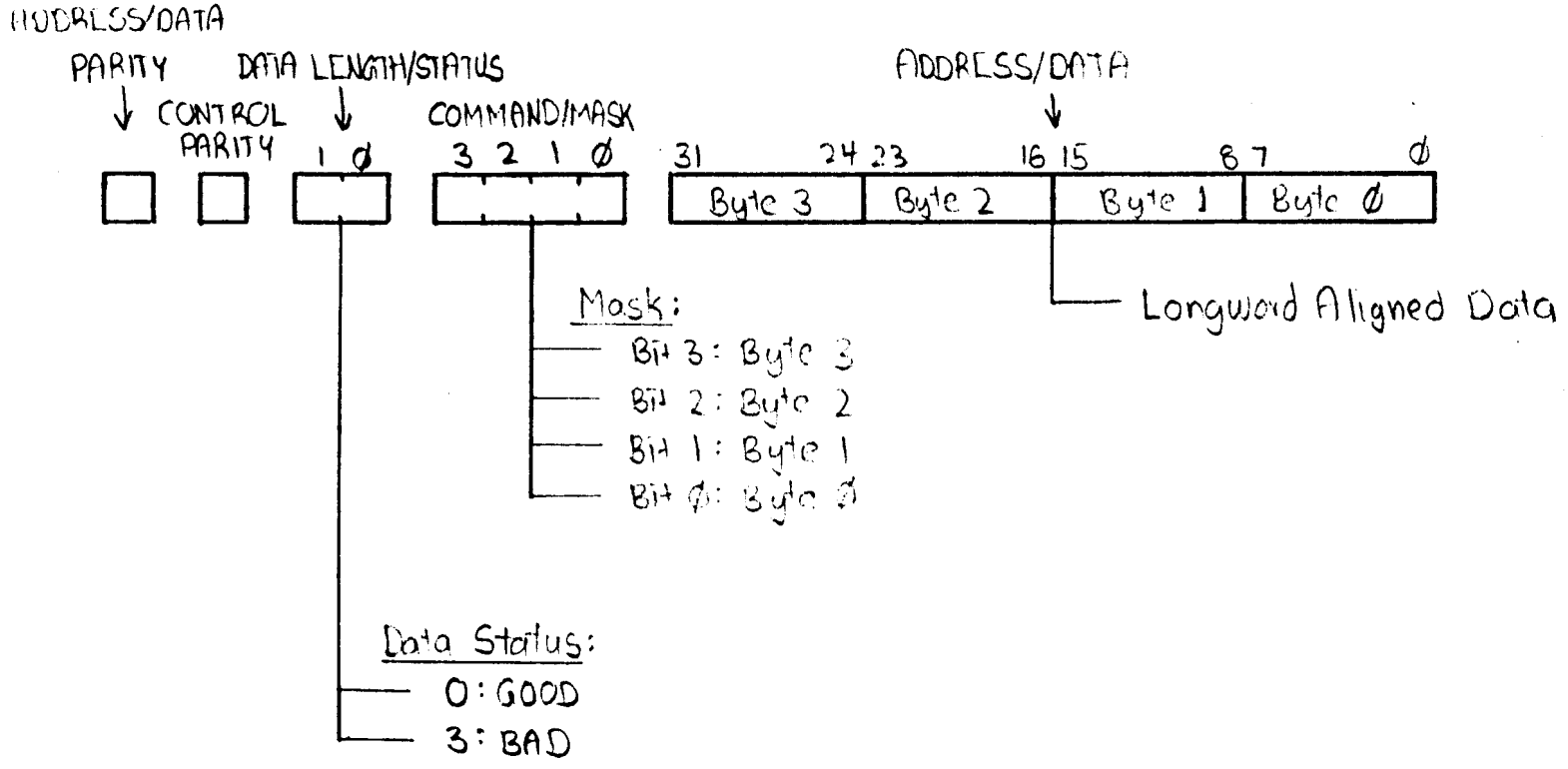


FIGURE 9: WRITE DATA CYCLE FORMAT

J. LACY
25-FEB-80

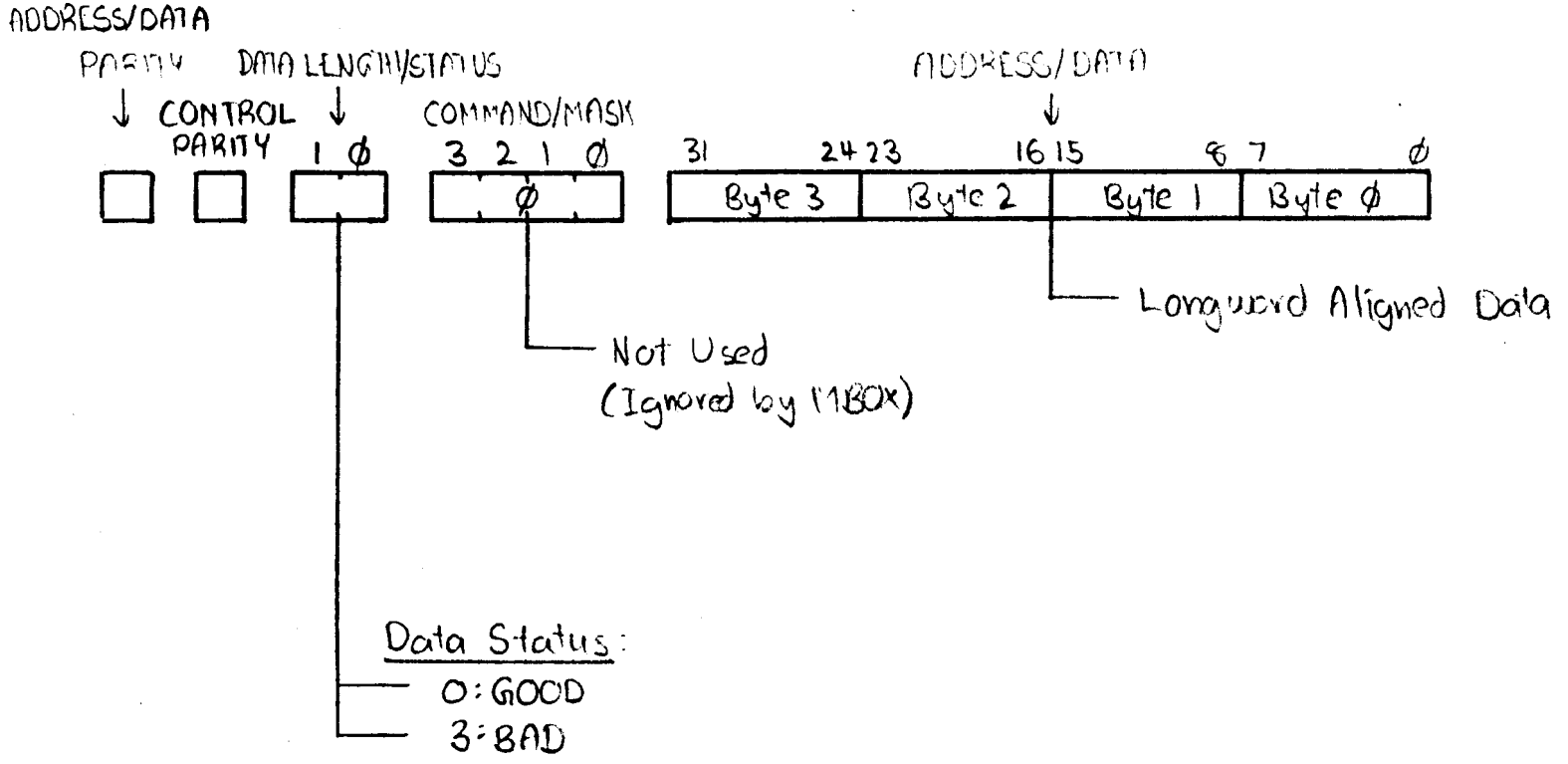
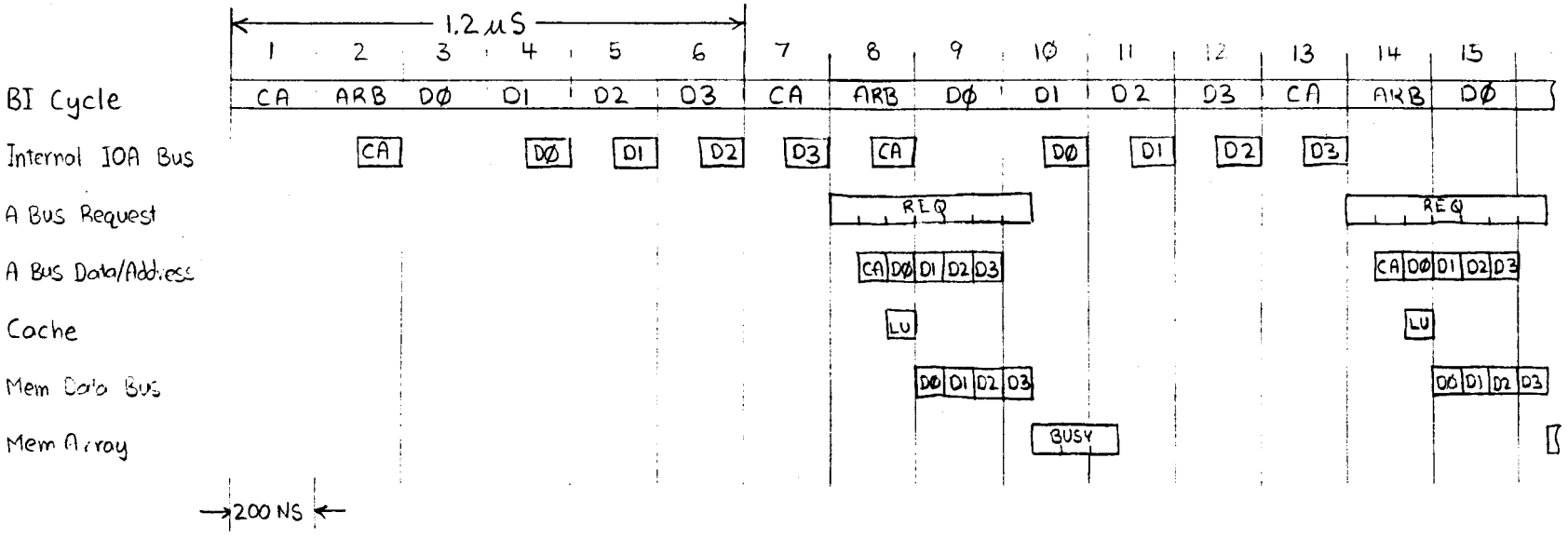


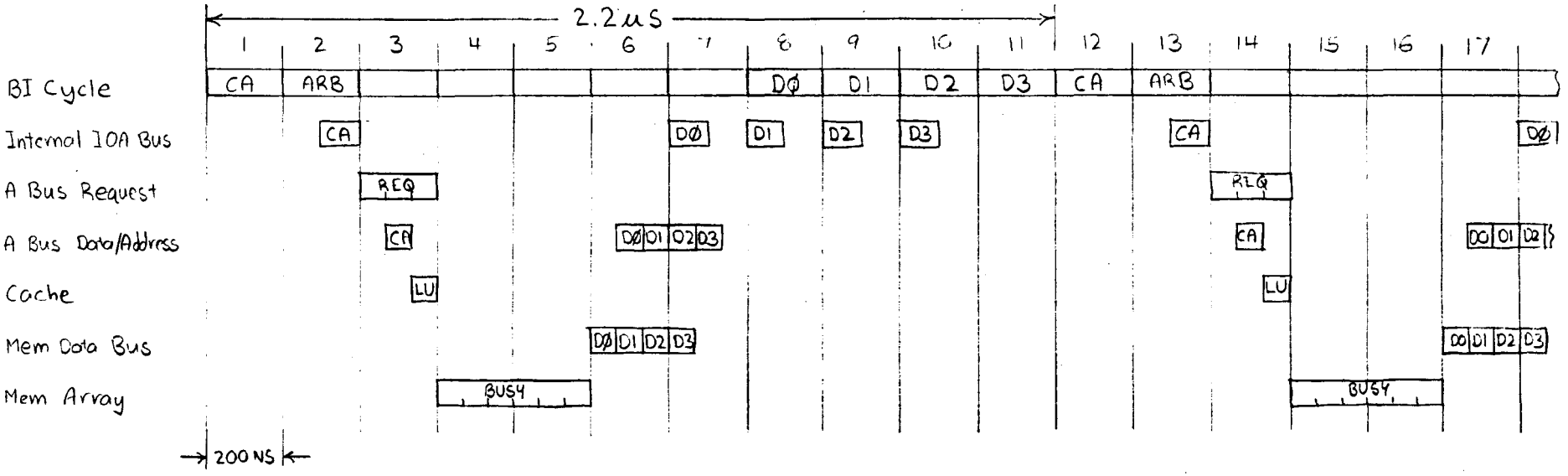
FIGURE 1Ø: READ DATA RETURN CYCLE FORMAT

J. LACY
25-FEB-ED



LU = Cache Look-Up

FIGURE 11: OCTAWORD WRITE TIMING DIAGRAM FOR BI ADAPTER



LU = Cache Lock-Up

FIGURE 12: OCTAWORD READ TIMING DIAGRAM FOR BI ADAPTER

LACY
25 TEXAS

VENUS SBI ADAPTER (SBIA)

REV 1 February 1980

Author: Barry Flahive
MR1-2/E18
DTN 231-5738

Revision History

Rev	Date	Description	Author
1	2/80	Prepared for system design review. Specification is between Rev-0.2 and Pev-1.0	BJF

1.0 OVERVIEW

1.1 VENUS SBIA

The VENUS SBIA is a two extended hex module option which provides an interface from the VENUS A BUS to the SBI. The SBIA is implemented with TTL-MSI logic including PALs (80% of chip count), ECL 10K logic (15%) and custom LSI (DC022 - >3%). The SBIA provides both Processor NEXUS and Memory NEXUS functions to the SBI as well as generating the SBI clocks and providing termination of the bus.

The goals of the SBIA design are:

- SBI-based I/O subsystem with MASSBUS, UNIBUS, DR32 and ICCS
- DMA performance at least equivalent to 11/780
- Must work with all SBI devices except CPUs
- SBIA should be invisible to all run time software except for error service routines
- SBI devices and registers should appear the same as on 11/780
- Must not exceed two extended hex modules

The SBIA is not intended as a means of interconnecting the VENUS processor with another VENUS processor or with an 11/780 processor directly on the SBI. SBI memory (MS780, MA780) will work with the SBIA but, because the VENUS M BOX will not cache data read from the I/O Adapters, I-streams and D-streams cannot be efficiently executed from the SBI memory. Transfers to/from the SBI over the A BUS are from/to the VENUS M BOX. Transfers directly between I/O Adapters are not permitted.

1.2 SBIA Performance

1.2.1 DMA -

The SBIA will perform all memory NEXUS functions required by the SBI specification. These functions are: Masked Read, Masked Write, Extended Read, Extended Write Masked, Interlock Read Masked and Interlock Write Masked with all combinations of mask bits.

The bandwidths for DMA read and write activity which the SBIA can support will depend upon the type of accesses which are being performed. Masked writes and

reads which access less than 64 bits use memory inefficiently and, therefore, reduce the available bandwidth. The A BUS - M BOX - Memory is capable of delivering 10.0 Mbytes/s of read throughput to the SBIA (quadword operations) with continuous cache misses and 13.3 Mbytes of write throughput (quadword writes--no byte masking). A limited caching of longword and quadword read data by the M BOX will allow the read bandwidth to reach 13.3 MBytes/s (assuming 50% hit rate).

The key to the SBIA transfer rate is the DMA buffer organization and operation which must queue sufficient commands for efficient operation without degrading memory access time. The extended read access time through the SBIA is 1800 nS to the lower quadword and 2000 nS to the upper quadword (of the octaword referenced by the M BOX). The extended read access time for a cache hit is 1.4 uS.

The maximum bandwidth for DMA access is:

	Bandwidth MBytes/s
- Extended Read	13.3
- Extended Write Masked	
no byte masking	13.3
with byte masking	<6.6 *
- Read Masked	<7.0 *
- Write Masked	
no byte masking (32 bits)	8.5
with byte masking	<4.0 *

* - bandwidth depends upon the number of bytes accessed.

1.2.2 Processor Port -

The SBIA as processor NEXUS has the following capabilities:

- Read Masked with mask combinations <0011>, <1100>, and <1111>
- Write Masked with all mask combinations
- Interlock Read Masked with mask combinations <0011>, <1100>, and <1111>
- Interlock Write Masked with all mask combinations
- Interrupt Summary Read (via SBIA register read)
- Fault latching (freezes the Fault/Status registers until service routines record the information)
- UNJAM (via SBIA register write) to restore an idle state to the SBI
- QUADCLEAR (via SBIA register write) to clear ECC errors in SBI memories

The SBI RAMP features of 11/780 are also part of the SBIA processor port. These RAMP features are:

- 16 line X 32 bit SBI Silo which records sequences leading to SBI Faults
- SBI Maintenance Register which provides diagnostic functions to aid in the diagnosis of SBIA and SBI device logic
- Margin frequencies for the SBI clock (no SBI single step)

1.3 SBIA RAMP Features

The SBIA provides and checks parity on the data paths between the SBI and the A BUS and where possible carries the parity through from one to the other. True parity carry through is not possible because of the protocol differences between the busses (i.e. mask bits at different times; even vs odd parity; different command functions; etc.). On data paths terminating on the SBIA parity is checked or generated. The register files which are used in the A BUS interface are protected by parity.

In addition, to checking the parity of information

received from the SBI and A BUS, the SBIA will check for correct protocol (i.e. write data follows write command/address, valid function codes, etc.)

When error conditions are detected in the SBIA information about the type of cycle and the originator of the cycle will be stored in the SBIA registers.

SBIA diagnostic features include the ability, under micro-diagnostic control, to loop back through the SBI. Parity invert/disable functions allow checking parity networks and error reporting logic.

2.0 BLOCK DIAGRAM DESCRIPTION

Figure 2.1 is a rough block diagram of the SBIA. The A BUS interface and its associated logic will be on one module; the SBI interface and its associated logic will be on the other module. The exact module partitioning of the remaining logic has at this time not been determined.

2.1 A BUS Interface, Control And Register File

The SBIA logic for the most part runs synchronous to the SBI and asynchronous to the A BUS. In order that A BUS transactions occur efficiently, these transactions are conducted by reading or writing a dual-ported register file which is loaded before or unloaded after the A BUS transaction.

The A BUS is implemented with ECL logic, and, therefore, the A BUS interface includes TTL to ECL conversion. The general details of the A BUS interface and control logic as well as the register file operation is defined by the A BUS specification.

2.2 S-Data Assembly

The S-Data assembly logic takes information which the M BOX has placed in the register file and converts the information to SBI format. Conversion to SBI format includes: changing the byte address to a longword address; generation of TAG codes and function codes; assembling the byte mask information; and in the case of I/O addresses, mapping the address. The S-Data assembly logic includes parity checking logic for checking the register file data.

If the information removed from the register file is a read or write command for an SBIA register, the appropriate control logic is signaled by the S-Data assembly logic.

2.3 A-Registers

The A-Registers is the set of registers which define and control the SBIA operation. These registers include: a configuration register which identifies the SBIA on the A BUS; the A-Memory register which defines the upper limit of the VENUS internal memory address range; at least one Error register which reports information on error types and status for errors

detected within the SBIA; 15 Interrupt Vector registers from which the vectors for SBIA generated interrupts are obtained; and a Maintenance register for enabling diagnostic functions.

2.4 S-Registers

The S-Registers provide control and status information about the SBI interface logic and the SBI bus. The S-Registers include the six registers containing SBI information which were defined by 11/780. These six registers are: SBI Fault/Status register; SBI Silo; SBI Silo Comparator; SBI Maintenance register; SBI Error register, and SBI Timeout Address register. The SBI Quadword Clear register is a write only register which is also included in the S-Registers.

2.5 A-Data Assembly

The A-Data Assembly logic converts SBI format commands and data to A BUS format commands and data. This conversion includes: conversion of the longword address to a byte address; a function code generator; the A BUS byte mask assembler; and parity generation logic.

2.6 SBI Interface

The SBI Interface includes the chips which sample the SBI information transfer lines each cycle and pass the result to the SBI Protocol logic and to the A-Data Assembly logic.

2.7 SBI Protocol

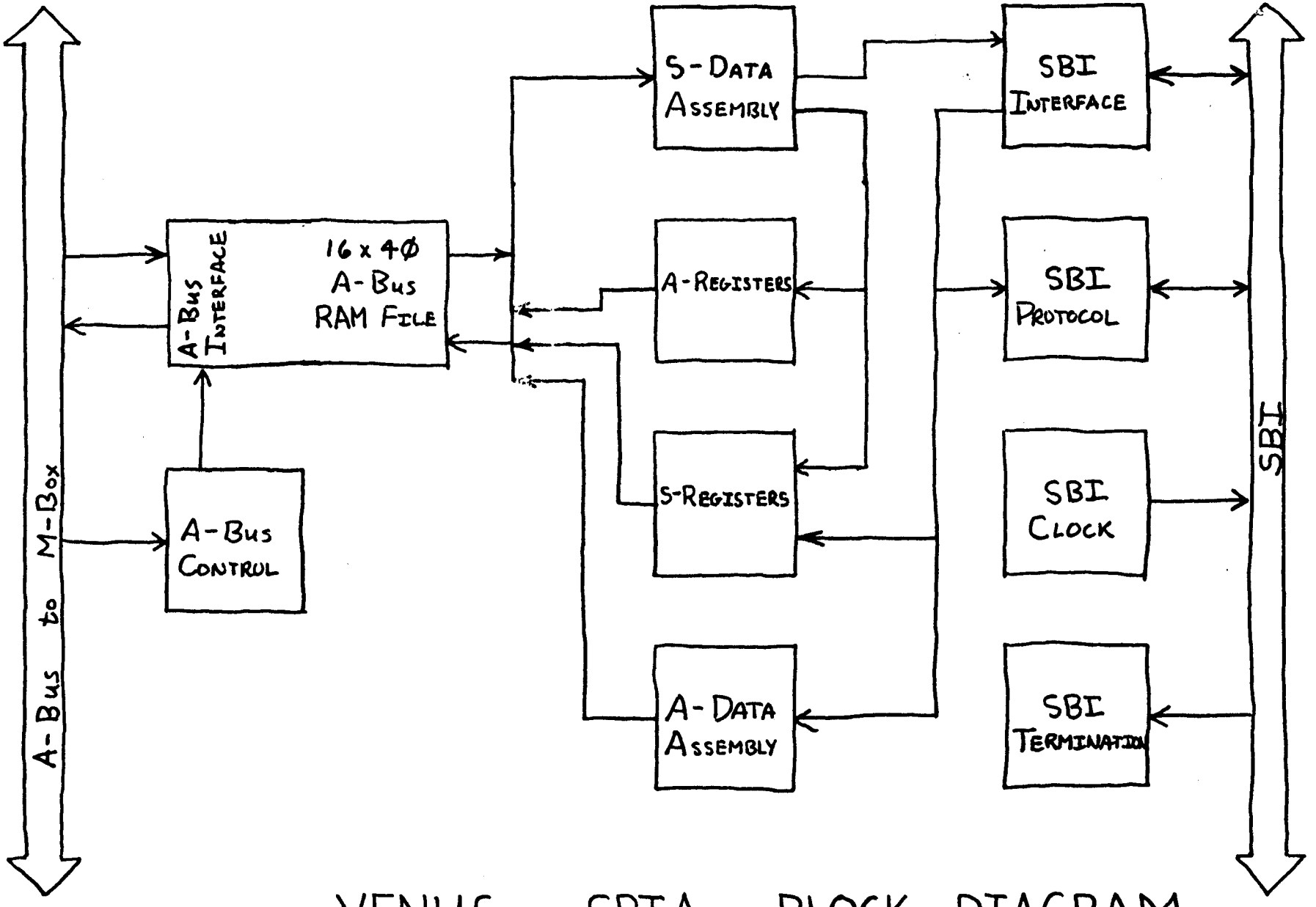
The SBI Protocol logic provides the SBI protocol checking required of all SBI NEXUS as well as verifying commands and data received by the SBIA. The SBI Protocol logic includes: parity checking of SBI information; check of valid functions; check for SBIA addresses; check for proper data sequence; SBI confirmation logic; SBI arbitration logic; and SBI power fail logic.

2.8 SBI Clock

The SBI Clock generates the timing signals required by the SBI. This circuitry will be very similar to the 11/780 SBI Clock.

2.9 SBI Termination

The SBI Termination provides TTL termination (70 ohms to +3.0 volts) for the 78 TTL SBI signals. The six ECL SBI Clock signals are not terminated at the SBIA end of the SBI.



VENUS SBIA BLOCK DIAGRAM

12-Oct-79 Rev B
Barry Flahive Feb. 80

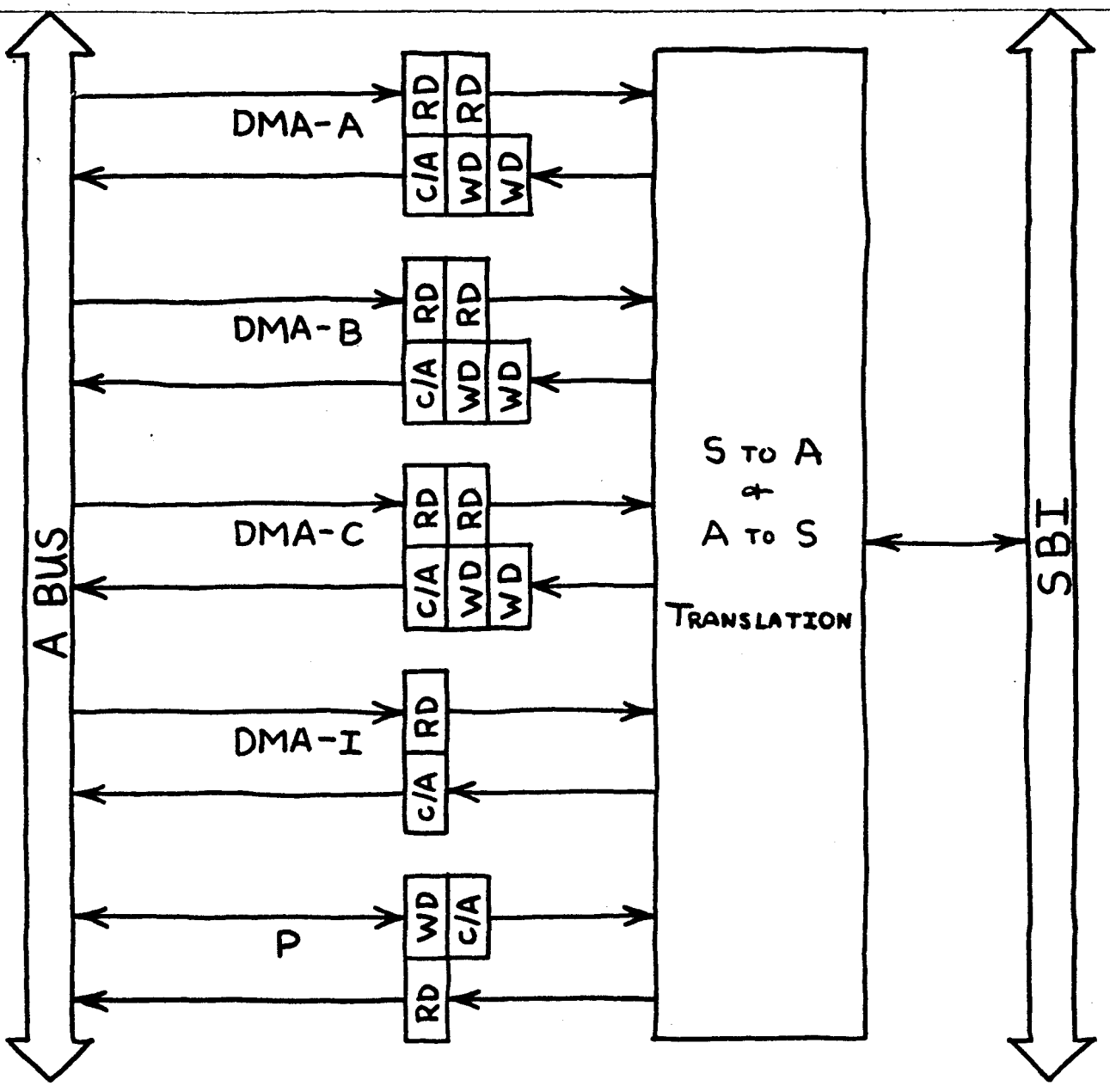
Figure 2.1

3.0 SBIA DATA PATHS

3.1 Buffer Organization

Information transferred between the SBI and the A BUS is passed through one of five buffers. SBI to/from VENUS memory transfers are passed through one of the DMA transaction buffers: DMA-A or DMA-B or DMA-C or DMA-I. VENUS CPU to/from SBI transfers are passed through the Processor (P) buffer. Each transaction buffer is of fixed length and is associated with a transaction from the time that the command is accepted untill all read or write data has been transfered.

Figure 3.1 shows the conceptual SBIA buffer organization. As shown in Figure 3.1, DMA-A, -B, and -C buffers require 3 lines for SBI to VENUS memory transfers and 2 lines for read data returned to the SBI. The Processor buffer requires two lines for CPU to SBI transfers and one line for read data returned to the CPU (the processor is limited to longword transactions). The DMA-I buffer is used for Interlock Read Masked commands and therefore requires one line for a command and one line for read data. The reason for a separate buffer is that the assertion of the "Memory Interlock" on the A BUS must stop any arbitration for Interlock read commands. By holding the interlock read traffic in a separate buffer normal DMA traffic will be unaffected when an Interlock read is held up in the DMA-I buffer. Figure 3.2 shows the organization of the buffers into a dual ported file using the same lines for read data and write data.

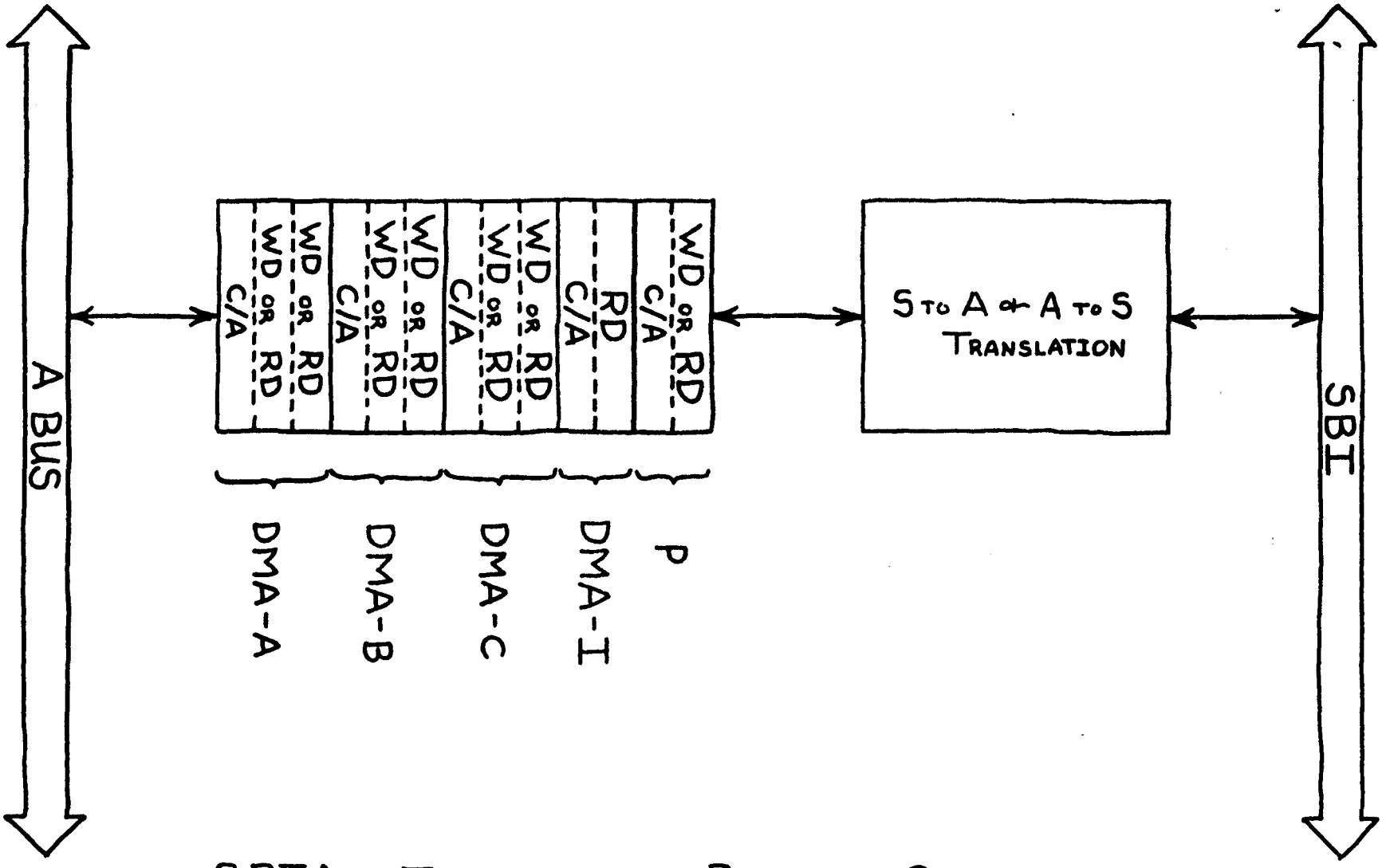


SBIA

CONCEPTUAL BUFFER ORGANIZATION

REV B
FEB. 80
B. FLAHIVE

FIGURE 3.1



SBIA TRANSACTION BUFFER ORGANIZATION

REV B
FEB. 80
B. FLAHIVE

FIGURE 3.2

3.1.1 DMA-A, -B, -C - These buffers are used to provide an efficiently buffered path between the SBI and the A BUS. The logic which acknowledges SBI commands and loads the buffers will not permit the queueing of more than two commands in the buffers. If more than two commands were queued then the worst case read access time would be significantly degraded. The reason for the third buffer is to allow one buffer to have read data to be returned to the SBI, while the other two buffers are holding commands. Under a continuous stream of reads one buffer will be returning read data to the SBI, one buffer will be loading a command, and the third buffer will be waiting for read data from the A BUS.

3.1.2 DMA-I - The DMA-I buffer operates in parallel with the other DMA buffers. The loading of the DMA-I buffer is not inhibited by the traffic in the other DMA buffers or vice versa. The processing of commands in the four DMA buffers must be kept in sequence so the DMA-I buffer will not arbitrate for the A BUS untill all commands in the DMA-A, -B, and -C buffers (a the time of DMA-I load) have been completed. The DMA-I buffer will then be the next buffer serviced by the M BOX (when Memory Interlock is deasserted).

Interlock write commands will be processed through the the DMA-A, -B, and -C buffers since no special arbitration must be performed for these commands.

While a DMA interlock is outstanding to the SBI no additional DMA interlock read commands will be accepted by the SBIA. During this time the SBIA will keep a 512 SBI cycle Interlock timer which will release the A BUS interlock if no Interlock write is received from the SBI.

3.1.3 P Buffer - The two line P buffer handles all processor traffic to: the SBIA registers; SBI device registers; and to SBI memory.

The P buffer operates independently of the DMA buffers. The command/address (C/A) is loaded explicitly by the M BOX (along with any write data). When the command address is loaded the SBIA asserts a wait code onto the A BUS. The SBIA examines the C/A at a time when DMA traffic is not using the interface file. The command will then be executed (again using cycles which DMA traffic does not need). When the processor command is completed (register write completed; or SBI write acknowledged; or read data loaded into the interface file), the SBIA will change the wait code into a done code signaling the M BOX that the read data can

be read or a new command can be loaded.

3.2 DMA Data Path Operation

The major components of the SBIA used in DMA transactions are shown in Figure 3.3. Commands received from the SBI are validated by the SBI Protocol logic and loaded into the interface file in parallel with synchronizing the A BUS request signal. The file bus is a tri-state TTL bus with a 100 nS cycle. Each 200 nS SBI cycle has one load and one unload cycle on the file bus. Unload data is latched before parity checking, and is converted to SBI format in parallel with the parity check. The parity check path is the longest delay path between the A BUS and the SBI. In order to accomidate the delay of the parity check, the file unload cycle is placed in the first half of the SBI cycle.

The components of DMA access to the VENUS memory are shown in Figures 3.4 - 7.

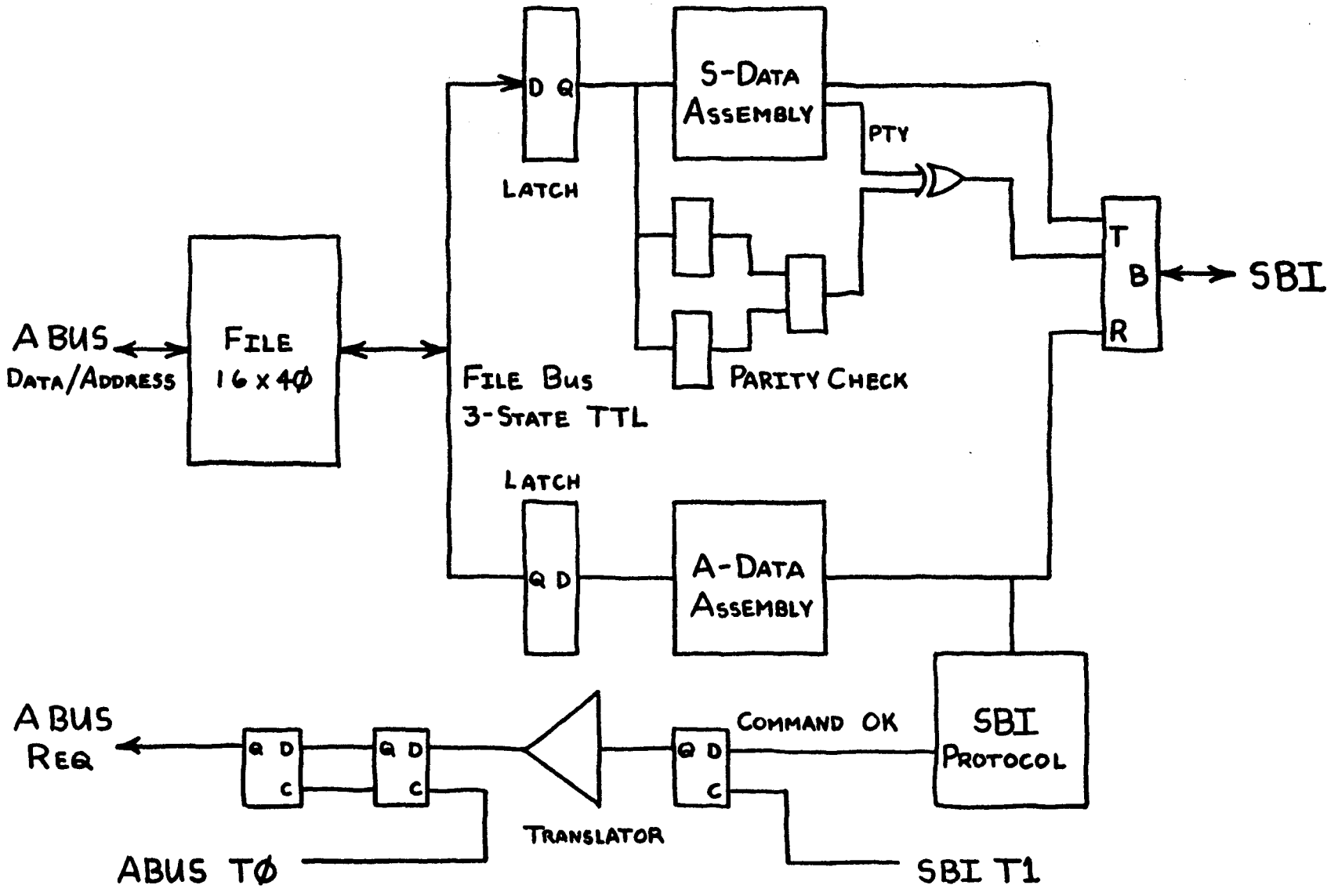


FIGURE 3.3

SBIA DMA DATA PATH

REV B
FEB. 8Ø
B. FLAHIVE

CYCLE

SBI B<31:

FILE B

A P

17

16

CYCLE

17

SBI B<31:

18

FILE A

A

CYCLE

∞

SP=

CYCLE

17

SBI B<

18

FILE

/

FEB 26 1980

digital

INTEROFFICE MEMORANDUM

TO: List

LOC/MAIL STOP

DATE: 25 FEB 1980
FROM: Sas Durvasula
DEPT. L.C.E.G.
EXT. 231-4426
LOC/MAIL STOP: MR1-2/E47

SUBJ: VENUS CPU READING MATERIAL

COMPANY CONFIDENTIAL

Attached, please find the "Venus CPU Reading Material" for your review prior to the February 29th meeting (PK3-1, 8:30 to 5:00). CVR not scheduled

The SBIA and A-Bus information will be mailed to you on the 25th (Blue Mail).

It is very helpful if all the attendees read the material ahead of time to make the discussions productive for the design team members.

Thank you and contact me if you have any questions in the meantime.

NOTE: Recommended Reading Sequence of the Material:

1. Venus Processor Overview
2. I Box
3. E Box
4. F Box
5. M Box
6. Console
7. RAMP Strategy
8. E/I Boxes Data Path Parity Scheme
9. A Bus Description
10. SBI Adapter

attachments

/pas

MAR 27 1980

digital

INTEROFFICE MEMORANDUM

TO: Distribution

DATE: 26 March 1980
FROM: George Hoff
DEPT: LSG Technology/VAX Eng.
EXT: 231-6524
LOC/MAIL STOP: MR1-2/E47

SUBJ: VENUS RAMP ACTIONS IN RESPONSE TO ATTACHED MEMO

A VENUS RAMP team has been established which is focused on addressing these issues. The team includes Hardware Design, Technology, Field Service and VMS. The goal is to resolve open issues and conduct a RAMP review to support Phase I closure.

The key issues being pursued at this point are:

1. MCA failure rate versus junction temperature. VENUS cooling design limits versus junction temperature. MTBF of CPU at proposed design point.

   Action: Walton/Zia/McElroy/Robey

2. Closure of checking strategy between Field Service and Hardware Design.

   Action: Leonard/Robbins/Robey

3. Resolution of what commitments we have from Central Power group on MPS MTBF, along with plans to prove via DMT.

   Action: Chin/Butala

4. Review of VMS RAMP features versus KL SEP (ML Service Enhancement Program), items for TOPS-10/TOPS-20.

   Action: Leonard/Samuelson

5. Review of VENUS hardware versus KL SEP hardware items.

   Action: Leonard/Zia

6. Resolution of open issues in the area of power/ environmental sensors and monitoring.

   Action: Chin/Robey

The goal is to get the data and review prior to a RAMP review before the end of April. I could use some inputs on who should attend the VENUS RAMP review.

GH:rj
Attachment

DISTRIBUTION

Gordon Bell	ML12-1/A51
Don Busiek	PK3-2/S17
Chuck Butala	MR1-2/E47
Derrick Chin	MR1-2/E47
Sas Durvasula	MR1-2/E47
Ulf Fagerquist	MR1-2/E78
Andy Knowles	MR1-1/A65
Vic Ku	MR1-2/E47
Jud Leonard	MR1-2/E47
Jim McElroy	MR1-2/E47
Mike Robey	MR1-1/S35
Frank Robbins	MR1-1/S35
Chuck Samuelson	TW/D08
Bill Walton	MR1-2/E47
Sultan Zia	MR1-2/E47
Dick Hustvedt	TW/D08

Geoff Bill knows this area. Matt & Jane Watt We can get the right size before we build it or modify it afterwards.

MAR 24 1980

+---+---+---+---+---+---+---+---+
| d | i | g | i | t | a | l |
+---+---+---+---+---+---+---+---+

INTEROFFICE MEMO

To: Sas Durvasula

cc: [redacted]
Sam Fuller

Date: 19-Mar-80
From: [illegible] Stracker
Dept: VAX [illegible]/PDP-11 SYS. ARCH.
Ext: 247-2180
Loc/Mail Stop: TW/B05

Subject: VENUS REVIEW

Overall, I am very impressed with the VENUS project. It looks like a real winner! I have only one serious reservation: The size of the cache. It appears that you have designed a machine which is 5 to 7 times a 780 and then slowed it down to [illegible] times a 780 by use of a small cache (with an assumed 95% hit ratio). Furthermore, at a system level (as opposed to single program benchmark level) we have no idea what the hit ratio will actually be.

I strongly recommend that SPA/VENUS engineering put together the following experiment. Attach counters to a 780 to record cache hits and misses. Run a significant system load (like the educational benchmark) and determine the real system level hit ratio. (It would be particularly nice to determine the translation buffer hit ratio at the same time.) Then, from the cache hit ratio, estimate the VENUS/780 performance ratio. I will be happy to consult on how to estimate the performance ratio for other cache sizes.

Based on this work, we can understand whether we have made the right tradeoffs on the VENUS cache.

MAR 2 7 1980

digital

INTEROFFICE MEMORANDUM

TO: Gordon Bell

DATE: 26 March 1980
FROM: George Hoff GH:2
DEPT: LSG Technology/VAX Eng.
EXT: 231-6524
LOC/MAIL STOP: MR1-2/E47

CC: Sas Durvasula
Ulf Fagerquist
Bill Strecker

SUBJ: CACHE SIZE FOR VENUS

As noted in the attached memo, we have decided to increase the cache size from 8KB to 16KB. This can be accomplished without any major impact on the program. We do not plan to provide facilities to expand beyond 16KB. This cannot be supported within our packaging and performance constraints.

We are presently working with Bill Strecker and Joe Emer to get measurement data.

GH:rj
Attachments

digital

INTEROFFICE MEMORANDUM

TO: List

DATE: 20 March 80
FROM: Sas Durvasula
DEPT: Large VAX Engineering
EXT: 231-4426
LOC/MAIL STOP: MR1-2/E47

SUBJECT: 16KB CACHE FOR VENUS

The recent Venus CPU structure review has opened up a question about the performance limitation due to the size of the cache (8KB at present) and there was considerable push to look at increasing the cache size. We have done a lot of thorough analysis in the past two weeks and we have also discussed the schedule, price/performance, and design stability issues. This memo summarizes the results of all the work.

SUMMARY OF TECHNICAL INFORMATION:

1. The Venus cache increase to 16KB can be achieved with no impact to the rest of the design.

2. The additional module design is very straight forward and the complete design has been thoroughly characterized by Al Helenius and Bill Bruckert.

3. All the MCA's can still be utilized without changes.

4. The parts estimate indicate that the board is easy to layout.

5. The three board cache speed analysis indicates it will run at 69ns based on the analysis. (Using worst case, Bill Walton's numbers for metal & F.O.).

6. The TB uses 256 x 4 RAM and the size of the TB is reduced to 256 lines.

7. Cache cycle time gain is between 22% (90% hit) to 29% (85% hit) compared to the 8KB cache.

8. The cache is fully ECC protected improving the RAMPNESS.

9. No slip in schedule.

10. Transfer cost increase is 1K.

11. Poses a backplane design and lack of spare slot problem which is being investigated and there are several alternative solutaion.

12. The new price/performance is worth 2 quarters in schedule and 16KB cache is an excellent selling feature.

The decision to base our Phase I Plan on 16KB has been unanimous (by the design team) and Jud Leonard & George Hoff back this decision. Discussions with product management (Carl Gibson) have indicated that the performance vs cost tradeoff is positive as long as schedules are not impacted.

SD:ph

****
* d i g i t a l *
****

TO: WAYNE ROSING

cc: see "CC" DISTRIBUTION

DATE: FRI 1 AUG 1980 10:14 AM E
FROM: GEORGE HOFF
DEPT: LSG TECH/VAX ENGR
EXT: 231-6524
LOC/MAIL STOP: MR1-2/E47

SUBJECT: SEE BELOW

SUBJECT: GORDON'S COMMENTS ON BiPOLAR MEETING MINUTES RELATIVE TO VENUS AND NAUTILUS

What I net out from Gordon's comments, is that experience gained in VENUS and comparing VENUS versus 2080, should be considered in the Nautilus Technology tradeoff process. To help in doing this , I have attempted to address some of the issues Gordon has raised.

Lessons Learned

1. Vendor Schedules: The MCA was originally planned for DOLPHIN and is coming in over two years behind the original Motorola forecast. I would put this type of adder on initial vendor schedules if the process is new and they do not have substantial experience with gate arrays of equivalent complexity.

2. Comet and VENUS both demonstrate that gate arrays add about one year to the development schedule. Given a fully characterized array and perfect CAD tools, this might be cut to six months.

3. It takes about one year to fully characterize a gate array. I believe the MCA characterization has provided a valuable base of knowledge which would benefit a Nautilus effort just as VENUS benefitted from the Comet effort. The availability of micro products engineers with extensive BiPolar experience, is crucial to success in this area. I would hesitate to tackle a gate array development in the future without a micro products like organization to back me up technically.

4. Things have changed since we made the 100K vs. MCA tradeoff over a year ago. Both technologies have slipped performance (100K slipped a little more than the MCA). The costs for MCA's have stayed constant while 100k has come down. The source situation for 100K has improved with RTC coming along rapidly and Hitachi getting in; this was speculative last year. I still believe the 100K market will be volatile until volume builds to provide stability. The volume commitment for the MCA has held firm - we had a firm commitment last year.

As for the incremental costs for a gate array machine, they include: CAD development (simulation, placement, routing, test patterns, timing analysis, soft tooling for fabrication), option tooling/qualification lot costs, second source capital, IC test equipment and programs. In the VENUS program, development costs associated with MCA, will total about $3.5M - 4M excluding Hudson.

6. There is an interdependency between IC technology and PCB technology that has a significant impact. The ability to pack/interconnect/cool devices on PCB's is a key element in the gate array payback model. This area of interconnect success prediction is complex and there are few tools, if any, available to measure this.

7. There is a relationship between LSI content and MTBF which increases the payback for gate array machines. This payback can be both quantitative (lower field service cost) and qualitative (customer satisfaction).

8. I believe there are manufacturing returns as well, i.e. testing more of the machine on auto feed LSI testers and less at QV testers with scopes. LSI forces a lot of discipline in the design process and a lot of investment in the test process which should yield dividends in manufacturing RAMP up, reduction of labor content, floor space etc. I don't know how to quantify this. Perhaps manufacturing experience with 11/780 vs. Comet, will provide some data.

9. The analysis Gordon is apparently requesting, is a life cycle cost/return evaluation of technology alternatives. We now have a life cycle model for VENUS and I could attempt to run 100K vs. MCA through this and see what results. My guess is that the time to market impact will dominate all other factors. The incremental development costs will be offset by product cost reductions/maintenance cost reductions.

10. There are issues beyond the cost/return model, i.e., if we drop out of BiPolar LSI, where will we be in five years? My belief is that we will be in deep trouble. Each pass at gate arrays yields learning and improved results, i.e., VENUS builds on Comet experience, Nautilus could build on VENUS experience. I feel there is an analogy to the LSI-11, FONZ-11, T-11 Scorpio.

    The initial steps are painful and there was always an easier off the shelf solution, however, the return builds as the process matures. We need a higher level strategic model for classes of products to measure the impact of product decisions. We could build Nautilus out of 10K ECL or super Schottky and eliminate our future in the market when IBM comes out with the

follow on to the 4300.

"CC" DISTRIBUTION:

*GORDON BELL
JIM CUDMORE
ULF FAGERQUIST
LARRY PORTNER
DICK CLAYTON
BILL DEMMER
BILL GREEN
BRIAN CROXON
RUSS DOANE
MITCH KUR

FEB 15 1980

digital

INTEROFFICE MEMORANDUM

TO: Venus Phase I Review List

cc: Gordon Bell
Si Lyle
Bill McBride

DATE: 7 FEB 1980
FROM: Vic Ku
DEPT: L.C.E.G.
EXT: 231-6202
LOC/MAIL STOP: MR1-2/E47

SUBJECT: VENUS PHASE I REVIEW CRITERIA

The Venus Phase I exit criteria, proposed by Vic Ku and agreed to by George Hoff, Carl Gibson and Len Kreidermacher, consists of accomplishing the following tasks (see the attached pages) with no major loose ends or unacceptable risks (unless waivered by Ulf Fagerquist or George Hoff). All major loose ends, unacceptable risks and waivers, if any, will be clearly documented by Vic Ku, the Program Manager.

A Phase I meeting is scheduled for March 10, 1980 (your secretaries will be contacted) to review the "System and Technology" and the "Venus Development Plans and Specifications" tasks with the goal of completing these tasks and to give Carl Gibson,by March 15, 1980, our signed Venus System Development Plan in order for Carl to review our proposal with the Product Lines, update his business plan and prepare the Phase I Product Contract by the end of March, 1980.

A second Phase I meeting will be scheduled, by Vic Ku, at the end of March, 1980 with the goal of completing all Venus Phase I criteria and officially launch the Venus program into Phase II, assuming the above two meetings did not reveal major issues.

All functional managers have been informed of the tasks, the time frame and the exit criteria. Please try your best to complete your tasks and be prepared to identify and discuss the loose ends/ issues during the March 10 meeting.

attachments

/pas

I - SYSTEM AND TECHNOLOGY

Task 1	:	Corporate-wide Venus System Design Review
Responsible Person	:	Sas Durvasula
Targeted Completion Date	:	February 29, 1980
Exit Criteria	:	Minutes of the review, resolution of all major issues and resolution dates for other system issues.

Task 2	:	Corporate-wide Venus Technology Review
Responsible Person	:	Sultan Zia
Targeted Completion Date	:	March 7, 1980
Exit Criteria	:	Minutes of the review, resolution of all major issues and resolution dates for other technology issues.

Task 3	:	Motorola Contract Signed
Responsible Person	:	Bill Walton/Peggy Wesley
Targeted Completion Date	:	February 29, 1980
Exit Criteria	:	Contract signed.

Task 4	:	MCA Process and Characteristic fully qualified
Responsible Person	:	Bill Walton
Targeted Completion Date	:	June, 1980
Exit Criteria	:	Resolution of all major issues, resolution dates for other MCA qualification issues.

II - VENUS DEVELOPMENT PLANS AND SPECIFICATIONS

Task 1	: Venus System Development Plan-
Responsible Person	: Vic Ku
Targeted Completion Date	: February 29, 1980
Exit Criteria	: Plan signed by Venus Development Managers, Carl Gibson and George Hoff.
Task 2	: Venus CPU/IO Adapters Development Plan- I/E/M/F Boxes Engineering & Diagnostic Console SBIA, BIA Engineering breadboard, prototype, manufacturing photo/pilot supports
Responsible Person	: Sas Durvasula
Targeted Completion Date	: February 29, 1980
Exit Criteria	: Signed by all functional managers affected by or committed to the plan.
Task 3	: Venus Technology Development Plan - Circuit Power System Mechanical and packaging Environmental International regulation CAD Memory array
Responsible Person	: Sultan Zia
Targeted Completion Date	: February 29, 1980
Exit Criteria	Signed by all functional managers affected by or committed to the plan.
Task 4	: Venus MCA Engineering/Mfg Development Plan- MCA engineering and charaterization MCA CAD tools Contract/purchasing MCA testing MCA 2nd source Hudson start up
Responsible Person	: Bill Walton
Targeted Completion Date	: February 29, 1980
Exit Criteria	: Signed by all functional managers affected by or committed to the plan.

Task 5	: Venus Diagnostic Development Plan- Venus CPU cluster SBI based peripheral BI/CI based peripheral Diagnostic qualifications CPU cluster simulation supports
Responsible Person	: Dale Cook
Targeted Completion Date	: February 29, 1980
Exit Criteria	: Signed by all functional managers affected by or committed to the plan.

Task 6	: Venus Peripheral Development Plan- SBI based peripheral BI/CI based peripheral I/O adapters engineering/diagnostics
Responsible Person	: Roger Lawson
Targeted Completion Date	: February 29, 1980
Exit Criteria	: Signed by all functional managers affected by or committed to the plan.

Task 7	: Venus Manufacturing Plan - New product start up Volume and FA & T plans Mfg prototype/pilot plans Multilayer PCB plans Dock merge plan, PMT
Responsible Person	: Mike Elia/John Grose
Targeted Completion Date	: February 29, 1980
Exit Criteria	: Signed by all functional managers affected by or committed to the plan.

Task 8	: Venus Customer Service Plan- Field Service start ups Software service start ups BMC Training
Responsible Person	: Mike Robey
Targeted Completion Date	: February 29, 1980
Exit Criteria	: Signed by all functional managers affected by or committed to the plan.

Task 9	: Venus System QA Plan- Breadboard, prototype QA 102 and other regulations DMT, Software QA, Field test I/O qualifications
Responsible Person	: Ron Setera
Targeted Completion Date	: February 29, 1980
Exit Criteria	: Signed by all functional managers affected by or committed to the plan.

Task 10	: Venus Software Development Plan- VMS support of SBI based Venus VMS support of BI/CI based Venus
Responsible Person	: Chuck Samuelson
Targeted Completion Date	: February 29, 1980
Exit Criteria	: Signed by all functional managers affected by or committed to the plan.

Task 11	: Venus Operation Plan- Computers SUDS PC, multiwire and MCA processes, layout resources and tracking
Responsible Person	: Nick Cappello
Targeted Completion Date	: February 29, 1980
Exit Criteria	: Signed by all functional managers affected by or committed to the plan.

Task 12	: Venus Technical Documentation Plan
Responsible Person	: Ed McFadden
Targeted Completion Date	: February 29, 1980
Exit Criteria	: Signed by all functional managers affected by or committed to the plan.

Task 13	: Venus Product Description
Responsible Person	: Sas Durvasula
Targeted Completion Date	: February 29, 1980
Exit Criteria	: Signed by all functional managers affected by or committed to the plan.

Task 14	: Venus RAMP Specification
Responsible Person	: Jud Leonard
Targeted Completion Date	: February 29, 1980
Exit Criteria	: Signed by all functional managers affected by or committed to the plan.

III - PRODUCT MANAGEMENT AND PRODUCT LINES

Task 1	: Venus Phase I Business Plan
Responsible Person	: Carl Gibson
Targeted Completion Date	: March 31, 1980
Exit Criteria	: Signed by Vic Ku, George Hoff, Per Hjerppe, Ulf Fagerquist and Senior Marketing Managers selected by Carl Gibson.

Task 2	: Venus Phase I Product Contract
Responsible Person	: Carl Gibson
Targeted Completion Date	: March 31, 1980
Exit Criteria	: Signed by George Hoff and responsible person representing Product Line selected by Carl Gibson.

Gordon Bell
MZ 12-1/A51

VENUS PHASE I REVIEW LIST

George Berger	MR1-2/T17
Nick Cappello	MR1-2/E69*
Joe Carchidi	TW/D08
Mike Conroy	MR1-2/P74
Dale Cook	MR1-2/E68*
Sas Durvasula	MR1-2/E47*
Mike Elia	AC/B38 *
Bill English	MR1-2/E18*
Ulf Fagerquist	MR1-2/E78
Carl Gibson	MR1-2/E78*
John Grose	MR1-2/P74*
Per Hjerppe	MR1-2/E78
George Hoff	MR1-2/E47*
Len Kreidermacker	MR1-2/E18*
Vic Ku	MR1-2/E47*
Roger Lawson	MR1-2/E18*
Jud Leonard	MR1-2/E47*
Dick Maliska	MR1-2/E68
Jim McElroy	MR1-2/E47*
Ed McFaden	MR1-2/T17*
Art O'Donnell	MR1-1/S35
Roy Rezac	MR1-2/E78
Mike Robey	MR1-1/S35*
Chuck Samuelson	TW/D08 *
Ron Setera	MR1-2/E18*
Vern Stitt	MR1-2/E47*
Bill Walton	MR1-2/E47*
Sultan Zia	MR1-2/E47*

*Required to be present at the March 10, 1980 meeting.

APR 4 1980

++++++++++++++
!D I G I T A L!
++++++++++++++

INTEROFFICE MEMORANDUM

TO: Distribution

CC:

DATE: 14 March 1980
FROM: Sas Durvasala
DEPT: LSEG
LOC: MR1-2/E47
DTN: 231-4426

SUBJ: VENUS CORPORATE REVIEW

We wish to thank all who attended our review and thus contributed to its success. The meeting was productive and beneficial for us not only for the action items it generated, but also because preparation for it required us to get our act together, and the general enthusiasm shown for our design plan has given us the confidence needed to forge ahead. Again, your participation is greatly appreciated.

Sense of Meeting

Why wait on the FPA? Dave Cane and Ken Okin said this was a Bad Scene on Comet and Nebula; most 780s have FPAs, and it is a must for our current major markets. Dave pointed out that if FPA is not at highest priority, the schedule will slip and the gap between it and the CPU will grow. Everyone agreed "Do It Now!" and have the FPA available at FCS.

SCHEDULE (Sas Durvasula)

Question — Dave Rodgers
Really expect to have four Manufacturing prototypes for DMT in 3/82 after prototype power on in 10/81? -- DMT will be done using four prototypes, but this does not necessarily mean all four will be available in 3/82.

RAMP AND PERFORMANCE (Jud Leonard)

Action — Dick Hustvedt
What is the basis in experience for MTBF and other quantities? Unknown, and claimed 40 hour MTBF for 780 in the field is not believed. Must get information for 780.

Suggestion — Alan Kotok
Should get MTBF data not only for 780 but also for KL10 since it had a writeback cache like Venus.

Action — Chuck Samuelson, Joel Emer, Alan Kotok, Dave Rodgers
Chuck asked what is the meaning of 3.5 x performance, and suggested that the answer (3.5 times as many users) means that HSC disk time is crucial. Joel and Alan inquired whether the same operating system will work at 3.5 times the disk handling speed? Dave wondered whether we expect to achieve the performance improvement just by using HSC even though the operating system is the same?

Question — Joel Emer
Any attempt to monitor performance while the machine is running?

Suggestion — Bob Stewart, Dick Hustvedt
Want check of 780 performance on a real machine with a typical load (they volunteered such a machine).

Concern — Dave Rodgers
About compatibility mode performance (word processor software will probably always be 11 ISP).

Questions Answered

Dileep Bhandarkar
Meaning of RAMP goal of twice 780? -- 1/2 BMC.

Tom Eggers
What percent of logic checked for single errors? -- Checking is done where most errors occur rather than for some gross percentage of the logic.

Ken Okin
Why not parity on the VA bus? -- Bus is generated by logic that is difficult to do parity on, and the benefit was not felt to be worth the cost.

Ken Okin
What percentage of the MCA logic is for diagnosis? -- Very little.

Alan Kotok
How is the 10K logic checked? -- Enough IIL will be put in to get board isolation but the implementation is not settled.

Bill Strecker
Could plan ahead for larger cache via larger RAMs keeping the cache on one board? -- Cache has been increased to 16K (for future larger RAMs can simply swap boards).

Alan Kotok
How do we know that if we get the cache hit percentage, we get the expected performance? -- VOTE simulation.

PIPELINE (Al Helenius)

Action — Dave Rodgers
Should analyze the relative value of different ways of handling branches: wait, assume take branch, assume not take branch. This was looked at during the early design phase, but a thorough analysis has not been done.

Suggestion — Dave Rodgers
Want whatever technology is added to the design process to be published widely -- Will do.

Suggestion — Tom Eggers
Should indicate wherever pipeline design can be utilized to speed up software, eg a Fortran compiler interleaving the instructions from two computations to guarantee prefetching. This has not been investigated and is not an action item vis-a-vis the hardware; but such information should be made known to the software people.

Concern — Dave Rodgers
One address register seems to be a performance bottleneck -- This is not so simple.

Question — Al Ryder, et al
Questioned no D stream prefetch by I box while fetching string bytes. -- E Box runs the operation, and more efficient for I box to get string bytes rather than just sit idle throughout.

I BOX (Tom Knight)

Discussion — General
Considerable questioning about branching and need of I box to wait for go-ahead from E box instead of I box handling it alone. This seems to be necessary as I box must have information from E box (condition code or computed result) to continue.

Discussion — Bob Stewart, et al
General problem of escape mechanisms for two-byte op codes (no interpretation of specifiers). Alan asked: if the I box throws up its hands, can the E box say start again at third byte down? -- No.

Suggestion — Dave Rodgers
Recommend ECC backup for dispatch RAMs -- Not being done. This will allow us to service solid faults; retry will allow intermittent.

Questions Answered

Ken Okin
If DEC can add instructions to microcode, why not customer? -- He can and also add the hardware, but tools for doing this are not supported.

Dave Cane
How handle compatibility mode? -- Still open.

Alan Kotok
Clocking scheme? -- Will be published when designed.

Ken Okin
Array processing? -- Not enough memory bandwidth to support it well.

F BOX (Mike Brown)

Suggestion Bob Stewart
Subroutine hooks for long integer mul/div -- Bottom priority after all floating point. This led to:

Action Al Ryder, et al
Should be analysis of tradeoffs of integer mul/div vs H floating point. Suggestion that might sell FPA to nonfloating markets if integer mul/div is fast enough using FPA. (Eggers says most of decimal conversion is long multiply.)

Question Dileep Bhandarkar
If FPA fails in the middle, can you switch to the E box to finish? -- No. Instruction will be retried.

Suggestion Alan Kotok
Optimize F box for dominant format (probably G).

Suggestion Alan Kotok
Since multiply is so fast compared to divide, should investigate series expansion algorithm for divide using many multiplies.

Question Tom Eggers
Unbiased rounding? -- Tom was pleased to learn that Mike assumes it will be required by the architecture.

CONSOLE (Ed Anton)

Action Dick Hustvedt and Chuck Samuelson vs Jeff Barry
Problem of file structure for diagnostics. Dick and Chuck say floppy formats must be RT11 or FILES-11. Jeff says LSD cannot use RT11 with APT because it uses permanent files on the floppy.

Action — Al Ryder
What is the total boot time? Load microcode in ten seconds, but also run microdiagnostics, load VMS, etc. How long to get the system going? Also how long for a warm restart?

Action — Al Ryder, Dick Hustvedt
RX01→RX02 conflict. Use only half of RX02 so plenty of room left over.

Suggestion — Ranjit Singh
Must consider RS-232 Rev 1 communication format. In Europe current modems expected to be phased out by 1985.

Question — Ken Okin
Is the console going to be a superset of the midrange spec? -- Yes.

Question — Ken Okin
The go-fail chain has a definite adverse effect on APT. Have other microdiagnostic strategies been investigated?

M BOX (Bill Bruckert)

Action — Alan Kotok, et al
Even though larger translation buffer, one-way associative may lead to thrashing. Various schemes for saving previous translations should be investigated (thought to be especially useful for I stream).

Action — Alan Kotok, Dick Hustvedt
Alan is afraid decisions are based on too little data: eg is write allocation by block rather than word appropriate for the load, the real way the operating system works, etc. Dick says this data is available for a real load from the SPA machine with ninety users. Get info from Steve Forgey. -- Bigger cache should take care of this concern.

Action — Bob Stewart, Dick Hustvedt
Need to consider performance with repeaters for larger memory.

Suggestion — Dave Cane
Either the cache is too small or the backing store is too slow, maybe both. The CPU is over 3 x 780 with a memory system only 2.5 x 780, despite the fact that many people think the 780 memory was too slow for its CPU. A 10% miss rate costs about 2:1 in performance. There ought to be a relatively cheap way to buy back a lot of that. -- Going to bigger cache.

Suggestion Ken Okin
Strongly urge a modeling effort to see what benefits can be gained by doubling the cache size. It would seem a good tradeoff to improve the hit rate and greatly improve performance at a cost of two or three boards in a 15-20 board system. The better performance we can ring out of the design, the longer its life. I dont,believe in redesigning the machine every six months to improve it 10%, but a chance of 50 or 60% should be studied carefully.

Question Alan Kotok
On write error would VMS zap only jobs with interest in the data? -- No way to know who it was.

Question Ken Okin
Since the cache is writeback, it sems apparent that IO traffic will reduce the cache size as well as cause extra delay to the CPU due to contention. What effect does this have on performance?

Suggestion Ken Okin
8 MB is likely to be too small (Nebula will have 5.25). The maximum appears to be 6 + 8 + 8 = 24 MB, which may also be too small in 1985. (N.B. Repeaters will each handle 12 MB, maximum 30 MB.) -- Expansion is readily available, and by 1985 larger RAMs may also be here.

A BUS AND SBIA (Jim Lacy, Barry Flahive)

Action Dick Hustvedt
Someone must list all the restrictions relevant to the A bus and its adapters.

Suggestion Ken Okin
I disagree with only two DMA buffers for the SBI - there should be three (or four if possible). My rationale follows:

> DR780s should have their own A bus adapter. This allows the DR780 to do its worst to the SBI without worrying about other IO adapters.
>
> Given then that no other SBI will have any device that can utilize the entire memory port bandwidth, the old SBI assumptions hold: namely the silo in the adapter is mostly empty, and the ability to buffer extra requests reduces their ultimate latency rather than increasing it.

The problems in the 780 come from a saturated memory subsystem. By moving the DR780 and devices of their

ilk to their own A bus adapter, UBAs and MBAs will be quite happy on the SBI and will work as originally intended.

MINUTES OF VENUS CORPORATE REVIEW
29 February 1980

DISTRIBUTION

DAVE RODGERS	TW/C04	INTERCONNECT PROGRAM
ALAN KOTOK	ML3-5/H33	OFFICE OF TECHNOLOGY
DILEEP BHANDARKAR	TW/B05	ARCHITECTURE MANAGEMENT
DAVID ORBITS	ML3-2/E41	CORPORATE RESEARCH
CAL CALAMARI	TW/C04	INTERCONNECT
JOEL EMER	ML3-3/H24	SYSTEM PERFORMANCE ANALYSIS
ALLAN RYDER	MK1-1/D29	TELCO
DICK HUSTVEDT	TW/D08	VMS DEVELOPMENT
RANJIT SINGH	ML5-5/E97	DISTRIBUTED SYSTEMS
TOM EGGERS	TW/B05	VAX ARCHITECTURE
WILLIAM STRECKER	TW/B05	VAX ARCHITECTURE
KEN OKIN	TW/C03	NEBULA
LE NGUYEN	ML1-2/E65	SCORPIO
BOB ARMSTRONG	TW/D06	DMS
DAVID CANE	TW/D46	COMET
JIM THOMPSON	ML5-2/E50	MDC ENGINEERING
BOB STEWART	TW/C04	INTERCONNECT
DON HOOPER	MR1-2/E85	2080
*GORDON BELL	ML12-1/A51	OOD
*SAM FULLER	ML3-5/H33	OOT
*JOHN HOLZ	ML5-2/E50	PRODUCT LINES
*DUANE DICKHUP	ML1-2/E65	SMALL SYSTEMS
*STEVE ROTHMAN	TW/D06	COMET

Venus Personell

JOHN ALLEN	MR1-2/E47	JIM LACY	MR1-2/E18
BARRY FLAHIVE	MR1-2/E18	BILL ENGLISH	MR1-2/E47
CHUCK SAMUELSON	TW/D08	BOB ELKIND	MR1-2/E47
PAUL KELLEY	MR1-2/E18	WILLIAM BRUCKERT	MR1-2/E47
ALLAN HELENIUS	MR1-2/E47	PAUL GUGLIELMI	MR1-2/E47
MIKE BROWN	MR1-2/E47	TOM KNIGHT	MR1-2/E47
CLEM LIU	MR1-2/E47	ED ANTON	MR1-2/E47
AL DELLICICCHI	MR1-2/E47	JOHN GROSE	MR1-3/P74
FRANK ROBBINS	MR1-1/S35	MIKE ROBEY	MR1-1/S35
TONY VEZZA	MR1-2/E47	BILL HILLIARD	MR1-2/E47
MIKE FLYNN	MR1-2/E47	GEORGE HOFF	MR1-2/E47
TRYGGVE FOSSUM	MR1-2/E47	EILEEN SAMBERG	MR1-2/E47
JOHN DEROSA	MR1-2/E47	LARRY CORNELL	MR1-3/M90
BILL WALTON	MR1-2/E47	RAY BOUCHER	MR1-2/E47
VIC KU	MR1-2/E47	JEFF BARRY	MR1-2/E68
JUD LEONARD	MR1-2/E47	JOHN BLOEM	MR1-2/E18
ROGER LAWSON	MR1-2/E18	SAS DURVASULA	MR1-2/E47

*Not in Attendance.

Subject : Console Project Status

1.Ø What design is completed; What remains to be done?

Done:

* A Console subsystem with an RX floppy load device interface, four serial ASCII ports, a time of year Clock, a serial Diagnostics interface, an eight bit parallel path to the EBOX and miscellaneous support logic.

Remains:

* Replace the RX interface with a QBUS adapter, the implementation of a front end design to the HSC multidrop connection and design refinements to the remaining logic as the result of design errors and possible changes to control registers for ease of software execution.

* CPU front panel.

2.Ø What functional/timing dependencies on other boxes not documented or understood.

* Multiprocessor interaction, IPA , in regards to reboot of the Venus CPU, A BUS DEAD signal utilization.

* Time Of Year Clock battery low implementation. Current design provides no status indicator to logic at system power on time. requires on board detector logic mechanism.

* T11 cycle slip does not work on revision parts 2.1 parts. No problem with breadboard power on because of CSL clock source is not the final design item.

  The design goal is to use the maximum T11 clock source of 7.5 MHZ with a cycle slip on uart read access.

  Using a clock source of less than 7.Ø MHZ elimates the need óf a cycle slip in the current design.

* T11 2.1 parts also have a sensitivity problem in DMA contention with refresh and instruction execution with register modes 5,6 and 7. Hudson says that this problem is solved on this pass parts by increasing VCC to 5.5 volts.

* SDB usage by the BOXES has not been fully documented or understood by myself or Bob Petty. Current BOX SDB implementations have conflicts in shift direction of channel most significant bit.

* When the QBA, QBUS interface adapter, design is completed T11 IO address space might have to be redefined. This effect the CSL prom, address pal and possible decode logic.

* Existing layout and manual placement has resulted in some logic being extensively distributed over the board. Not sure of the results in critical timing paths.

3.0 Status of the Console Spec.

* The following changes must be made to the Console spec as the result of additional design and deficiencies:

1. QBUS adapter interface operation and description.

2. Possible changes to IO address space. This effectes entire spec.

3. Lack of functional use and electrical description of EMM and HSC interface.

4.0 Timing Analysis

1. QBUS design must be implemented and integrated into the current design.

2. Elimination of cycle slip to increase T11 increased performance with 7.5 MHZ clock source.

3. Critical timing paths were evaluated at the initial design but the system timing is not documented in one place.

5.0 Known Problems:

Some are mentioned above but additional logic required for the QBUS could result in a second pass relayout problem if CAD tools are not in place, TTL pincut vs manual this and that!

6.Ø Possible Functions Decommitted:

Does anyone know where HSC multidrop stands?
This week it`s :

LCG	YUP!
CX	Maybe!

7.Ø Major Risks with current design

* What`s required to detect TOY battery low?

* T11?? No Please!! May have to wait foe pass three parts. Pass three parts due August.

* Console connections to outside world, FCC.

* Console connections off the backplane.

* Not enough SDB channels????

SUBJ: Status of the SBIA Design -- May 1981

What design is completed, what remains to be done?

The SBIA design was frozen for layout in January 1981. At that time the diagnostic group had not completed their analysis of the logic. Meetings in February and March resulted in my agreeing to some defined functional changes in the design. In addition I agreed to look at providing visibility to some portions of the logic. There remains one major open diagnostic issue, testing of the PROMs. [see the detailed list at the end]

Also during February and March I discovered several bugs as a result of working on the specifications. [see the detailed list at the end]

The logic necessary to support interprocessor bootstrap over the CI has not been defined. This logic will presumably require only a few (two to six) gates to implement.

No logic fixes to implement the above changes have been designed.

The SBIA microcode for the SBI state machine has not been written. This microcode has been flowed, and Eileen Samberg has agreed to generate the code.

The patterns for address decode PROMs and function translation PROMs have not been defined.

The diagnostic design is not complete and there remains risk of additional functionality being required.

What functional/timing dependencies on other boxes are not documented or committed or understood?

The M Box timing of the A Bus signals has not been

completely specified--the A Bus spec contains "best guess" numbers (for some signals) from August 1980, which was before much of the M Box had been designed. (John Manton has been trying very hard to meet these numbers.)

The M Box stubbing of the A Bus has not been completely been defined--the A Bus spec contains "best guess" guidelines from August 1980.

The A Bus wire propagation times are not well understood. The simulations for the A Bus wires were run in February, March and April 1980. Until the M Box stubbing of the A Bus wires is defined new simulations would not be very useful. Similarly the maximum expansion of the A Bus should be reviewed--is four I/O Adapters with the A Bus stubs at one inch increments the correct maximum expansion?

The M Box has not specified, or totally committed to, support of the diagnostic functions necessary to diagnose the ECL port of the DC022 register file and its support logic (address generation, counter, read/write logic). This is a factor in the diagnostic design not being complete. [The diagnostic requirement document has not yet been published but most needs are more or less understood.]

What is the status of the specification?

The SBIA and A Bus specifications were updated in February and March. Both specs were released in April. These documents do not, in general, contain all the information which they eventually should contain (functional changes and visibility points must be documented; the A Bus timing and physical dimensions should be frozen).

The performance (maximum data rates, access times, etc.) for both the SBIA and the A Bus need lots of work--particularly multiple device performance.

What is the status of the timing analysis?

The timing analysis of the TTL logic was done at design time. The 11/780 timing rules were used: worst case DEC spec over temperature and voltage; no wire delays; 10 nS skew in general--slightly better for same phase and/or same driver. Detailed documentation was not generated.

Timing analysis of the 10K logic on-board paths was done at design time. Skew used was 2.5 nS but most on-board paths have margin. Again no formal documentation.

Timing analysis of A Bus signals is based on "best guess" numbers (i.e. the closest simulation to the situation which we then had). Depending upon clock system parameters

(position skew, width and growth/shrinkage) the A Bus may or may not work at 66.66 nS. The DCØ22 is further compounding the problem by slipping its specs in the wrong direction. A Bus timing assumptions are in the A Bus spec. DCØ22 timing is in the DCØ22 spec--new spec numbers for the DCØ22 are currently being negotiated.

What are the known problems with the current design including power margin, module density (ECO space), pinning etc? Potential solutions.

1. The biggest problem is the softness of the A Bus timing on a transfer of command/address or data from the M Box to the DCØ22 in the adapter. Quite literally every major piece of that transaction is incompletely specified or slipping in the wrong direction: the M Box data valid time; the A Bus wire propagation time; the DCØ22 address and data setup and hold; and the clock system which references the data valid time and produces the write pulse.

   One possible solution is to phase the SBIA slightly behind the M Box (about 1Ø nS). This would use excess in the SBIA-to-M Box transfer to make up the difference in the M Box-to-SBIA transfer. This would, however, present some real problems in the control lines of the A Bus (and might not even work). Given that the DCØ22 has already been pushed as far as we can push it, and the M Box can not be speeded up, the other alternatives come down to:

   a) a better clock;

   b) an alternative means of generating the write pulse (a tapped delay line?);

   c) slowing down the cycle time of the system (A Bus cycle equals CPU cycle);

   d) slowing down the memory to SBIA transfer by inserting a cycle in the M Box where the data waits so that it will be available at the start of the next cycle (I have no idea if this is possible).

2. Other A Bus signals are very tightly timed: the IOA REQUEST to IOA SELECT path (which is running at about 65 nS depending on the skew) is probably the tightest path after the M Box-to-SBIA data transfer. Clock skew is a major consideration in this path. This path starts on the SBIA, passes

across the A Bus to the M Box, and returns back across the A Bus to the SBIA in one cycle. There is no stubbing on the wires and the logic is already cut to the minimum number of levels. If this path can't make it we either lengthen the machine cycle or add one more pipeline cycle to the A Bus.

3. The next biggest problem, from a completeness standpoint, is a continuing discussion of the diagnostic coverage/support requirements and the reasonable cost bounds to test or isolate a circuit. This issue has taken many forms but one example is the major open issue: the question of whether it is necessary for the diagnostic to be able to verify every cell of PROMs. The cost of supporting verification of two 1k x4 PROMs used in the SBI State Machine appears to be in the five to ten chip range (all information seems to indicate that the grow-back problem only results when the parts are improperly processed--i.e. not blasted with the correct voltage/current/time).

   A clear definition of what classes of faults we are trying to detect and isolate and some cost constraint guideline would be helpful. (For example: shorts inside/outside chips; stuck at one/zero at internal nodes of SSI/MSI chips; chip processing problems--slow chip or PROM cell grow-back; partially functional chips--counter which counts 1,2,7,4,3,9,6,5...)

4. The chip count on both SBIA modules is very high (203 DIPs on SBA; 276 DIPs on SBS) but I do not see a problem. Both boards have been successfully routed by ALGOREX, and I believe that both placements could be greatly improved by PINCUT. Both of the current placements include spares (2 ECL and 3 TTL on SBA and 4 TTL on SBS).

5. The power dissipation on the SBIA modules is high. The power estimates are, in my opinion, overly conservative but even with less conservative numbers the SBIA modules dissipate a lot of power. The power dissipation is due in a large part to Schottky-TTL parts which do not have the noise margin problems of 10K (i.e. cooling them is less of a problem because wider temperature differences are acceptable). The bottom line is that I don't think that we have a serious problem on the boards.

   If it becomes necessary to reduce power dissipation some substitution of LS-TTL parts might be made (LS was used where I knew without a doubt that no timing or loading problem would be encountered).

Major redesign would be necessary for wide-spread use of LS parts. Another alternative might be the use of PALs (which would mean substantial redesign--if PALs will work in the SBIA).

6. I/O pins on the SBIA modules are not a real problem--about 15 spare pins on SBA and about 15 spare pins on SBS.

7. The pinning of the current layouts may be a problem for the backpanel. The SBIA design inherently contains constraints which make the intermodule pinning difficult (the A Bus is constrained by cooling to the "C" paddle; the SBI is densest in the "A" paddle; resulting in "A" + "B" on one module connecting to "B" + "C" on the other). ALGOREX did not take the backpanel pinning of these signals into account when they assigned the pins which resulted in a larger crossing effect than we had hoped for.

8. One issue which is only semi-closed is the SBI margin clocks question. The margin clocks were raised as a diagnostic problem to which most everyone said "through it out!". Manufacturing (Steve Moro) called and said "wait!", which I was doing anyway (so as not to disturb the layout process), and I'm still waiting. If this functionality is removed about ten chips would be saved on SBA.

9. The diagnostic isolation of faults in the SBIA logic is less than the module isolation which we originally set out to achieve. In many cases there will remain a probability that either board could contain the fault. This problem is most serious with control logic. I have few options on improving the isolation because pinning and chip count prevent the partitioning which I had originally hoped to make (and the current logic design would not support that partitioning now).

   The only other alternative would be installation of Serial Diagnostic Bus control and visibility channels, along with clock stop and step logic of some sort. This would increase chip count in the SBIA and add additional wires to the A Bus since the SDB channels for four adapters would have to be added. While this would provide a means of isolation it would mean a very complex scheme because the diagnostic would have to step both the SBI clock and the Venus clock.

What functions in the design could possibly be de-committed to relieve problems?

1. SBI margin clocks (almost gone), about ten chips on SBA.

2. DMA Error registers which store the Command/Address and ID from the SBI--these are useful RAMP features which may not be used sufficiently to justify their cost (about ten chips on SBS).

3. One of the DMA Buffers--likely one of the general purpose buffers. This would save no data path logic but would save about 10 chips of control logic on SBA. This would reduce the maximum throughput which the SBIA can support from 13 Mb/s to about 10 Mb/s--the system effects could probably only be predicted by simulation.

4. It may be possible, with some performance penalty, to reduce chip count by the use of PALs. This would be a large redesign and would possibly present diagnostic problems if every cell must be checked.

What are the major risks with the current design?

1. The mixed technology layout of SBA is at least a schedule risk. I believe that the module can be placed and routed but it is not a "push the button" process because of the many rules including ECL-TTL separation, differential pairs, cooling restrictions, pinning restrictions (the one thing which the ALGOREX layouts do not take into account is the SBS-SBA backpanel pinning), power plane split, etc.

   The tools are not, today, in good shape to support the second pass layouts for the SBIA. TTL Pincut will require libraries to be completed, and there are several features of PINCUT which do not adapt well to use on TTL designs. (One example is load balancing--TTL loads have three loading characteristics which are important in balancing: high current; low current; and capacitance.) (Other PINCUT issues are: multiple chip sizes; areas restricted for only ECL chips or only TTL chips; etch length restrictions.) Routing and checking of ECL-TTL separation rules is an entirely manual process--we don't even have a way to highlight the runs of one technology for identification.

2. The system performance of the I/O-Memory subsystems is not well understood. The simulations were never completed and the M Box has changed substantially since the original definitions of the A Bus (M Box stops DMA traffic for CPU traffic on the A Bus rather than interleaving it; CPU traffic has a higher priority than DMA traffic on the A Bus rather than a lower priority; the M Box stops DMA traffic and uses the A Bus for internal M Box transfers when performing some register reads; the M Box does not provide an opportunity for DMA to start between the write-back and refill for a CPU cache miss as originally specified).

Any other information appropriate to clarify what must be done to get the final design.

The A Bus Tester can be made to provide a nearly complete functional checkout of the SBIA. The tester will however be subject to assumptions about the M Box operation and will not (as currently defined) support multiple SBIAs. Detailed functional checks may require some additional logic in the tester. Some moderately sophisticated software will be necessary to provoke some of the more subtle cases.

The tester has two major shortcomings: assumptions about how the M Box will work (made by the same person who designed the SBIA logic interface--me) may not show up actual interface problems; and the tester will not be able to prove out the SBIA diagnostics and thus the usefulness of the diagnostic functionality. Minor shortcomings of the tester include: inability of the tester to exactly duplicate the M Box timing--I believe that we can get the tester to run at a 66.66 nS cycle; and the resources that will go into the design and programming of the tester.

On the whole I believe that the tester can be a useful tool for the debug of the SBIA logic. It will be necessary, though, to manage the tester effort to prevent getting caught up in diminishing returns.

One aside issue is the question of the new clock system. What clock do we want built into the breadboard? If the new clock differs in mechanical profile from the three 16-pin DIPs and three 7-pin SIPs now on the SBA module installing a new circuit may be difficult without some relayout.

Where do we go from here?

I would like to:

1. Finish the design by fixing all known bugs, closing the open issues, generating the microcode, and generating the decode and translator PROMs.

2. Get the diagnostic design completed to reduce exposure to new feature requirements. M Box help needed here.

3. Design and build the A Bus tester.

4. Conduct a rigorous review of the SBIA. I would prefer a one-on-one with someone familiar with the SBI so we can get down to the gate level.

5. Start a effort to define the A Bus wire timing. This will require running SPICE simulations of many combinations of length, stubbing, loading, etc.

Known Bugs and Open Issues and Undesigned Logic

NOTE: some of these were mentioned above, and some general things mentioned above were not included in this list.

1. ISR, QUADCLEAR, and UNJAM not blocked by "Disable SBI Cycles" bit. see also items 7, 14 and 22

   FIX: redesign CPU address evaluation logic on sheet SBSR

2. There are both ECL and TTL versions of "DMAx CMD IN PROG" Since one is asserted high and one low there is no signal ambiguity but confusion can result.

   FIX: rename signals

   SBA5 DMAA CMD IN PROG L --to-- SBA5 DMAA ABUS CMD L

   SBA6 DMAB CMD IN PROG L --to-- SBA6 DMAB ABUS CMD L

   SBA7 DMAC CMD IN PROG L --to-- SBA7 DMAC ABUS CMD L

3. The gate which produces the signal DMAx LOCK WRITE H is redundant.

   FIX: remove the 3 gates and change the signal names

   SBAC DMAA LOCK H becomes SBAC DMAA LOCK WRITE H

   SBAD DMAB LOCK H becomes SBAD DMAB LOCK WRITE H

   SBAE DMAC LOCK H becomes SBAE DMAC LOCK WRITE H

4. SBSB FAULT INTR H has the term SBSB FAULT SILO LOCK in it--should have SBS4 FAULT LATCH H.

5. SBS4 INTLK SEQ FAULT H has the wrong flavor of SBS6 INTLK FF. Replace SBS6 INTLK FF L signal with -SBS6 INTLK FF L.

6. SBAN REC SBI DEAD H appears to have been inadvertently wired into some circuits which should have gotten ?init?? -- need to figure out what happened.

7. Address bits <26:23> are not fourced to zero when a CPU transaction is within the adapter's address range.

   FIX: redesign the CPU address decode logic on

sheet SBSR (see items 1, 14 and 22)

8. The DEC 5433 JFET has a pinning error in the library. ALGOREX has fixed the device for the layout--we must fix the library and the print set. (logic sheet SBAN)

9. On sheet SBSK a 6330 PROM was used (6330 is open collector)--should have used the 6331 tri-state PROM.

10. Wrong SBI TAG is generated on an SBI QUADCLEAR command/address transaction.

    FIX: add logic to generate correct TAG code

11. DC102 input high current is wrong in the library--this results in a loading problem in several circuits.

    FIX: place a pull-up resistor to +5.0 volts on the overloaded lines--value and lines --tbd--.

12. SBI margin clock oscillators have been recommended for removal.

13. Parity invert for register data is unnecessary.

14. Current concept for disabling SBI functions cannot test disabling of DMA traffic since CPU traffic to the SBI is also disabled--look at an override bit for CPU traffic.

    FIX: is part of CPU address decode redesign (see items 1, 7 and 22)

15. On sheet SBSH the signals SBSH DMAx ERR LOCK L cut off the signal which set the 74S112 before the minimum pulse at the preset pin has been reached--i.e this is a race condition.

    FIX: change circuit to eliminate the race (several options open)

16. The signal BUS SBI ALERT L (sheet SBAN) cannot be forced--look at ways to force this signal.

17. There is no visibility that the SBIA is receiving clocks from the CPU if the SBIA is dead.

    FIX: (this does apply to the current clock system but may not be sufficient for the new system) place two LEDs on the SBA module to indicate that the clocks for the A Bus is being received. Two LEDs

are necessary to indicate that the clock is toggleing.

18. In order to diagnose the DCØ22 TTL data write circuits for all cells it is necessary to generate arbitrary data to both the WD1 and WD2 locations--this can be done by a special diagnostic function which will cause a quadclear to transmit the command/address three times (with a couple of bits forced to ones on the second and third transfer) instead of command/address followed by two all zero words.

19. In order to provide visibility to the BUS SBI DEAD L wire a flip-flop or register must be chosen and SBAN REC SBI DEAD H placed in the place of one of the INIT signals--question as to the correct circuit for INIT generation.

2Ø. The signal BUS SBI INTLK L is to be removed from the SBI SILO since the signal no longer has meaning.

21. Logic to request the A Bus for the DMAI (interlock read) buffer needs review to determine if the "count=1" circuits can be removed without performance degradation. M Box operation may be a factor in this decision.

22. Remove slot code jumpers from the CPU address decode logic. See also items 1,7 and 14.

23. Signal names SBA8 ABUS INTLK FF L and SBS6 INTLK FF H(L) have been said to be confusing. Consider change to SBS6 SBI INTLK FF H(L).

The following is the status of the E Box control store and data path modules (CSA, CSB, EDP):

a) What design is done (SUDS)?

CSA: This module is completed and raw boards have been received from National Bureau of Engraving.

CSB: This module is completed and raw boards are due in.

EDP: The logic drawings and manual placement are done for this board. It is in placement (PINCUT).

All MCA's for the above modules are done. The list includes:

MIC: microsequencer
BEN: branch enable
ALU: data path alu slice
DPC: data path controller
SCE: shifter control
SHF: shifter slice
ADD: GPR, scratchpad addressing
CCD: condition code logic
CLA: carry look ahead, Bmux conditions
PDP: parity logic

b) What dependencies on other boxes?

WBUS ABORT function is not solid at this time.

c) Status of box specifications:

Micro sequencer/control store:

Microl.mem describes the usequencer/control store hardware. This document is out of date with respect to console interface and current timing (T3 data latches). The current console interface has been described in detail in a separate document, CS.mem.

E Box Data Path:

A specification exists which describes the data path from a microcoders point of view, but, it does not qualify as a hardware spec.

d) Status of timing analysis:

A hand calculated timing analysis has been done on the control store and data path.

e) What are known problems/solutions with current design?

Control Store:

prob.: Some uword bits are needed earlier.

solu.: Can add more address buffers to decrease address line loading and speed up the bits.

prob.: Additional copies of some uword bits are needed.
solu.: Add data buffers for those bits to get more copies.

prob.: Clock system on CSB is two "clock systems" old.
solu.: Wait for new clock system.

prob.: Ustack data latch is not stalled.
solu.: Add wire.

prob.: Need a way to insert ustack parity error.
solu.: Add a wire.

prob.: Ustack parity error signal could be faster.
solu.: Change 10160 to 10113 if timing shows this signal needs to be faster.

EDP module:

prob.: Muxing of control into ALU slices is a problem. Data waits for stable control information. Clocks make this scheme tough to fix.
MCA REDESIGN solu.: Get rid of the muxing.

prob.: Internal clock generation in ADD MCA susceptible to clock skew on external inputs.
MCA REDESIGN solu.: Clean up when know what new clocks look like.

prob.: GPR read data is slow.
" solu.: Speed up uword bits, increase pulldowns on addressing path in ADD MCA.

prob.: Clock problem in SCE MCA...could have two latches which loop on each other open at the same time.
"? solu.: Wait and see what new clocks look like (how well can I control the new clocks).

prob.: Timing problem turning EDP ERROR around into EWBUS ABORT signal.
| solu.: All boxes have to analize abort timing.

prob.: Module has a lot of tight clocks with skew problems.
solu.: Using 2080 style clock routing should eliminate many of the clocks (adjustments). This means the IDEA router must not re-route runs which are manually routed to particular specifications. This is a definite problem today.

prob.: All mutiplexed lines are tight.
solu.: Have to handle individual cases.

prob.: Multiplexing lines between DPC and PDP with valid data on every phase of the clock (60MHZ) must work or PDP is blown out of the water.

prob.: Module currently uses 235 pins, 156 watts, about 150 square inches with 4 spare dip locations.
solu.: None for the pins. Power and space will go down if move some of the 10K parts back into MCA's and eliminate some of the clocks.

f) What functions could be decommitted to relieve problems?

Answer to this requires negotiating with microcoders.

g) What are major risks with current design?

Mostly living with the current clock system and making the multiplexing of signals work in the data path.

h) Any other information:

*Need the time to use the available tools (PINCUT) to accomplish the desired placement for EDP. This is not a push button operation.

*Need to convert CSA, CSB module prints to latest SUDS library so can enter in PINCUT to get LDIF file for DLYED program.

*Possibility exists that CSA and CSB could be compressed to two copies of one board type by converting usequencer to 10K and moving to one of Jim Lacy's modules. Need time to analize requirements if anyone is interested in this.

STATUS OF E BOX CONTROL LOGIC

Date: 20*MAY*81

Designer: James Lacy

Modules: EBC (L0210)
EBD (L0211)
EBE (L0219)

Status:

1. What design is completed (SUDS prints available), what remains to be done?

   * E Box W Bus Registers (NICR, ICR, ICCS, CBUS/INT, PSL, IBE, EBCS): 90% complete (updates need to be made to EBE (L0219)).

   * E Box Context Logic (WBUS CTX, EBOX MEM CTX, EDP CTX): 100% complete.

   * uMCF Decode Logic: 90% complete for decodes going to E Box control; 50% complete for decodes going to E Box port logic inherited from M Box.

   * Cycle in Progress Flags: 90% complete (last 10% is dependent on design of E Box port logic inherited from M Box).

   * Port Status Logic: 95% complete (last 5% allocated to potential use of demultiplexing functions provided by this logic to reduce M Box to E Box module interconnects).

   * E Box Microtrap Logic (conditioning inputs, prioritization, request and vector generation): 90% complete (last 10% involves changes to handle I Box errors differently); possibility of more redesign to correct timing problems associated with inputs (i.e., STRING VALID and OP VALID from I Box).

   * E Box Stall Logic: 100% functional design, but timing problems associated with inputs (i.e., STRING VALID and OP VALID) and destinations (I Box valid time requirements for creating ISTALL) will require repartitioning of E Box and I Box control logic and/or design changes.

   * E Box Abort Logic: I have a design which must be tested against the requirements of the other boxes.

- E Box Microsequencer Branch Conditions from E Box Control: 100% complete for conditions from E Box Control logic; 0% complete for conditions from E Box port logic inherited from M Box; ?% complete for diagnostic branch conditions.

- Interrupt Logic (Internal, External, IPR/IPL Comparison): 100% complete.

- Diagnostic Hooks: Where the need for diagnostic hooks has been identified, they have been incorporated into the design; however, there is considerable instability in diagnostic requirements and little analysis has been done to determine what hooks are required.

2. What functional/timing dependencies on other boxes are not documented or committed or understood?

   - Dependencies within the E Box are pretty well understood, but not formally documented.

   - Dependencies between E Box Control and the M Box are understood pretty well; interface signals and their timing are specified in SYS.MEM which is kept up to date in this respect.

   - Dependencies between E Box Control and the I Box are not well understood; they are discovered as design in both boxes proceeds and solutions can be difficult to find. The I Box Specification is reported to be sufficiently out of date to make it an unreliable document to which to design and it does not contain signal timing information.

   - Dependencies between E Box Control and the F Box are understood pretty well, but the current F Box Specification does not reflect the current interface. Signal timing between E Box Control and the F Box is not formally documented and may have some problems.

   - Dependencies between E Box Control and the SBI Adapter are well understood; interface signals and their timing are specified in the A Bus Specification.

3. What is the status of the specification for the box?

   There is no specification which ties together the whole of E Box Control or specifies its interface with the rest of the machine. Approximately half of E Box control and/or its interface to other parts of the

machine is described in parts of other documents, including:

W Bus Register Specification
Interrupts, Exceptions, and Microtraps Specification
Stalls Document
W Bus Specification
Aborts Notes
SYS.MEM
I Box Specification

4. What is the status of the timing analysis of the box?

Those paths which are perceived to be most critical have been analyzed. A complete timing analysis is futile until the design of E Box Control and the other boxes to which it interfaces is more mature so that data valid times can be assigned to the interface signals between boxes.

5. What are the known problems with the current design including power margin, module density (ECO space), pinning, etc.? Potential Solutions.

* E Box and I Box stall logic have timing problems which will necessitate repartitioning and/or redesign. We will better understand how to solve the timing problems when a functional design of the I Box Control logic is complete and the timing of both E Box and I Box Control logic is analyzed.

* STRING VALID and OP VALID are too late to meet timing requirements of the E Box Stall and Microtrap logic. If this problem cannot be solved in the I Box, design changes must be made in the E Box.

* Some redesign of the E Box Stall and Microtrap logic is required to handle I Box errors differently.

* Aborts are still not well enough understood by the box implementors for me to be confident that I have a design which meets the requirements of the other boxes and is acceptable to their respective designers.

* Diagnostics will need more hooks into the E Box Control logic than currently exist, but it is not clear at this time what those hooks will be.

6. What functions in the design could possibly be

de*committed to relieve problems?

- Eliminate aborts. The penalty is a small loss in the number of errors that could be recovered from.

- Handle I Box Errors the same way E Box Errors are handled, namely that any I Box Error will cause an E Box microtrap when it occurs, instead of only causing an error microtrap if the E Box attempts to consume information from the I Box which is in error. The penalty is reporting some hardware errors which do not affect the data integrity of the machine.

- I believe that by working more closely with the E Box microcoders, we could identify some hardware functions which could be more easily implemented in microcode. We might also find some simple changes in the microword encodings which simplify the hardware design with little or no impact on existing microcode routines. The uLTCHST bit, for example, is not used by microcode; it might be put to better use as an indication of the start of an instruction, something which is messy to figure out in hardware.

7. What are the major risks with the current design?

- Module Density. The E Box control modules will be inheriting an indeterminate amount of logic from the M Box and I Box. Although these modules are well within the power, component density, and pinning constraints, the additional logic could cause problems in this area.

- Missing functionality. As design continues we may find that the E Box Control logic is lacking some function to support the I Box, memory management, or whatever.

8. Any other information appropriate to clarify what must be done to get to the final design.

I believe that better documentation of this machine in the following areas would help reduce our exposure to design and implementation errors:

Comprehensive description of the pipe
Update existing functional specs
Description of E Box Control (similar in format to E Box Microsequencer Spec)
Signal description list with data valid times for all signals going between boxes

The following is an appraisal of the current state of the E*Box data path design, roughly following the format of section 1.0 of the "Design Completion/Review Process" memorandum.

1.0 DESIGN COMPLETION

A first*pass, less*than*optimal data path design is complete in the sense that a full set of logic gates has been implemented.

2.0 FUNCTIONAL/TIMING DEPENDENCIES

Practically speaking, none of the timing dependencies between the data path and the rest of the machine are adequately resolved or specified. Because there is no common set of rules for analysing and/or specifying timing parameters, generating meaningful timing numbers for use by others is difficult (or marginally useful) at best.

On a broader level, the task of surfacing general timing dependencies is approximately 80%*90% complete. On a more detailed level, these dependencies simply cannot be adequately resolved or specified with the current inconsistencies in timing techniques.

3.0 E*BOX DATA PATH SPECIFICATION STATUS

The spec for the data path is approximately 40%*50% complete, with some errors and inconsistencies. Four to six man*weeks would be required to update and complete the spec. The following is a partial list of the missing parts of the document:

1. Condition code setting functionality.
2. Parity checking functionality.
3. Basic memory reference example.
4. Basic W*Bus interface functionality and example.
5. Description of the hardware implementation. (Spec is currently a microcoder's users' guide.) This would include a description of each MCA type; and a description of the specific problems, algorithms, solutions, and dependencies involved with each MCA. This is important if any of the MCAs must be re*designed in the future, so that the knowledge acquired in the first*pass design process is not lost.
6. Timing information on internal and external signals and busses.
7. Hardware signal interface spec between data path and the rest of the world.
8. List of microcode restrictions and/or dependencies.
9. Specification of clock adjustment procedure, clock phase information, etc.

4.0 TIMING ANALYSIS

There is no standard means by which timing analysis is done throughout the machine. Therefore no useful detailed timing information is available on signals between the E*Box data path board and the rest of the machine.

Establishment of common timing analysis rules is a high priority task for the Venus group.

5.0 KNOWN PROBLEMS

Pin count, parts count, internal timing, and parts placement (for reduced skewing and etch length) are all known problem areas in the E*Box data path.

There are currently three MCA types (ALU, DPC, and SHF or SCE) in the data path which need to be redesigned, because of known timing or logic bugs within them.

6.0 POSSIBLE FUNCTIONALITY REDUCTIONS

Reduced functionality will either impact the microcode design or the machine's RAMP goals; and currently both sets of functionality are highly utilised. Possible reductions are as follows:

1. Removal of decimal arithmetic assist logic in the E*Box ALU path.

2. Reduction of shifter functionality from 63*place to 31*place rotate.

3. Reduction of the number of parity error checks within the PDP MCA.

4. Reduction of the resolution of the parity checking within the PDP MCA from byte to longword.

5. Reduction of the amount of parity error information available from the PDP MCA.

6. Reduction of data path speed, which would presumably affect the machine cycle time.

7. Reduction of the scratchpad/GPR addressing functionality.

7.0 MAJOR RISKS IN CURRENT DESIGN

The single greatest risk is that the various pieces of logic on the data path module do not work with each other, or with the rest of the system.

The second greatest risk is that the timing of the data path has not been analysed with rules that are common to the entire machine.

The third greatest risk is that a clock system will be proposed that allows less (or different) functionality than the current clock system, especially with respect to non-standard clock widths (used in multiplexing applications and in the scratchpad read/write/addressing path).

8.0 MISCELLANEOUS

The tasks which will get us to our final design are as follows:

1. Specification of universal timing rules for delay analysis. Accuracy is unimportant relative to consistency and ease of use.

2. System-wide design review, from the top down, with all project menbers attending.

3. Update the functional spec to include the results of the system-wide review.

4. Redesigning the parts we know are incorrect of risky.

5. Box-wide design review from the top down, with all box members attending.

6. Update the functional spec to include the results of the box-wide review.

7. Gate-level review of the implementation, with two designers understanding and checking each page.

8. Update the functional spec to include the results of the gate-level review.

9. Path-level simulation, which would tie together all the MCAs on the E-Box data path module as well as the control logic associated with it. Care should be taken that any assumptions made about interfaces should be signed off by all the designers concerned with that interface.

10. Re-design of the problem areas, with re-simulation and re-review.

11. Parts layout of the final design on a module, taking into account the timing analysis that has been done at each review point.

12. Incorporating the timing analysis information into the E-Box data path functional spec.

13. Are diagnostics in place?

14. Is manufacturing following the action?

15. Breadboard build and debug.

Ø.1 I-box Redesign Process

1. Do an interconnect study of the 3 modules and the related MCAs.

    1. Set a top level goal for the partitioning

        1. Data path module to include adrs data path MCAs, GPRs, and w-bus interconnect.

        2. Byte Buffer module to include prefetch buffer and control, native mode DRAM and compatibility mode DRAM.

        3. Control module to include micro machine, stall logic, memory interface and miscellaneous functions.

    2. Draw an interconnect diagram showing the relationship of logic in MCAs to where it is used. This will identify what logic wants to go where.

2. Repartition the logic within the MCAs based on where it it is needed.

3. Design/Redesign about 8 MCAs.

4. Partition and complete design of 3 to 4 modules.

Ø.2 Assumptions

1. This approach assumes that it is O.K. to redo existing MCAs for better partitioning.

2. This should solve the over the top connector problem.

3. This should reduce the 1Øk parts count.

Ø.3 Risks

1. ICL is currently one and three quarters modules full of logic. It is not reasonable to assume we can remove this much overflow by better partitioning. We should provide space for this spillage within the machine.

Ø.4 Comments

I have looked at way to remove functionallity for performance and other items without success. The 1Øk control logic is the portion which has been underestimated. The cause for a large amount of the control logic has to do with a tradeoff made on the basic pipeline structure of the machine. The trade is as follows; If we make the depth of the pipeline short, the impact of changes in flow (branches) will be reduced to the point that a single instruction prefetch unit will suffice. In order to keep the depth short, the instruction decode and address calculation is done within the same micro cycle or pipeline stage. (This is done on only the larger of the IBM systems) This means that the hardware must perform many functions based on the contents of the Instruction Register. The alternative to this would be to decode the IR and dispatch the I-box micro machine to a routine to do the address calculation and perform the functions in microcode.

- What design is completed, what remains to be done?

- What functional/timing dependencies on other boxes are not documented or committed or understood?

90+% of the Ibox control logic is completed and documented in SUDS drawings. The following is a list of tasks being worked on:

1. Ibox micromachine 64 way branch encode logic.

2. Ibox diagnostic modes. It is envisioned that there will be several Ibox diagnostic modes, each of which will enable only the portions of Ibox under test. Waiting for inputs from diagnostic group.

3. Miscellaneous diagnostic hooks. Quite painful at this stage.

4. Ibox reset issue. Need to define how Ibox (at least the control logic) resets during power up, flushes and in diagnostic mode. Currently the two other Ibox modules IDP and IBD have little or no notion of a reset. I also know very little about how the rest of the machine is reset.

5. Ibox flushes, aborts and cancels. Partially implemented. I have a luke warm feeling the present definition will be the final one.

6. IB WRT issues. Need to finish implementing cancelling an IB WRT and generating signals to produce an IB ACK.

7. Ibox flags to Ebox. These flags are implemented to perform handshake with the Ebox. Need to qualify them somemore to guarantee they don't have errors associated with them. This is necessary to ensure Ebox will utrap for the appropriate Ibox error.

8. When Ebox asks for an operand Ibox needs to provide a signal to indicate the expected source of an unavailable operand. Ebox will use this signal to look at the appropriate stautus code.

9. Need to decode IBUFF word for miscellaneous control signals.

- What is the status of the spec for the box?

Ibox spec is dismally out of date, especially at the box boundary (interface with other boxes) level. The microcode group should also update the DRAM definition and constraint

file.

A basic problem of our specs is that different portions are written for different readers. There is a wide range of granularities which end up pleasing nobody.

Speaking for hardware designers, I believe there should be a place where all the box level interface signals (module level will even be better) are listed and described. The description must include the exact derivation of each signal, the projected usage of the signal and (heavens forbid) valid times.

The lack of this information is prohibiting one designer from understanding (and appreciating or criticizing as the case maybe) others' designs. The end result is very few people know anything beyond their own small piece of design.

The accumulation of this information is actually the missing step in our current design process. If only we had done this before drawing gates...

- what is the status of the timing analysis of the box?

Timing analysis has only been done for what the designers believe to be the critical paths, and there are a few:

1. IBUF bytes to DRAM access and latch time.

2. Ibox micromachine branch conditions to control store access and latch.

3. Various inputs into the three input adder to IVA.

4. Ebox stall to Ibox stall.

5. MD RESP to ISTRING VALID to Ebox stall logic. This is just one example of a signal crossing more than one module boundary in order to generate another critical signal.

- What are the known problems with the current design including power margin, module density (ECO space), pinning etc? potential solutions.

- What functions in the design could possibly be de-committed to relieve problems?

Among the three Ibox modules there currently exist severe space and pinning problems. These problems originated from poor upfront partitioning and estimation of 10k logic. Hind

sight tells us a major re-partitioning and MCA-packing can significantly alleviate both problems.

Few, if any, functions can be trimmed in the Ibox. What will help, however, is a relaxation in cycle time requirement.

- What are the major risks with the current design?

- Any other information appropriate to clarify what must be done to get the final design.

There is an inconsistency in design techniques throughout the machine. We don't have someone who carries enough respect (or a big enough club) to arbitrate differences and enforce consistency.

Speaking for the Ibox, we have been short-handed for quite awhile. Speaking for myself, I'm dead tired of drawing gates which should have been drawn by someone else.

MBOX STATUS - 5/20/81

1.0 BOARD STATUS SUMMARY

1.1 L204 MCD (Data Path Module).

1. Component status:- MCA's = 7, 10K parts = 109, RAM's = 44.
2. Density in estimated sq. in. = 183.
3. Current etched board completed, no disconnects.
4. Spare DIP locations = 5.
5. Power = 146W, -2.00V = 13.8A, -5.2V = 22.6A.
6. Pins used = 198/244.

1.2 L205 MAP (Address Path Module).

1. Component status:- MCA's = 9, 10K parts = 113, RAM's = 58.
2. Density in estimated sq. in. = 210.
3. Current etched board completed, 200 disconnects.
4. Spare DIP locations = 0.
5. Power = 159W, -2.00V = 13.7A, -5.2V = 25.4A.
6. Pins used = 225/244.

1.3 L220 MCC (Control Module).

1. Component status:- MCA's = 9, 10K parts = 146 (estimated), RAM's = 23.
2. Density in estimated sq. in. = 197.

3. Wire wrapped modlue, 50% completed.

4. Spare DIP locations, probably 0.

5. Power = Estimated similar to MAP module.

6. Pins used = 234/244.

2.0 DESCRIPTION OF CURRENT BOARD DESIGN.

2.1 MCD Data Path Module

The data path module is relatively solid with the design completed.

2.1.1 Known Necessary Changes. -

1. New clock system.

2. Abort and flush logic to be changed.

3. Add two rotation signals for the fast byte parity path. Small DIP change.

2.1.2 Posible Changes For Logic And Micro Code Simplification. -

1. Change of data path MCA, add another latch to hold ABUS data, this will simplify operation. Use another clock for the parity error register, this will simplify byte write operation. Must evaluate feasability and pay back of change.

2.2 L205 MAP (address Module)

This module needs to have an equivalent of 36 DIP's removed before it is routable.

2.2.1 Known Neccessary Changes. -

1. New clocking system.
2. Correct abort and flush logic.
3. Change to word counter in WVP MCA.

2.2.2 Possible Changes For Logic Compression, And Simplification. -

1. Remove syndrome decode and correction logic from address MCA.
2. Remove redundant copies of address drivers from address MCA.
3. Simplify control of PA MUX latch, i.e. separate PA HOLD signal.
4. Expand existing parity checking and generation circuits inside address MCA, and MATCH MCA. This should include TB parity generation, ABUS parity generate and check, and PA address parity generation for cache tag and inclusion in ECC check bit generation.
5. Remove duplicate valid bits from cache tag. Only the written bit would be duplicated. This would mean that a parity error on the valid bits would be unrecoverable if the written bit was set. This would simplify the WVP MCA, and remove the need for a tag correction cycle thus simplifying micro code. Approximately 10 DIP's would be saved.
6. Design new MCA for control and status registers and mux's that would interface to the register bus. This MCA could be used in a nuber of places.
7. Some existing 10K logic can be simplified.

2.3 L220 MCC (control Card).

The control has all the MCA's completed except for the PPR MCA, some of the 10K logic is completed (20%). This board is the most complex of the three and considerable effort is needed to complete the design and understand whether all functionality has been included.

The pin count is high and care must be taken to ensure adequate spares are available for ECO's. The DIP count is too high and the logic must be simplified and compressed into MCA's to ensure routability and sufficient spare DIP locations for ECO's.

2.3.1 Item's Left To Be Completed. -

1. EBOX request queue logic, located on an EBOX module.
2. PPR MCA needs 20% of logic to be completed.
3. Support logic for the processor port interface.
4. Cycle abort and flush logic.
5. Clock logic.
6. Interrupt logic.
7. Error handling and checking logic.
8. Branch logic for micro sequencer.
9. ABUS control support logic.
10. Control and Status register logic.
11. Miscellaneous array control logic.
12. ABUS diagnostic features.

2.3.2 Known Neccessary Changes -

1. Micro sequencer MCA - add mux for arbitrator address and next micro address.
2. MMD MCA - Logic change to ensure control store can be written correctly.
3. ABUS MCA - Change for correct polarity of Abus address control lines and ABUS IOA select lines.

2.3.3 Possible Cnanges For Logic Compression And Simplification. -

4. Create two more MCA's to ensure logic will fit the module with a number of spare DIP locations.

3.0 MBOX MICRO CODE.

The MBOX micro code is a crucial part of the design and has an effect on the hardware and control word definition.

3.1 Items Completed.

A skeleton state diagram flow for all the MBOX operations have been completed and the main CP operations and some ABUS operations have been intrenally reviewed. This has defined the main branch conditions. The micro word has been defined together with some macro's and is documented in mbox.mcr.

3.2 Items Incomplete.

1. Review of remaining ABUS skeleton flows.
2. Write first pass micro code.
3. Revise micro word to simplify logic.
4. Revise micro code as hardware proceeds.
5. Revise branch conditions as hardware proceeds.

4.0 MBOX TIMING

In order to complete the timing in a thorough manner and have confidence in the design the following items must be done.

1. Understand the new clocking scheme and its parameters.
2. Perform a desk timing calculation on the design.
3. Complete detailed timing charts.

4. Understand the critical timing paths of the MBOX and how they relate to other boxes.

5. Use the latest technology group timing numbers.

6. Detailed timing analysis.

Where are we in the MBOX?

Hand drawn timing charts have been completed on the major CP and ABUS operations. These have been done to understand the relation between the micro sequencer control and hardware control. They are very usefull in understanding the MBOX operation. These charts must be reviewed and more detailed work performed.

5.0 MBOX SPECIFICATIONS.

There are two main specifications defining the details of how the MBOX works and how it relates to the EBOX and IBOX. These specifications are MBOX.MEM, which describes the functionality and operation of the mbox, and SYS.MEM, which descibes the system functions and interface effecting the interaction of the MBOX with the EBOX and IBOX.

5.1 MBOX.MEM

5.1.1 Status Of MBOX.MEM -

The spec. is very complete in its description of the cache operation, hardware registers, error handling, MBOX - ABUS interface, MBOX - ARRAY BUS interface, CP cycle description, ABUS cycle description. There are some details on the micro sequencer description and diagnostic features.

5.1.2 What Is Not In The Spec. But Should Be. -

1. Better description of hardware register bit funcionality.

2. Update and complete description of MBOX error handling.

3. Added detail on MBOX - ABUS interface with complete timing information on all signals.

4. Complete timing information on signals interfacing between the MBOX and array cards.

5. Ensure signal names listed are all correct and consistent.

6. Ensure CP cycle description agrees with micro code flows.

7. Ensure ABUS cycle description agrees with micro code flows.

8. Add details on the micro sequencer operation and how the control store is written and read.

9. All the ABUS diagnostic features and their operation has to be described.

10. A signal list and termination chart for all the MBOX interfaces should be inserted in the appendix.

5.2 SYS.MEM

5.2.1 Status Of SYS.MEM -

This document describes an overview of each port in the Venus processor, the trap and interrupt operation, memory management operation and the requeue process, the signals and timing of the interface, and how the MBOX is to respond to the various port memory control fields.

The document is felt to be in very good shape.

5.2.2 What Is Not In The Spec. And Should Be. -

1. More detailed description of the memory management operations.

2. Added description of how the MBOX treats unaligned references, second references, and write across page boundary conditions.

+-+-+-+-+-+-+-+
|D|I|G|I|T|A|L| I N T E R O F F I C E M E M O R A N D U M
+-+-+-+-+-+-+-+

TO: Venus design team

DATE: 22-June-81
FROM: Bob Elkind
DEPT: Venus Design
EXT: 231-6512
LOC/MS: MR1-2/E47
SYS.ID: <BELKIND at 1031>

CC: List

SUBJ: A description of a "data-flow" machine design

In the last month I've been asked to document the differences between a "machine built with flip-flops" and a "machine built with latches."

Attached is the result of my effort to describe these differences, in sufficiently general terms so that the document is useful beyond the scope of the Venus design alone, and with sufficient detail that useful information about the various dependencies and analysis criteria involved with "data-flow" and "clock-driven" design approaches is provided.

In addition, a discussion of the value of a "multi-phase" or "variable-phase" clocking system is included.

Hopefully this paper will be of some value to you. If you have any comments upon the content of this paper, please forward them to me.

Bob Elkind
Venus design group

List:

Gordon Bell	ML12-1/A51
George Hoff	MR1-2/E47
Bob Stewart	TW/C04
Ulf Fagerquist	MR1-2/E78
Pat Sullivan	MR1-2/E85
Don Hooper	MR1-2/E85
Don Lewine	MR1-2/E85
Dave Cane	ML3-2/E41

A Discussion of a Data-Flow and Asynchronous Hardware Architecture as an Alternative to a Clock-Driven and Synchronous Hardware Architecture

First Release Draft by Bob Elkind, 22-June-1981

1.0 SCOPE

This paper covers two sets of opposite approaches to the design of hardware systems, where the two opposite approaches of each set are treated as opposite sides of a single coin.

This is not intended to be a rigorous treatment of the issues involved, but rather should serve as a basis for further discussion. There are certainly many other issues related to the hardware design of computer systems, as well as other planes of issues and discussion. These are beyond the scope of this paper.

The summaries deal with the implications of the comparisons made in a very narrow context, that of high-speed ECL computer systems. Again, the summaries merely serve to relate the comparisons made to a practical context example; and the summaries are not exhaustive by any means.

2.0 DATA-FLOW VERSUS CLOCK-DRIVEN DESIGN APPROACH

The various stages (as in pipeline stages, whether pipelining is used or not) of the various paths within a machine are created by the use of storage elements. These storage elements and their application can be either clock-driven or data-flow (or data-driven) in nature.

2.1 Clock-Driven Approach

The storage elements in a clock-driven machine may be either edge-sensitive flip-flops, or level-sensitive latches which are driven by a clock with narrow pulse width. For the purposes of this discussion, the two behave similarly if not identically; they are both edge-sensitive.

In either case the data input to the storage element must be valid before the earliest possible occurrence of the clock edge (set-up times will be ignored for now). The data outputs cannot be considered valid until the latest possible occurrence of the clock edge (clock->output delays will be ignored for now).

There are three interesting fallouts from this behaviour. First, because the propagation of data from the stage input to the stage output is controlled by the clock edge, a limiting factor to (the lower limit of) the maximum delay through a clock-driven stage is the difference between the earliest and latest possible occurrences of the clock edge. The skew of the clock edge must be incorporated into the propagation delay of the clock-driven stage; i.e, the maximum delay through a clock-driven stage is greater than or equal to the skew of the clock edge.

Secondly, any time before the earliest possible occurrence of the clock edge that the data input is valid, including "fudge factor" and safety margin, is useful for the one stage only. Regardless of how early the data input to the stage became valid, the output of the stage becomes valid no earlier than the latest possible occurrence of the clock edge. Wherever data set-up time (or data valid time) margin has been designed into the machine, that margin will be spent after each clock-driven stage.

In addition, the outputs may be considered unchanged until the earliest possible occurrence of the clock edge. A third fallout from clock-driven stage behaviour is that the amount of cycle time during which the stage output represents either the result of cycle N or cycle N+1 is large, limited predominantly by the skew of the clock edge (the clock->output skews of the stage are ignored until later). The time during which the stage output is indeterminate is, for the purposes of this discussion, determined by the skew of the clock edge.

2.1.1 Characterisation -

1. The stages within a machine must be skewed from each other by at least the delay through each stage (plus the logic delay between stages). The maximum delay through a clock-driven stage is greater than or equal to the skew of the clock edge. If the stages are clock-driven, then

the minimum separation between stages in the machine is at least as great as the skew of the clock edge; where separation refers to the logical distance between stages, which will in turn be defined by the skew between the clocks which drive the various stages.

$$\text{Min. separation between stages} \geq \text{Clock skew}$$

2. The amount of time in each cycle during which the output of a clock-driven stage is valid or determinate is limited primarily by the skew of the clock edge. This will affect the maximum separation between clock-driven stages, for any given cycle time. The maximum separation between consecutive clock-driven stages is equal to or less than the clock cycle time minus the clock skew minus the minimum logic path delay between the stages.

$$\begin{aligned}\text{Max. separation betw. stages} \leq\ & \text{Cycle time} \\ & + \text{Min. logic delay betw. stages} \\ & - \text{Clock skew}\end{aligned}$$

3. Any timing margin (fudge factor) must be added to the minimum separation between any clock-driven stages (which already includes clock edge skew).

4. For any given separation between consecutive stages of a path, the amount of logic delay which can be inserted between the stages will be less than or equal to the separation of the stages minus the maximum propagation delay through one of the stages. Using the identity of maximum stage delay and clock skew:

$$\text{Max. logic delay betw. stages} \leq \text{Separation betw. stages} - \text{clock skew}$$

A measure of stage-to-stage efficiency may be derived as the amount of logic delay between stages (which is defined as the difference between the stage-to-stage separation and that portion of the separation which must be consumed by the propagation delay of a stage), divided by the stage-to-stage separation.

$$\text{Efficiency} = \frac{\text{Separation betw. stages} - \text{Clock skew}}{\text{Separation betw. stages}}$$

$$= \frac{\text{Cycle time} + \text{Min delay betw. stages} - (2 * \text{Clock skew})}{\text{Cycle time} + \text{Min delay betw. stages} - \text{Clock skew}}$$

It may be implied that the effectiveness of a clock-driven machine is heavily dependent upon both the management of clock skew (keeping clock skew small relative to the cycle time), and the careful matching of the delay between stages to the separation of stages.

2.2 Data-Flow Approach

The storage elements in a data-flow (or data-driven) machine will be level-sensitive latches, driven by clocks with pulse widths which are large relative to the logic gate propagation delay of the technology being used. When the clock is active, the latch outputs follow the data inputs; i.e. the latch is "open". When the clock is inactive the outputs are held; and the latch is "closed".

Before a given clock is associated with the latch, it is first determined when the data input will become valid and stable (i.e. when the data "arrives"). Then a clock is associated with the latch such that the clock pulse's midpoint is aligned with the arrival of valid and stable data at the latch input. Data-flow stages are opened well before the input data arrives, and are closed well after the input data arrives. While the clock is active, valid data will enter and flow through the stage; hence the term data-flow.

2.2.1 Margins -

Because the clock is centred around the arrival of input data, and because the clock pulse width is large; if the input data arrival is early or late, then the data output from the stage will be similarly early or late. This allows deviations in path delay to average themselves out over several stages and/or cycles. Timing margin (fudge factor) that is added to the system is carried throughout the machine from stage to stage, rather than being paid for and spent in each stage.

If the data arrives at a stage earlier than the clock's leading edge, then the data must wait for the arrival of the clock (and a latch designed for this timing arrangement must now include leading-edge clock skew in the prop delay through that stage). Conversely, if the data arrives later than the clock's trailing edge, then the data missed the clock entirely. The range of the averaging effect is therefore limited by the width of the clock pulse.

2.2.2 Effects Of Clock Skew And Pulse Width -

Clock skew and clock pulse width both have a large effect on the behaviour and performance of a data-flow machine. However, the role played by clock skew in the data-flow machine is entirely different than in the clock-driven machine. As long as the clock pulse width is sufficiently large such that the stage is open before the arrival of data, then the propagation delay of data flowing through a data-flow stage is independent of clock skew.

At this point it is convenient to analyse the data-flow approach in terms of data valid time. The stages of both clock-driven and data-flow machines perform one primary function, widening the valid time of data input to the stage.

The amount by which a stage must widen the input data valid time, on the average (depending upon accumulated margins), is greater than or equal to the valid time lost between this stage and the previous stage; where the amount of lost data valid time is the data path logic skews (the difference between the maximum and minimum data path logic delays) between the stages. Conversely, the amount of data path logic skew spent between stages, on the average, is limited by the widening of the data valid time achieved by the stage at the data's destination.

The amount by which the data valid time must be widened in a given machine cycle, regardless of the number of stages required to do so, must be equal to or greater than the skews within a given clock cycle. Conversely, for a given number of stages, the minimum supportable clock cycle (i.e. the maximum supportable bandwidth) is limited by the degree to which each stage can provide additional data valid time width.

The degree to which each stage can provide additional data valid time can be analysed in infinite levels of detail. To merely characterise the effects of the various factors involved, however, a very simple model which ignores minimum propagation delays (among other less significant factors) can be used. In addition, the assumption is made that the between-stage delays have been distributed evenly so that the data valid time at the input to each stage is the same.

Given this simplified model, the amount by which a data-flow stage widens the valid time of data flowing through it will be equal to the time during which the stage is closed (i.e. the clock is off). [Note that we have normalised the relationships between data arrival time and the trailing edge of the clock pulse at each stage, which will normalise the "margin" at each stage] By starting with the following definition, the effects and relationships of clock pulse width and clock skew can be related to the maximum supportable bandwidth (i.e. minimum supportable cycle time).

(nominal clock pulse width ON)
\+ (nominal clock pulse width OFF) = clock cycle time

When the skew is taken out of the second term,

```
  (nominal clock pulse width ON)
+ (minimum clock pulse width OFF)
+ (clock skew)                    = clock cycle time
```

The amount of work (in terms of logic skew) which can be done between stages has already been equated to the amount by which a stage can widen the data valid time of its output with respect to its data input, which has in turn been equated to the time during which the stage is closed (i.e. minimum clock pulse width OFF). Once the equated terms; amount of work (skew) done between stages, the increase in valid time at each stage, and the minimum clock pulse width OFF; have been held constant; it is seen that the maximum bandwidth through a stage (or the maximum rate at which the stage may be cycled) is limited by the clock skew and the clock pulse width (ON).

By moving the various terms around again; it can be seen that the amount of work (which will turn out to be primarily logic path skew) which can be done between stages (which; when added to the minimum propagation delay of the logic path between stages; will yield the maximum propagation delay, distance, or separation between stages); is limited by the clock skew, the cycle time, and the clock pulse width (ON).

```
  clock cycle time
- (nominal clock pulse width ON)
- clock skew              = minimum clock pulse width OFF
                          = maximum skew between stages
```

Converting skew to maximum propagation delay, by adding minimum propagation delay to both sides of the equation, yields the following:

```
  clock cycle time
- (nominal clock pulse width ON)
- clock skew
+ minimum logic delay between stages
                          = maximum logic delay between stages
```

2.2.3 Characterisation -

1. Because the data input to a data-flow stage is not waiting for a clock, the clock skew is not added to the logic delay to derive the total path delay between stages. If the clock skew is X and the separation between two stages' clock inputs is N; N amount of logic delay may be placed between the two stages, not N-X.

   The minimum separation between data-flow stages is

limited only by the maximum propagation delay between the stages, and is not limited at all by clock skews.

2. The maximum separation between data-flow stages is limited by clock pulse width, clock skew, and (substituting a more accurate term in the place of clock cycle) the skew of the logic between stages (max prop delay minus min prop delay).

   Max. separation betw. stages
   = < Cycle time
   + Min. logic delay betw. stages
   - Clock skew
   - Clock pulse width ON

3. Timing margin may be accumulated and averaged over many cycles and stages, limited only by the clock pulse width. Timing margin is added by reducing the logic path skew between stages, rather than reducing the logic propagation delay between stages.

   While the cycle time, the clock pulse width (ON), and the clock skew largely determine an upper limit to the amount of skew which can be placed between data-flow stages; the averaging property of data-flow designs allow this limit to be violated, as long as the average amount of skew between stages does not violate this limit.

   Thus, logic elements producing large amounts of skew (e.g. RAMs) may be used without a detrimental effect on the system's cycle time; where otherwise the cycle time might have to be lengthened in order to accommodate the large chunk of skew. Again, the degree to which this averaging may occur is limited by the clock pulse width (ON).

4. For data-flow stages, clock skew is not added to the logic delay between stages when calculating the separation between stages. Keeping in mind that the delays of the gates which make up the stages have been ignored in the characterisation of both data-flow and clock-driven designs, the maximum logic delay between data-flow stages is less than or equal to the separation between stages.

   Using the same definition for data-flow stage efficiency as was used for clock-driven stage efficiency,

$$\text{Efficiency} = \frac{\text{Separation betw. stages}}{\text{Separation betw. stages}}$$

   While it may seem that the solution to the perfect machine has been discovered and proven, this result can

only be achieved when the propagation delays of the data-flow stage latches have been reduced to zero.

Notwithstanding this fleeting encounter with perfection, when the less esoteric details (those which have been ignored until now) are included in the analyses which follow; it will be shown that data-flow stages are not quite perfect. From that point on, the performance of a data-flow machine depends upon the maximum separation which can be achieved between stages; whereupon the effectiveness of the data-flow machine depends heavily upon the management of clock skew and clock pulse width (ON) with respect to the cycle time.

2.3 Stage-Level Comparison: Clock-Driven Vs. Data-Flow

When comparing the efficiency of one approach to the efficiency of the other, it will be necessary to introduce more detail (more factors) into the general relationships already described. It will also be necessary to introduce some practical considerations, such as margining and technology implementations. The following abbreviations will be used in order to condense the equations which will be derived:

C = Clock cycle time
S = Clock skew
P = Clock pulse width (ON)

The next two definitions are simplifications which are entirely appropriate where they will be used.

MAX = The maximum propagation delay of a gate
MIN = The minimum propagation delay of a gate

2.3.1 General Stage-Level Comparison -

The basis for which the data-flow and clock-driven approaches will be compared is their degrees of efficiency. The definition of efficiency used up to this point is the same, although the factors ignored for reasons of simplicity and abstraction will now be introduced. Efficiency is described as the amount of work which may be done divided by the sum of the work done and the overhead required to support that work. On a stage-level basis, this is written as:

$$\text{Efficiency} = \frac{\text{Work done between stages}}{\text{Work done between stages} + \text{Cost of a stage}}$$

Whatever the design approach, cycle time, the skews, the pulse widths, or any other factors; it is clear that the most efficiency will be derived from making the work done between cycles (and therefore the maximum separation between stages) as large as possible relative to the cost of a cycle. It is important to keep in mind which, of the parameters which dictate system performance, will affect both the maximum separation between stages and the cost of the stage; and which parameters will affect the separation between stages without affecting the cost of the stage.

Restating this last equation in terms already used in this discussion, the maximum efficiency is described as the ratio of the maximum time available for logic (work) between stages to the total time (separation) between stages:

$$\text{Efficiency} = \frac{\text{Maximum logic delay between stages}}{\text{Maximum separation between stages}}$$

Using the relations which have been already derived, but adding the factors which have so far been ignored, the maximum separation between clock-driven stages is:

$$\text{Max. sep. betw. clock-driven stages} = C + X*MIN + MIN - S$$

Where X equals the number of gates between the stages, X times MIN equals the minimum logic delay between the stages, and the second MIN equals the minimum clock->output delay of the driving stage. The assumption has been made that all gates and stages exhibit the same MIN and MAX values.

The maximum separation between data-flow stages is:

$$\text{Max. sep. betw. data-flow stages} = C + Y*MIN + MIN - S - P$$

Where Y equals the number of gates between the stages, Y times MIN equals the minimum logic delay between the stages, and the second MIN equals the minimum clock->output delay of the driving stage. The assumption has been made that all gates and stages exhibit the same MIN and MAX values.

The maximum logic delay between clock-driven stages is:

$$\begin{aligned}\text{Max. logic delay betw. clock-driven stages} &= (\text{Max. sep. betw. stages}) - S - MAX \\ &= C + X*MIN + MIN - S - S - MAX \\ &= C + (X+1)*MIN - 2S - MAX\end{aligned}$$

Where MAX equals the maximum clock->output delay of the driving stage.

The maximum logic delay between data-flow stages is:

$$\begin{aligned}\text{Max. logic delay betw. data-flow stages} &= (\text{Max. sep. betw. stages}) - MAX \\ &= C + Y*MIN + MIN - S - P - MAX \\ &= C + (Y+1)*MIN - S - P - MAX\end{aligned}$$

Where MAX equals the maximum data->output delay of the driving stage.

For the purposes of clarity, three variables have been introduced to the equations where two would suffice. Because X and Y are not the same for both clock-driven and data-flow analyses, X and Y need to be defined in common terms.

The equation deriving maximum logic delay between clock-driven stages is defined in terms of C, S, MAX, MIN, and X; where X is the maximum number of logic gates between stages. However, the maximum logic delay between stages can also be defined in terms of X, MAX, and MIN:

$$\begin{aligned}\text{Max. logic delay between clock-driven stages} &= X * MAX \\ &= C + (X+1)*MIN - 2S - MAX\end{aligned}$$

The following is the progression for isolating X:

$$X*MAX = C + (X+1)*MIN - 2S - MAX$$

$$(X+1)*MAX = C + (X+1)*MIN - 2S$$

$$(X+1)*(MAX-MIN) = C - 2S$$

$$X + 1 = \frac{C - 2S}{MAX - MIN}$$

$$X = \frac{C - 2S}{MAX - MIN} - 1$$

By making the same kind of replacement and isolation, the number of gates Y between data-flow stages can be derived:

$$Y = \frac{C - S - P}{MAX - MIN} - 1$$

By making the replacements mentioned and plugging them into the definition of efficiency arrived at earlier, the following equations result:

Efficiency of clock-driven approach

$$= 1 - \frac{(S + MAX)*(MAX - MIN)}{(C - S)*MAX - S*MIN}$$

Efficiency of data-flow approach

$$= 1 - \frac{MAX - MIN}{C - S - P}$$

These last two equations help to analyse the interactions between the various system components as they affect the performance of the target system. For this purpose, the assumptions that all gates and all inputs exhibit the same delays (both min and max delays) is appropriate. In reality, the fact that clock inputs generally behave differently than data inputs to a latch will have to be included.

For now, it is clear that the maximum efficiency which can be derived from a system design is very heavily dependent upon the technology used (MAX and MIN), regardless of the type of approach used.

Where the efficiencies of the two approaches are expressed in terms of the difference which results from subtracting a quantity from one; the quantity which is being subtracted from 1 represents the inefficiency which arises in the necessity for having stages. If stages are not needed, then the efficiency for a given system will rise to 1. Barring the achievement of that state, the reduction in the quantities subtracted from unity will result in added efficiency; and the quantities themselves represent the overhead (the proportional cost) of the stages themselves.

2.3.2 A Comparison Example -

A comparison such as this is limited to expressing relationships rather than presenting major conclusions. However, a design example will serve to underline the process by which these relationships would be used.

Using the MECL 10K ECL logic family as a sample case, the variables used until now are resolved as follows:

C = Clock cycle time = 40 nS
S = Clock skew = 4 nS
P = Clock pulse width (ON) = 10 nS

MAX = The max. prop. delay of a 1ØK gate plus interconnect
= 3.3 nS (avg. of several types) + 1.7 nS (etch)
= 5.Ø nS

MIN = The min. prop.delay of a 1ØK gate
= 1.1 nS (avg. of several types) + Ø.2 nS (etch)
= 1.3 nS

CMAX = The max prop. delay of a 1ØK clock input plus interconnect
= 5.3 nS (avg. of several types) + 1.7 nS (etch)
= 7.Ø nS

CMIN = The min prop. delay of a 1ØK clock input plus interconnect
= 1.1 nS (avg. of several types) + Ø.2 nS (etch)
= 1.3 nS

Note that where the assumption had been made that all gates and latches exhibited the same MIN and MAX values, for all inputs and all outputs; in this MECL 1ØK technology example, in the interests of accuracy, this assumption is avoided. Values for clock input->output delays are now distinguished from other delay values.

Using these rule-of-thumb values yields the following:

Efficiency of clock-driven approach

$$= \frac{C + X*MIN + CMIN - 2S - CMAX}{C + X*MIN + CMIN - S}$$

$$= \frac{C + \frac{C - 2S + CMIN - CMAX}{MAX - MIN}*MIN + CMIN - 2S - CMAX}{C + \frac{C - 2S + CMIN - CMAX}{MAX - MIN}*MIN + CMIN - S}$$

$$= \frac{40.Ø + 9.2 + 1.3 - 8.Ø - 7.Ø}{40.Ø + 9.2 + 1.3 - 4.Ø}$$

$$= Ø.763$$

Efficiency of data-driven approach

$$= \frac{C + Y*MIN + CMIN - S - P - MAX}{C + Y*MIN + CMIN - S - P}$$

$$= \frac{C + \dfrac{C + CMIN - S - P - MAX}{MAX - MIN}*MIN - S - P - MAX}{C + \dfrac{C + CMIN - S - P - MAX}{MAX - MIN}*MIN - S - P}$$

$$= \frac{40.0 + 7.8 - 4.0 - 10.0 - 5.0}{40.0 + 7.8 - 4.0 - 10.0}$$

$$= .852$$

2.3.3 Margin -

Timing margins are necessary for reliable system design. There are two aspects of the timing margin problem which favour the data-flow design approach.

The first aspect deals with the normal variances of delay and skew characteristics between different machines in a production environment; additional margin is built into the timing relationships in the machine to allow for greater tolerance of both parts and manufacturing timing adjustment.

The second aspect deals with the risk associated with a common occurrence in system designs: long serial paths with similar part types (e.g. RAMs, XOR gates, etc.); where slippage in the tolerances of a single part or part type can affect system timing in one critical area, which will in turn affect manufacturing volume for extended periods. This is a particularly serious area of risk where either a gate array or a new part is being used for the first time. In the case of the gate array; if a given gate structure/type should have to be redesigned in either structure or fabrication process; and the re-design affects the timing of the gate type; then a serious timing exposure will exist wherever a serial logic path contains multiple occurrences of the part type. Similarly, where paths are heavily RAM-intensive, a significant exposure to system-level timing is created, due to consistent timing shifts in the part type.

Both aspects of the timing margin issue are potentially alleviated by the nature of the data-flow stage. Given sufficient clock pulse-width (ON), the deviations of the data arrival time at the input to a stage will be reflected in similar deviations in the generation of valid data at the stage output. Hence, deviations from the nominal timing values from stage to stage are averaged over several/many stages; and as long as the absolute, cumulative variance doesn't exceed the tolerance for deviation (which is determined by clock pulse width (ON)) each stage will work.

In addition, the bandwidth of the data-flow system may be greater than the largest amounts of skew between any two stages would otherwise allow, due to the property of averaging timing margin across consecutive stages.

Where margins must be built into (and consumed at) each stage of a clock-driven machine, margins in a data-flow machine may be averaged over many stages and/or cycles within a given loop.

2.3.4 Flip-Flops Vs. Latches -

Throughout this discussion, a clock-driven stage has been assumed to be a latch (rather than a flip-flop) which is driven by a narrow clock pulse (where the clock pulse width is equivalent to the propagation delay of the latch).

When a latch is used in this manner, the overhead of the clock-driven stage is the sum of the clock skew and the clock->data latch skew. If a flip-flop had been used instead of the latch, the overhead of the stage would have been the sum of the clock skew, the clock->output flip-flop skew, and the data set-up time of the flip-flop. The data set-up time is essentially the maximum propagation delay from the data input through the master latch of the master-slave flip flop. When flip-flops are used instead of latches, in a clock-driven machine, this additional overhead must be taken into account when assessing the efficiency or performance of the machine.

A second consideration is the additional circuitry which is required to build a flip-flop from components on a custom or semi-custom logic part, compared to the cost of building a latch. The sacrifice in part count and power (ignoring the speed degradation mentioned above) may be significant.

Since an ECL latch is essentially an AND-OR gate, with its output fed back into an input to achieve the latch function, very often a latch may be expanded with additional AND-OR inputs. In this manner, many logic functions can be achieved in the same logic level as the stage function. In ECL, such added functions are typically ANDing, ORing, XORing, and MUXing. While the 10132, 10134, 10165, and 10173 part types are examples of this; this potential is greatly enhanced in the area of gate-array

implementations, where many required functions may be available for configuration into the latching level at the time of design implementation.

2.3.5 The Tight, Closed Loop -

Very often the general rules-of-thumb which define the various limits for an entire system can be locally optimised, for increased performance, in a small area of the system. One of these cases is where a stage's outputs drive some logic path, which in turn feeds back to the driving stage's inputs.

In the case where the entire stage is narrow enough to be contained on one silicon substrate, driven by a single clock input; then the clock skew is reduced to zero. Even where the stage is not contained on a single substrate, if the area over which the stage is spread is small enough so that a small subset of the clock distribution system is required to feed the clocks to the stage, then the skew for that stage is probably less than the skew over the entire system.

In this context where the clock skew is small, or even zero; the limits to the path's design are affected differently by the various characteristics of the data-flow and clock-driven stages.

In this context, the separation between stages is in fact the cycle time itself. The maximum limits for the separation between stages, for both data-flow and clock-driven designs, are already defined. By introducing the cycle time as the separation between stages, the limits of the various other factors may be assessed:

Separation between clock-driven stages

$$= C = < C + X*MIN + CMIN - S$$

Remembering that the clock skew is zero, and subtracting C from both sides:

$$\emptyset = < X*MIN + CMIN - \emptyset$$

Because the MIN delays must be positive, there is no limiting factor to the single clock-driven stage feeding itself (i.e. when the clock skew is zero).

Separation between data-flow stages

$$= C = < C + Y*MIN + CMIN - S - P$$

Identifying S as zero, and subtracting C from both sides:

$$\emptyset = < Y*MIN + CMIN - \emptyset - P$$

$$P = < Y*MIN + CMIN$$

From this result it is clear that, for a data-flow stage to feed itself, the total minimum propagation delay through the loop must be greater than or equal to the clock pulse width (ON); where there is no such limit or restriction for a clock-driven design.

The requirements for minimum, loop path delay are different between the data-flow design and the clock-driven design. The maximum delays which can be included in the loop path are also different between the two designs:

Logic delay in clock-driven design
= < Separation between stages - stage delay
= < C - stage delay
= < C - S - CMAX
= < C - CMAX

If the clock-driven stage is a flip-flop, then

Logic delay in flip-flop design
= < C - CMAX - MAX

Where MAX is the data setup time (i.e. the propagation delay through the master latch of the master-slave flip-flop).

Logic delay in data-flow design
= < Separation between stages - stage delay
= < C - stage delay
= < C - MAX

Relative to the separation between stages, the data-flow design gains nothing in logic delay in the loop when the clock skew approaches zero. In the clock-driven design the maximum logic delay in the loop, relative to the separation between stages, is increased by the amount the clock skew is reduced, as the clock skew is reduced to zero. The net difference between the amounts of logic in the clock-driven design versus the data-flow design is the difference between the maximum data->output delay of the data-flow stage and the maximum clock->output delay (plus setup time if a flip-flop is used) of the clock-driven stage.

The minimum requirements for logic delay in the data-flow design, again, are dependent upon the clock pulse width (ON); and if the ratio of minimum delay required in the path to the maximum delay allowed in the path is higher than that supported by the logic technology in use, the data-flow approach will not work without changing the clock pulse width (ON). The clock-driven design approach does not have a minimum requirement for loop delay.

Finally, the benefits of cycle to cycle margin and delay averaging which accrue to the data-flow design approach are applicable to this design problem. However, because the degree to which averaging can occur is proportional to the clock pulse width (ON);

and the minimum loop delay requirements are also proportional to the clock pulse width (ON); the determination of clock pulse width (ON) is critical to the behaviour of the design.

3.Ø ASYNCHRONOUS CLOCK VS. SYNCHRONOUS CLOCK DESIGN APPROACH

The comparison between the asynchronous clock and the synchronous clock design approaches is not amenable to the same sort of cut and dried comparison techniques useful for relating clock-driven and data-flow design approaches. Where the clock-driven vs. data-flow comparison involved two discrete antithetical absolutes; the clock design approach compares two extremes between which there is a continuous middle ground: the infinite degrees of multiple-phase clocking systems.

Therefore, the comparison will rest primarily upon the description of the considerations which must be made, their qualitative effects upon a system's performance, and an implementation example.

3.1 Delay And Skew And Cycle Time And Bandwidth

Throughout the discussion of the data-flow and clock-driven approaches to systems design, both the minimum and maximum gate propagation delays (MIN, MAX, CMIN, and CMAX) were vital components of the equations which assessed system performance. For either design approach, the maximum separation between stages is determined by two primary factors: the degree to which a stage can produce additional data valid time (of its output with respect to its input), and the rate at which the given logic technology consumes the data valid time (i.e. produces skew).

Skew between each stage = < data valid time increase per stage

The data valid time increase per stage is primarily a function of the cycle time, the clock skew, and (for data-flow designs) clock pulse width (ON). The amount of logic which can be placed between stages, once the clock system has been specified, is determined by the skew characteristics of the logic technology in use. By separating the maximum and minimum delay components which constitute skew, the following relationship develops:

```
  Max. delay betw. each stage
- Min. delay betw. each stage
                = <   data valid time increase per stage

Max. delay betw. each stage
                = <   data valid time increase per stage
                    - min. delay betw. each stage
```

The maximum delay between stages, which is dependent upon the minimum delay between stages, may also be related to the separation between stages. There are (at least) two significant aspects to this relationship: the delay between stages, which is related to the separation between stages, is determined by both the cycle time and the minimum delays within a system; and therefore the bandwidth (and efficiency) of a design is determined by the skews within a system rather than the maximum delays within a system.

The separation between stages (and the stages' clocks) is related to, but not identical to, the system's clock cycle time. It will be rare that the various factors which affect the maximum separation between stages will produce a result equal to the clock cycle time. Because the efficiency of a design depends upon the maximum separation of its stages, which in turn directly affects the maximum amount of logic which can be placed between stages (for a given cycle time, etc.); an optimised design may depend upon the ability to space its stages, and its clock phases, at some distance other than the clock cycle time.

This independence of cycle time (i.e. bandwidth) to logic delay, and cycle time to separation between stages, is equally important for data-flow and clock-driven designs. Systems which depend upon a single-phase, common clock for all of the various stages within the system usually sacrifice bandwidth and machine efficiency.

3.2 An Implementation Example

This example will use a design style commonly used in minicomputer and microcomputer systems as a basis for comparison between a synchronous, common-clock design and an asynchronous approach.

Using the same parameter values in this example as in the previous comparison between data-flow and clock-driven design implementations; a single-phase common-clock clock-driven path is analysed as follows:

All stages are clocked by the same nominal clock, the cycle time is 40 nS, and the clock skew is 4 nS. Therefore, there must be at least 4 nS of minimum propagation delay between stages to ensure that race conditions from stage to stage, due to the clock skew,

do not occur. Subtracting CMIN from the 4 nS yields

Min. Logic delay betw. stages > = 4 nS - CMIN = 4 - 1.3 = 2.7 nS

This is the only area in which minimum propagation delays must be considered when a common-clock approach is used. In order to provide the 2.7 nS of minimum logic propagation delay, using the values for MAX and MIN which have already been specified, a minimum of approximately 10 nS maximum logic delay must be placed between stages.

By subtracting from the cycle time the various skew and delay components which limit the amount of logic which can be placed between the stages, the amount of logic delay which can be used in a single cycle can be derived.

Logic delay = Separation betw. stages - S - CMAX
= C - S - CMAX
= 40.0 - 4.0 - 7.0
= 29.0 nS

This maximum allowable amount of logic between stages easily fulfills the requirements for the minimum logic delay between stages.

By deriving the maximum allowable logic delay between clock-driven stages, using the equations derived in the previous discussion of clock-driven and data-flow stages; the sacrifice in efficiency due to the use of a common-clock design may be quantified:

Max. separation betw. stages

$$= C + \frac{C - 2S + CMIN - CMAX}{MAX - MIN} * MIN + CMIN - S$$

= 40.0 + 9.2 + 1.3 - 4.0
= 46.5 nS

Maximum logic delay betw. stages
= Maximum separation betw. stages - S - CMAX
= 46.5 - 4.0 - 7.0
= 35.5 nS

While maintaining the same bandwidth (i.e. cycle time), by placing the stages and their clocks 46.5 nS from each other (instead of 40 nS apart) an additional 6.5 nS (max. delay) of logic could be placed between each stage. This represents a 22% increase in the amount of logic (i.e. work) which can be placed between stages.

While holding the 29.0 nS of logic delay between stages as a constant, by abandoning the common-clock approach the cycle time can be decreased to 35.2 nS, for a 14% increase in system throughput. In this case the separation between stages (and their

clocks) would be 40 nS.

3.3 Fitting Stages Around Logic

The fact that the common-clock synchronous design approach depends upon the immovability of stages, and their clocks, presents problems when the logical ordering of a data path places a large, indivisable logic block (i.e. a RAM or cable, etc.) where a stage is/should be. In such cases, there are two alternatives: the data path may be re-ordered, if that is feasible; or the stage may be dropped, while two additional stages (and special, non-common clocks) are added (bracketing the logic block).

The advisability of violating the consistency in the system's clock scheme may well depend upon the necessity for attempting such a violation, or upon the frequency with which this type of problem arises, or both. In the normal course of the design of general-purpose systems, several of the more critical paths in the design may well be RAM-intensive; in which case the cost in cycle time degradation which would result from fitting the logic paths into a rigid pipeline structure would far outweigh the cost of the extra clocks which would be required to optimise the paths.

In general, the logic must fit between the stages or the stages must fit around the logic. If the fit is sometimes difficult, either the cycle time will grow to fit the worst-case logic chunk problem; or the clock system will grow to accommodate the logic path.

3.4 Degrees Of Asynchronous Behaviour

The benefits of a less rigid clock system are significant; and the degree to which the clock system is allowed to be flexible may have a significant effect on the throughput of a system. However, the solution space is a range of degrees rather than an either/or set of alternatives.

The consistency of a common-clock system allows certain assumptions to be made when performing timing analysis; which may in turn make the design task simpler and/or easier. Having the freedom in the clock system to design the stages around the logic (where necessary) may also make the design task easier; while adding to the complexity of the clock distribution, clock setting, and clock de-skewing problems.

The comparison of these two design approaches to the clock system cannot generate a simple result. The means by which analysis may be done can be documented, but ultimately a judgmental tradeoff must be made. On one extreme, the system's throughput is clearly degraded by the adherence to a single, common clock. At the other extreme, the cost of the clock system rises geometrically with little additional benefit in system performance.

Rarely would a system require or benefit from a clock system which offers clock variations to a resolution which is less than the maximum gate propagation delay of the technology being used. Depending upon the tightness of the design, a clock-phase resolution granularity greater than a single gate delay may be acceptable, with minimal impact upon the system's performance.

At points between the two extremes, the clocking system allows for multiple phases of "standard" clocks. The tradeoff between system performance and clock system cost and complexity should yield the proper solution.

JUN 4 1981

********** FOR INTERNAL USE ONLY **********

DIGITAL INTEROFFICE MEMORANDUM

To: Charlie Mitchell
Ike Nassi
cc: Bob Glorioso
Jud Leonard
Sas Durvasula
George Hoff
Ulf Fagerquist
Gordon Bell

Date: 3-June-1981
From: Earl Van Horn
Dept: Corporate Research
Loc: ML3-4/I50
DTN: 223-5272
MAIL: RDVAX::VANHORN

Subj: Monthly report for May 1981

I have been asked by Gordon Bell to spend some time with the VENUS development group to see if software engineering methodology, particularly top-down techniques, can be applied to help them cope with the complexity of their system so they can verify its correctness before putting power to hardware. Consequently my work on system query facilities for ATMOS is on hold for an undetermined period: at least one month, perhaps more.

VENUS WORK

My first order of business for VENUS was to understand the nature of the problem and determine what, if anything, I can do to help. To this end I attended some review meetings, read the System Plan, looked over some of the design documentation, and talked with Roy Rezac, George Hoff, Sas Durvasula, and Jud Leonard.

Of the several problems facing the project, there are two on which I could conceivably be of some help. First, the existing design, which is fairly complete at the gate level, is not adequately documented. Second, there is a need for tools to help validate the timing aspects.

To address both problems, I proposed that the structure and behavior of the system at a high level of abstraction be described in a form that can be checked by computer for timing sanity. The description would be useful in design reviews and walkthroughs, and the sanity check might help avoid some timing problems. Jud Leonard pointed out, however, that the check would not be of as much value as I had hoped, because signal timing is not as dependent on high-level function as I had thought.

Nevertheless, a tool that encourages a top-down description and performs some useful checking would be well-received, I think, particularly if the description is useful in a walkthrough. So I am continuing to explore some ideas I have in this area.

Jud and others are preparing high-level design documentation, which I will review for understandability. Since Jud is fully committed to making the design be understandable, I expect that my review will be a useful contribution.

OTHER ACTIVITIES

I attended the Corporate Research woods meeting, where I gave a presentation on my recent activities, including lessons learned on the STEP project. I also gave two more presentations on Ada tasking to the IPBS group, participated in the Architecture Task Force for the Methods and Tools WORKSHOP project (formerly SCOPE), and assisted Chuck Bradley in presenting his System Builder proposal to the RAD committee. I discussed trends in software tools with Dennis Yeow of Software Services for a report he is writing, and referred him to some other people. I also attended a STEP phase review, seminars on CAP and WEB, and a presentation by Ralph London.

BS 1-34

MAY 29 1981

------*---*---*---*---*
* d i g i t a l *
------*---*---*---*---*

INTEROFFICE MEMORANDUM

TO: Gordon Bell
Rick Casabona
Brian Croxon
Bill Demmer
Sas Durvasula
Pauline Nist

DATE: 28 May 1981
FROM: Bob Stewart
DEPT: Interconnect Program Office
EXT: 247-2125
LOC/MAILSTOP: TW/C04

SUBJECT: 11/780 TO MCA TRANSLATION

Gordon asked me to spend two days estimating and exploring what would happen if we took the existing 11/780 design and translated it directly into MCA technology. This memo is a report on that project.

1. Cycle Time: I started with a document prepared by Steve Jenkins when Venus was proposed, in which he listed out the components in the Instruction Decode Path, Data Path, and TB Cache Path on the 11/780, translated these into MCA cells (with output cells in appropriate places), and added up the MCA times, plus an allowance for wiring. I plugged in my recollections of current MCA timing, and came out with estimates just under 80 nanoseconds.

2. CPU Performance: An 80 nanosecond cycle time would give an improvement of 2.5x, but this will be degraded because the cache miss time will not improve. (I assumed we did not want to try and change the SBI and all peripherals.) I estimate that the ultimate improvement in CPU performance would be about 2x.

3. Partitioning into MCA's: As an experiment to determine how effective a post facto partitioning of MSI logic into MCA's could be, I spent a few hours trying to partition the TBM board of the 11/780. I ended up with half the total chip count inside MCA's. Six MCA's of five types were needed. All were limited by input and/or output cells, not by internal logic cells, with none of the MCA's being more than half full. All the critical paths did go into MCA's.

It is probable that longer effort would improve these results, and it is also probable that not all logic will go into MCA and that most MCA's would be only partially full.

4. Cost: ECL is more expensive than TTC, MCA's are not cheap, and neither are the boards used. Very roughly, I would guess that an ECL/MCA 780 would cost between five and ten thousand dollars more to build than the vanilla flavor. This is less than Venus, but not as much less as the performance.

5. Redesign: Certain sections cannot be translated directly. The two SBI interface boards would require changes to interface ECL to TTL and to account for the 80/200 cycle time difference between processor and bus. The console interface board would need changes for similar reasons. Since there are no useful ECL PROM's, it would be necessary to use all writeable control store.

6. Software: Diagnostics would undoubtedly be effected.

7. Mechanical: Changes to cooling requirements may effect choice of blowers. It might be necessary to increase board spacing, effecting what will fit in the basic box. New power supplies would be needed. Change to L series connectors to get some spare pins for surprises.

JUN 2 1981

digital INTEROFFICE MEMORANDUM

TO: Gordon Bell ML12-1/A51

DATE: 28-May-81
FROM: Rick Casabona
DEPT: HARDWARE INT. ENG.
EXT: 247-2373
LOC/MAIL STOP: TW/C04
FILE: FAST780.MEM

cc: Bill Demmer TW/D19
Pauline Nist TW/C04
Brion Croxon TW/C04
Bob Stewart TW/C04
Steve Jenkins TW/C04
John Holz TW/D20

SUBJECT: "FAST" IMPLEMENTATION OF THE VAX 11/780

References:

Memo: Gordon Bell, "What if the 780 were realized using MCA's or NEW TTL/FAST logic?", 25-May-81.

Memo: Bob Stewart, "TEXAS INSTRUMENTS 74AS", 13-MAR-80.

WORK done by Steve Jenkins VRT MCA equivilant of the 780.

I was asked to spend a couple of days evaluating the feasibility of a "direct substitution" design of the 11/780 using the new FAST/TTL. The results of that investigation are included herein.

APPROACH:

The approach chosen was to use the work that Steve Jenkins had done in evaluating and MCA implementation of the 11/780, substituting FAST components for Schottky, New Rams and Proms for old, and the II Cache Chip for the CACHE ram/equals checker. No attempt was made at redesigning the paths involved.

CPU CYCLE TIME (Max of 1.6 X 11/780):

The paths chosen were:

1. Data Path Loop,

2. Control Store Loop,

3. TB, Cache Loop.

The worst of these three paths came out to a cycle time of 125 ns. as opposed to the 200ns now used by the 11/780, for a CPU performance increase of 1.6 X 11/780. The summary of 11/780 vs. MCA equiv. vs FAST for the above paths is included at the end of this memo. The comparison also contains data provided by Steve Jenkins and Bob Stewart.

CPU PERFORMANCE DEGRADED BY SBI ACCESS:

Since the CPU time is not an integral multiple of the SBI cycle, an SBI access requires either:

1. The SBI to be slowed down by 25% such that it is 2X the CPU cycle time. This is intollerable with respect to I/O thoughput.

2. The CPU misses cycles, thereby degrading CPU performance, during:

    1. Cache Misses,

    2. Memory Writes,

    3. I/O accesses,

    4. Interrupt Fielding.

This approach requires a redesign of the CPU to SBI interface to provide the synchonization between them.

FAST (FAIRCHILD ADVANCED SCHOTTKY TTL) SUMMARY

FUNCTIONALITY

The design target for the FAST technology was 'Faster than Schottky Speed at Low Power Schottky Power'. The propagation delay of FAST components is in the order of 15% to 50% less than that of the equivilent Schottky component. The relative prop delay improves as the complexity (number of Logic Layers) of the Chip increases.

AVAILABILITY

Some FAST parts are available today. It is expected that production QTYs will be available by DEC. 81. Expected second sources are Signetics and Motorola.

SPECS.

Specs are currently in the preliminary or incomplete stage. The parts that I encountered are not speced over temp and voltage. Some are speced only at typcial at "normal" voltage and temp.

COST

The Cost of FAST is currently projected at 2-3 times that of Schottky. The goal is to bring the cost 3 years from now to the cost of today's Schottky.

TECHNOLOGY

FAST has not been evaluated with respect to design constraints or layout rules. It is expected that Schottky PC board technology can be used with some extra attention paid to transmission line effects and decoupling requirements.

SUMMARY

The performance of the 11/780 likely cannot be increased by a factor of 2 by a direct substitution design using FAST and new RAMS. Some advantage can be gained with respect to power requirements (possibly reducing the CPU power system by one supply ($800)). This savings would not offset the increased cost of the FAST logic. A redesign of the CPU to SBI interface would be necessary. The power/ cooling/ package system of the 780 could likely remain unchanged. Relayout (edit) of most CPU modules would probably be necessary since some of the components do not have a direct FAST replacement.

11/780 Built with ECL

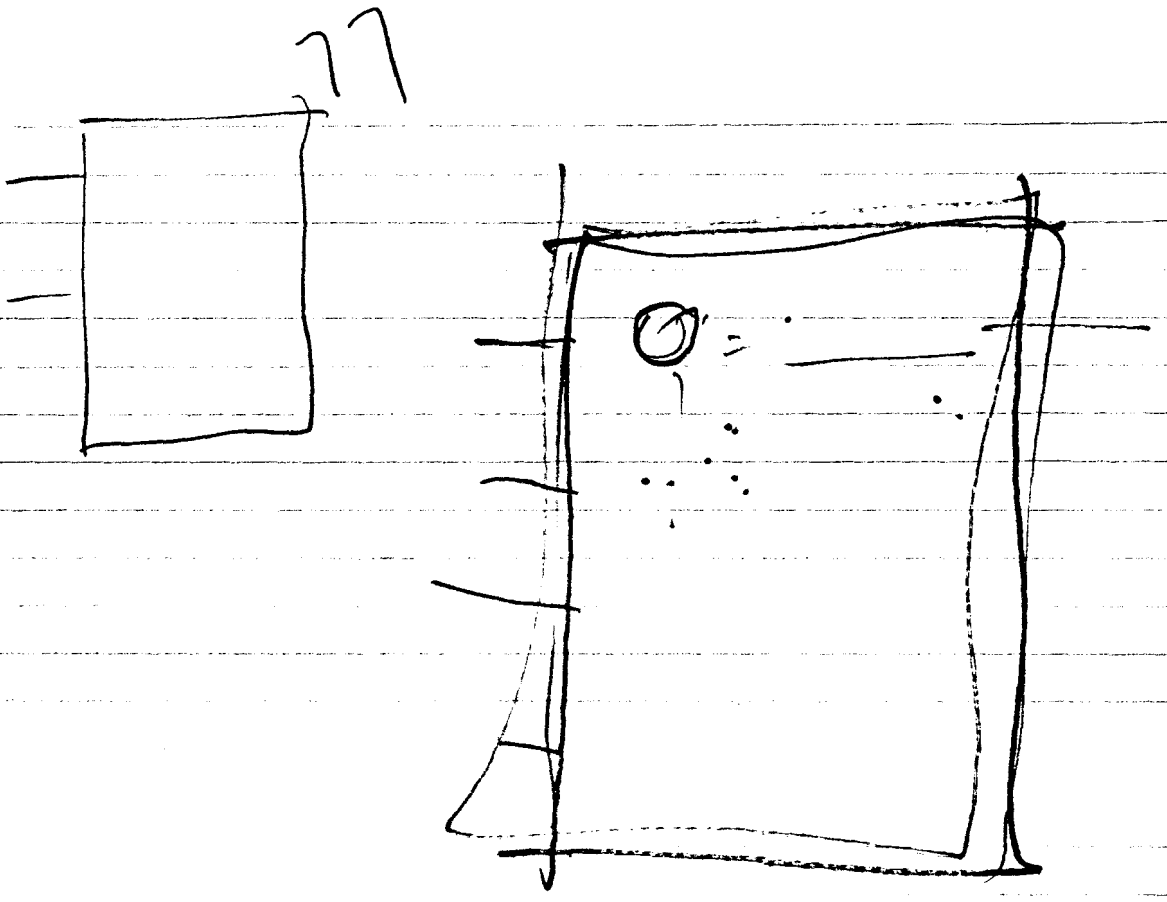
*** SUMMARY WORST CASE PATH ANALYSIS

FUCNTION	11/780	MCA EQIV	FAST EST WC.	FAST TYP
DATA PATH LOOP	173	53	116	96.3
*CONTROL STORE LOOP	198.5	76	120	103.8
TB, CACHE LOOP	244	96	157	140

*

MCA 80 ns. vs. 200 ns. ---> 2.5 X 11/780 cpu

FAST 125 ns. vs. 200 ns. ---> 1.6 X 11/780 cpu


77
R →

****
* d i g i t a l *
****

TO: *GORDON BELL

DATE: MON 18 MAY 1981 17:34 EST
FROM: BILL STRECKER
DEPT: VAX ARCHITECTURE
EXT: 247-2130
LOC/MAIL STOP: TW/A08

SUBJECT: VENUS

You asked me what we would do without VENUS, I'm not certain I know the context of the question (e.g. cancel it and use the money elsewhere; what happens if it fails; etc.); but I'll venture an answer anyway. I consider VENUS absolutely essential for two reasons:

1. As a continuation to the very profitable; image-setting; upper mid-r by the 11/70 and 11/780.

2. As a critical component to the VAX strate offer a wide range of comp

The corporation should identify/provide/draft the resources to make VENUS succeed.

The State of the Design
It seems like we have:

- text spec of entire machine, since the box names are incorrect, its unclear whether this has any validity
- text spec of boxes, together with some block diagrams, description doesn't reflect the design
- flow charts specifying behavior of a few parts
- behavioral description, in TUMS of the boxes, non-operational as a complete system. Even if it operated as a complete system, it is unclear what it means due since the timing interaction of interconnect signals like stalls and micro-interrupts are so complex.
- 500 SUDS logic drawings specifying the machine
- 500 SUDS logic drawings for MCAs
- MCA physical layout, simulation description, test patterns, test data as to whether the MCAs meet their specs

What would be the IDEAL state of the Design?
A document describing all the documents and what the design ground rules are going to be. This would
It seems clear that we must have an accurate, structured design at all stages of
Ideally, this design description is in machine readable form so that it can be c
possible by machine. Since the main design tool is SUDS, this implies SUDS compatilble software that must be written asap.

What would be the IDEAL design system?
0. Information only appears once and notification is made to users of information as it changes. Ideally a data base could do this, however, this has shown to be virtually unworkable using DBMS and the IDEA database. Alternatively, there must be batch update to flag changes.
1. Interactive, with access to any part, ~~with~~ protection of data ~~as to~~ the owner.
2. Report writing ability to get parts out as needed when there are large pieces that need to be looked at or put on the wall. No document should be more than 15 minutes away, with instant access to everything else, including access via SUDS terminals.
3. Overnite batch update of key part such as indexes, rule checking, etc. that is less frequently needed.
4. Calculator functions like the clock checker that operate on the data to give the designers quick access for making critical decisions ← like clock phases.
5. Data base-like access of critical information that is structured in order of detail.
6. An index and on line access to all signals, giving:
   - name, together with polarity, location, time generated, and the source drawing it is created on (derived from SUDS)
   - a list of all the drawings it goes to and the time the pin expects to see the signal (derived from SUDS)
   - a text description of the meaning of the signal (designer input, together with date created,
   - a list of all the specifications that reference the signal
   - a list of all the other documents that reference the signal

++++++++++++++
!D I G I T A L!
++++++++++++++

INTEROFFICE MEMORANDUM

JUN 8 1981

TO: VENUS Project Leaders
Gordon Bell
Ulf Fagerquist
Alan Kotok
Sas Durvasula

DATE: 5 June 1981
FROM: George Hoff
DEPT: LSG VAX Engineering
EXT: 231-6524
LOC: MR1-2/E47

SUBJ: ISSUES RAISED IN COST CENTER MEETING 4 JUNE 1981

1. 10KH: Appears to provide the solution to clock system implementation. Gordon raised the following question:

   Why not apply 10KH across the board instead of 10K?

   What will the speed advantage and voltage compensation buy us?

   Paul Guglielmi will look into this and generate a response.

2. What resource limitations are limiting the schedule?

   Response:

   - Graphics terminal limitations in the SUDS area.
   - Plotter performance.
   - Can't keep high performance graphics (VBIO) running.
   - Computes for peak batch loads.
   - Computes for simulation load which is coming.
   - Disk space/data base management.

George will coordinate a review of solutions with Dave Copeland - review next week.

3. In the discussion of resource limitations, some suggestions were made:

   - Get a faster 11 for the Versatec.
   - Get SUDS on Gigi.
   - Find out what Hudson is doing for SUDS.
   - Get another KL for simulation (Tim Beers) has a plan for this).
   - Get more advanced SUDS terminals.

DESIGN CLOSURE PLAN

Goal: Reduce risk in shipping the committed product on time.

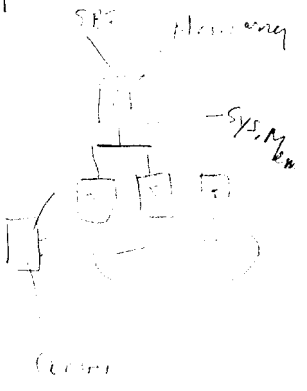
Problem: The VENUS hardware design is too complex to be developed under the current plan/technical organization structure at an acceptable risk level.

Strategy:

- Establish and maintain technical control of the hardware design via a hierarchical approach to design and implementation.
- Complete the design of the final product, complete debug via paper check/simulation, to a point where build schedule can be committed at a quantified risk level (bug find model).

Implementation Priorities:

- Develop and install hierarchical technical control process at top level/box level. Jud is the owner/tie breaker at the top level ← a replacment must be found as the top priority. Complete box interface specifications/box specifications.
- Develop tools to make hierarchical design feasible. Top level timing model, is the critical item short term.
- Resolve partitioning of control versus box boundaries. Resolve box module boundaries (one owner per module).
- Complete current design including timing model based on standard technology specifications. Complete design of final clock system.
- Technical review the current design to establish design completeness, performance, risk, which will result in approval of plan to complete final product design. Need to demonstrate technical control and ability to manage change from this point on.
- Get the tools we need to complete final design.
- Complete the final design.
- Verification/review/simulation.
- Release to build.



5/27/81

cc: Gordon
Brian
Hal
F

JUN 12 1981

SR11/14-1

+---+---+---+---+---+---+---+
| d | i | g | i | t | a | l |
+---+---+---+---+---+---+---+

INTEROFFICE MEMO

To: John O'Keefe TW/A08
Bill Demmer TW/D19

Date: 5-June-1981
From: Steven Rothman
Dept: 32 Bit Planning
Ext: 223-2391
Loc/Mail Stop: PK3-1/M56

Subject: Some Thoughts on 780 Enhancements and Venus

I personally believe it's unlikely there is anything that could be done to the /780 that would allow it to become a Venus "backup" or "replacement". This is my reasoning:

Should assume there is a VENUS.

1. We couldn't use custom LSI. Any project that did that would likely take at least as long as Venus.

2. Without custom LSI, the product would not have sufficient performance for its cost to make it a full lifetime member of the product set; that is, it could only be a short term backup, but not a replacement.

3. I don't believe there is any way for us to make a significant performance improvement in the basic /780 CPU, without a major redesign. The attempt to drop faster logic into the current design doesn't appear practical (see report of Rick Casabona).

4. I assume that all the improvements that are practical for the memory system are already planned as part of the new memory controller for 64K chips.

5. I don't believe that any significant performance improvement could be made in the FPA for F and D formats, since the FPA already takes nearly "no time" for ADD and SUB (which are most of the FP instructions executed). That is, the CPU and FPA have nearly complete overlap for those instructions. However, if G and H format become a significant factor, we could always create a new FPA that includes the new formats. I believe it only makes sense to work on a new FPA to include the new formats.

6. I suspect that some applications will see a significant improvement in performance when we make the COMBO board available; a version with 8 lines with modem control will probably be necessary. The 32 bit system managers have agreed to formally ask Distributed Systems for such a product.

RECEIVED
JUN 8 1981
BILL DEMMER

7. Speeding up the SBI might make a significant difference to some applications, if it were possible and practical. I suspect that the amount of improvement we could get compared to the costs of changing all the SBI devices (CPU, Mem Cntl, UBA, MBA, DR, MA) make this also a bad idea.

8. I'm not aware of any major problems with our software that cause us performance degradation (other than the terminal handling already covered in item 6), so I doubt that software changes could help make the /780 a Venus backup. (See item 10 below)

9. Some product lines would like to see some of the cost taken out of the /780, mostly in the packaging. This might extend the life of the product, but doesn't really help to make it a Venus backup.

10. Last, but not least, is the possibility of using multiple /780s, connected by either CIs or the MA780, as a backup to Venus. There are probably some applications where this would be acceptable. I suspect that we would need major changes to our software plans (at least the calendar) to accomplish this in the FY 83-84 timeframe. I expect to get some more information about this choice from Bill Heffner (Program Team action item).

****
* d i g i t a l *
****

TO: BILL DEMMER
SAM FULLER

DATE: SUN 14 JUN 1981 19:31 EST
FROM: GORDON BELL
DEPT: ENG STAFF
EXT: 223-2236
LOC/MAIL STOP: ML12-1/A51

SUBJECT: BUILDING VENUS AND OTHER VAXES

I don't want to lose any time on getting this project started. We should have news shortly on the I-box. My greatest fear is that the I-box can't be proved to work, or will work so slowly that the performance will be something like a 780. The first estimates for the path are around 100 ns in the I box. I don't understand pipelining well enough to know what that means though.

Our problem here:
780 mid life kickers
Venus alternative
Nautilus
Staffing adequately to give Venus a fair shot
Cane, and a high performance machine
Scorpio, so it doesn't become a Venus

BJ is thinking of moving the office systems work to the UK. This would leave a team of very bright programmers without a project. They could be the basis of a system that would design the next machine, given a couple of good logicians and electrical engineers. The more I see us grunt away on logic, the clearer it is that it's not the way to do the job.

++++++++++++++++
!D I G I T A L!
++++++++++++++++

INTEROFFICE MEMORANDUM

To: Supnik, Teicher, Will Sherwood, Fuller, D. McGinnis, Bob Stewart, Croxon

TO: VENUS Project Leaders

CC: Gordon Bell
Sas Durvasula
Ulf Fagerquist
Alan Kotok
Jud Leonard

DATE: 1 June 1981
FROM: George Hoff
DEPT: LSG VAX Engineering
EXT: 231+6524
LOC: MR1+2/E47

Is this possible? Can we use it on Nautilus?

SUBJ: DEVELOPMENT OF PROGRAMMABLE CONTROL LOGIC

Do we have the tools?

To date, on VENUS, we have used a strategy of microcode and 10K implementation to keep high risk control logic out of MCA implementations. During my discussions with Gordon last week, he suggested an additional strategy: develop a PLA type structure on an MCA which could support implementation of complex control logic structures in VENUS. I believe this approach would support the goal of a more structured design and should be considered in areas where we are considering moving control logic into MCAs to solve partitioning problems.

Can I get some creative ideas on how this could be done and how it might be applied in each of the major boxes?

++++++++++
G I T A L!
++++++++++

INTEROFFICE MEMORANDUM

TO: Gordon Bell

CC: Project Leaders
Sas Durvasula
Alan Kotok
Jud Leonard
Ulf Fagerquist

DATE: 1 June 1981
FROM: George Hoff
DEPT: LSG VAX Engineering
EXT: 231-6524
LOC: MR1-2/E47

SUBJ: YOUR NOTE ON MC68000/Z8000 COMPARISON (attached)

Recently, I did some metrics evaluation on several processors. In this evaluation, I used the number of gates, excluding ROM/RAM, as the complexity factor. None of the CPUs considered, used a PLA approach (which I would count as a ROM).

I found that development cost (which one could equate to schedule), normalized for inflation, tracked gate count. The variation around the mean tracked the degree of specialization of the gates, i.e., 2020 using bit slice, was on the low side, MSI machines in the middle and gate array machines on the high side.

This lends credence to your suggestion of structuring control logic, i.e. build PLA type structures with gate arrays where possibe. I intend to pursue breakdown of VENUS to get a handle on the distribution of gates in regular structures (data path/address path) versus random control. Data path/address path are rarely a problem since they are easily verified via desk check and microcode simulation. Control logic is the problem area and the first step in developing a simulation strategy will be to size the problem.

Attachment

Bob Stewart

Cane, Sherwood, Steve Jenkins

This also applies to our regular design too! (not)

note the Sam Low

very long gestation times of 2½ Mohawks, Comet, Venus ... with No simulation

The impact of microcoded control on the design and debugging of chips can be illustrated by comparing the history of two 16-bit microprocessors that have complex control requirements: the MC68000 (Motorola 79, Stritter 79) and the Z8000 (Zilog 79, Peuto 79). The MC68000 was designed using microcoded control, while the Z8000 uses combinatoral logic for its control. Although the MC68000 was fabricated using a somewhat denser and faster technology than the Z8000, this difference is not important. The overall architectural complexity of the two microprocessor chips is comparable. Figures 3-1 and 3-2 show the actual chip layouts, illustrating the obvious regular microcode memories in the MC68000.

Doesn't this say keep it regular → ; simulate it; don't worry about # of transistors ?

	Motorola MC68000	Zilog Z8000
Design parameters:		
Elapsed time to first silicon	30 months	30 months
Design time	100 man-months	60 man-months
Layout time	70 man-months	70 man-months
Lambda	1.75 μm	2 μm
Chip size	246 mils x 281 mils	238 mils x 256 mils
Number of transistors	68000	17500
Number of transistors/ Number of designed transistors	74:1[6]	5:1 (estimate)
Debugging parameters:		
Elapsed debugging time[7]	6 months	18 months
Number of mask sets	3	3
Percent of chip design simulated at any level	100%[8]	20%
Percent of chip area devoted to test structures	4%	0%
Total time to fcs	36	48

Wow!

Table 3-1: A comparison of the MC68000, and Z8000 design efforts. *Data supplied by Tom Gunter of Motorola, Inc., Bernard Peuto of Zilog, Inc., and Skip Stritter of Nestar Systems, Inc*

As seen in Table 3-1 the most striking testimony to the advantages of microcoded design is the fact that the Z8000 required three times as many months to produce a working production version (i.e. a chip with no known bugs) as the MC68000. The various forms of simulations of the MC68000 and the test logic on the chip itself made it possible to examine essentially all internal signals in order to locate the source of errors. Moreover, even bugs which did not originate in the microcode of the MC68000 could often be fixed using microcode instead of eliminating the true source of the error by redesign. On the other hand, locating the sources of bugs in the Z8000 was often tedious and required difficult changes to the random-logic control.

Quite apart from testing and debugging efficiencies, use of microcode offers substantial savings in layout effort as well. Even though the MC68000 has four times as many transistors as the Z8000, both chips required approximately the same amount of effort to design and lay out.

[6]This is actually the weighted average of three different ratios. 50000 transistors sites are in PLA/ROM structures with a ratio of 50:1. 5000 transistors sites are in the registers with a ratio of 500:1, and 13000 transistors sites are in random logic with a ratio of 5/1.

[7]Time from first silicon to a working production version (i.e. no known bugs) of the chip

Bell Labs Teichan Carnegie-Mellon Reddy SAIT Kusik C. Rupp Kotok Bauman

Sav.

1. [illegible] FX.

→ Not there yet ... part of constraints

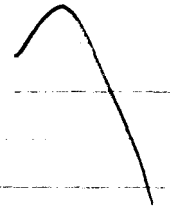
⇒ 2. Upfront, to use most value, [illegible] in the company

⇒ [illegible]

(Like up front, detailed problem solving [illegible])

⇒ [illegible] to be [illegible] Value [illegible]

not on backend



* d i g i t a l *
****

TO: ULF FAGERQUIST

cc: *GORDON BELL
BILL KEATING

DATE: WED 20 MAY 1981 13:22 EST
FROM: SAM FULLER
DEPT: SA&T
EXT: 223-4562
LOC/MAIL STOP: ML2-2/H33

SUBJECT: MAKING 2080 AND VAX MORE COMPATIBLE

Richard Stallman of MIT will be meeting with Ron Criss in the near future to discuss moving some of the MIT "TOPS-20" environment into and onto VMS. This will no doubt be a risky, difficult to manage "adv. dev." effort but you might talk with Criss, if you see a positive potential here, and encourage him to proceed with an open mind.

Bill Keating has been coordinating the meeting and he has talked with the VMS group about the proposal. The are not opposed to the proposal per se but are not interested in being the group that would manage Stallman's work.

Sam

* d i g i t a l *
****

TO: *GORDON BELL

cc: see "CC" DISTRIBUTION

DATE: TUE 19 MAY 1981 13:22 EST
FROM: ALAN KOTOK
DEPT: OOT
EXT: 223-7381
LOC/MAIL STOP: ML2-2/H33

SUBJECT: RE: VENUS AND ITS CONTINGENCIES

I, as usual, have serious concerns about a strategy which proposes to move TOPS-20 customers to VMS-based systems, at ANY time in the future. I couldn't care less how many bits the machine has in its words. What I do care about is the "feel" of the operating system. To me, VMS doesn't feel as good as TOPS-20. I am unaware of anyone who has used both who perfers VMS. Symptomatic of the probem is a note from Len Kawell circulating on the Engineering Net (which I would have attached, except for the disjointedness of EMS and Eng Net). This note says that "command completion" and "in line help", which are considered by the TOPS-20 community to be two of the major features of the system, are of no value, and would not be considered for inclusion in VMS. With that attitude, it's going to be a long time before TOPS-20 customers will consider VMS a viable alternative.

Another problem which just might be worth spending some time to evaluate again is the amenability of VAX architecture, versus PDP-10, to very high speed implementations. We seem to keep finding that VAX architecture has a number of "gotchas" which make pipe lining very painful. Maybe Cane and Orbits can shed some light here. Don Hooper feels that the variable length of VAX instructions, the byte alignment problems, and the difficulty of parsing VAX instructions due to the dependence on previously parsed fields all cause difficulties in building high speed machines. I guess the question I'd ask is whether the apparent factor of 2 reduction in bytes of instruction processed per unit algorithm is wiped out a factor of 2 in added complexity to process those bytes of instruction.

What it all comes down to, in my view, is let's make sure we have a product in the VAX/VMS space which is apparent to THE CUSTOMERS as superior to PDP-10/TOPS-20 before we cut them off at the knees.

"CC" DISTRIBUTION:

BILL DEMMER	ULF FAGERQUIST	SAM FULLER
GEORGE HOFF	BILL MCBRIDE	LARRY PORTNER

****
* d i g i t a l *
****

TO: ULF FAGERQUIST

cc: *GORDON BELL

DATE: THU 25 JUN 1981 15:12 EST
FROM: ROY REZAC
DEPT: MR SITE ENG'G MANAGER
EXT: 231-4140
LOC/MAIL STOP: MR1-2/E18

SUBJECT: CRITICAL PERSONNEL FOR VENUS SIMULATION

Can you help me get the people needed to get the Simulation Process completed? I have completed all necessary planning for the Simulation Process sub-project of the overall Venus Simulation effort (its goal is to get the Simulation Models and Process in place so we can actually begin doing simulations). To do this project on its October 1st delivery schedule, I need additional resources beyond those I can allocate from the LSCAD Group. The resources I have are as follows:

1. Project Leader - Gary Brown

2. Programmer - Amancio Hasty

3. Simulation Engineer - Kin Quek (for one month and then back to F Box Simulation).

4. An agreement for Steve Ching for 50% of his time for the next 6 months.

5. Mike Newman for two months.

The additional resources that I need by July 15th are as follows:

1. Agreement that Steve Ching (Supervisor Omur Tasar) can work with us 100% of the time for four months and 50% of the time for two months.

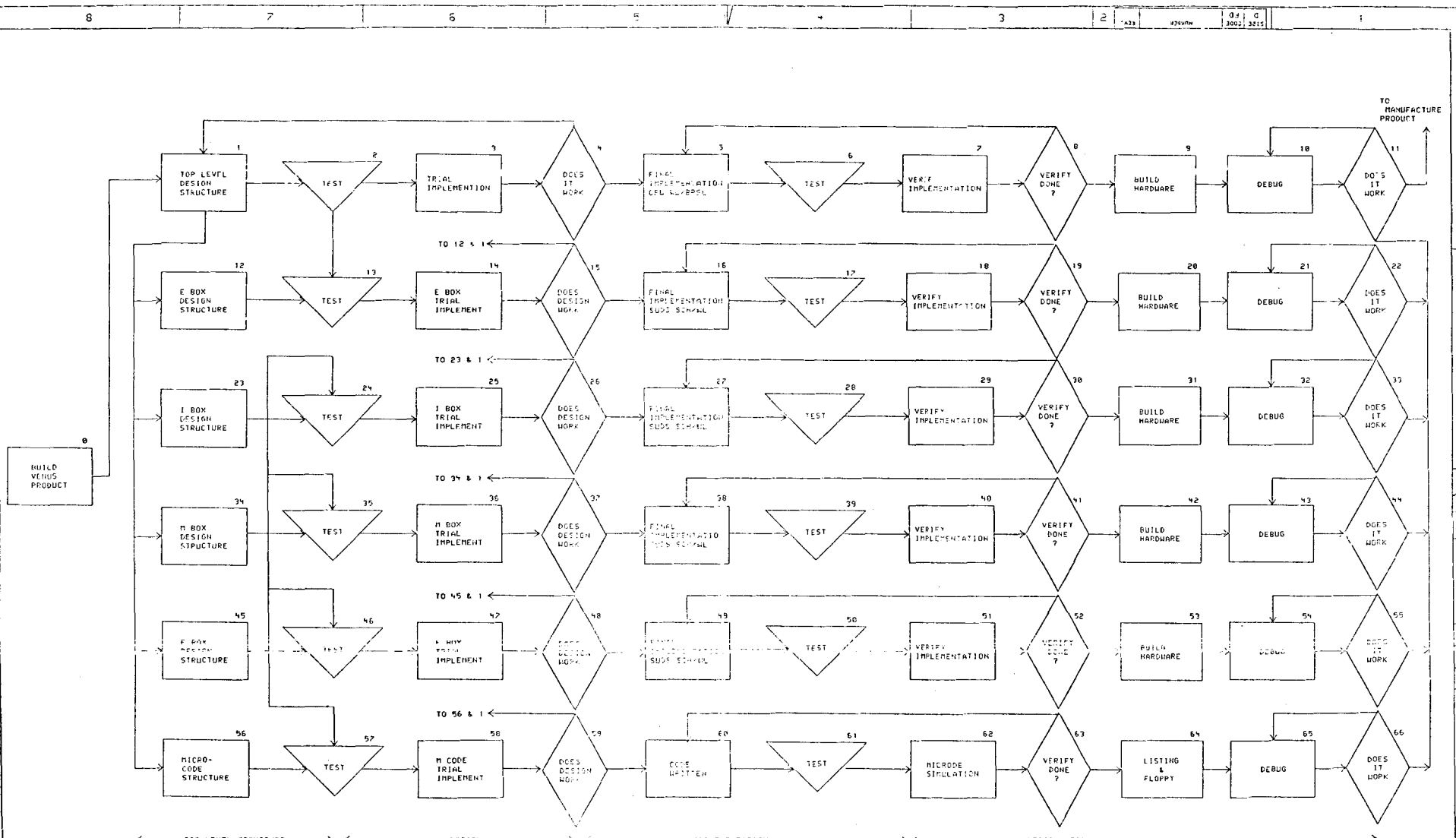
2. Agreement that Dave Gross (Supervisor Will Sherwood/Joe Zeh) can spend 2 weeks with us now and then 1 day per week for four months to do consulting work.

3. That Tim Aldridge (Supervisor Len Dalton) can work with us full-time for the next 6 months.

4. That Richard Langer (Supervisor Bill Blake) can work with us full-time for the next 6 months.

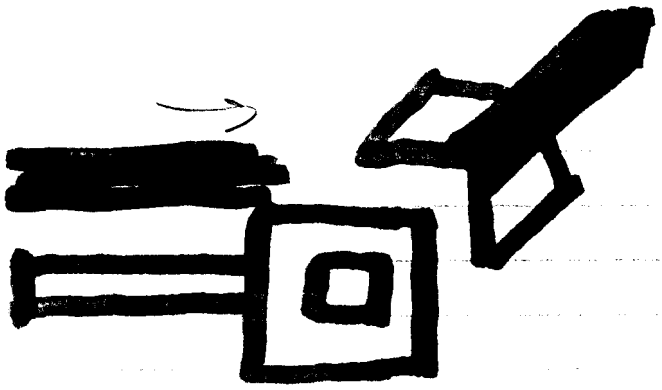
5. That Jeff Leavitt (Supervisor not known at this time) can work with us for the next 2 months to develop models.

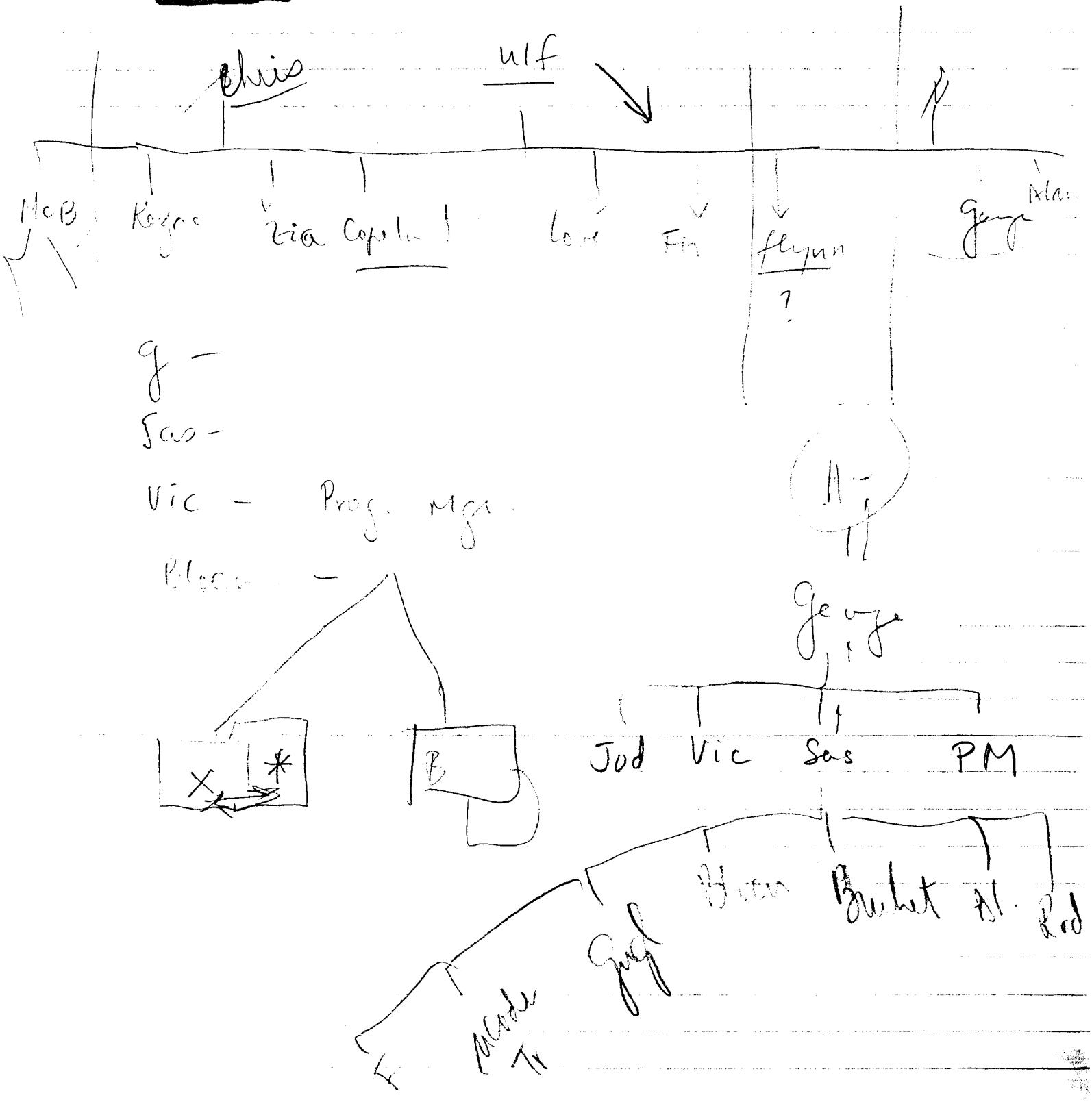
I have called their managers, and am waiting for return calls. I need your help to expedite getting these people. Can you help?



REVISIONS		
ITR	CHANGE NO.	REV

DRAWN	DATE	DATE	TITLE: VENUS TOP DOWN DESIGN METHODOLOGY
CHK'D	DATE	SHEET OF	
PROPRIETARY DIVISION	NEXT HIGHER ASSEMBLY	SIZE C	CODE
FIRST USED ON		NUMBER	REV.




Chris
Ulf
McB
Kevin
Tim
Flynn
?
George
Jason -
Vic - Prog. Mgr.
Blair -
George
Jud
Vic
Sas
PM
Brent
Rod

JUN 25 1981

digital

INTEROFFICE MEMORANDUM

TO: Attendees

DATE: 25 June 1981
FROM: Sas Durvasula
DEPT: Large VAX Engineering
EXT: 4426
LOC/MAIL STOP: MR1-2/E47

SUBJECT: MEETING NOTICE

Beginning Monday, June 29th, I will be holding Design Review Status meetings. The Agenda for next week will be announced at Monday's meeting. Following, is the meeting schedule for the next two weeks:

Monday, June 29th	1:30 - 4:00	Pheonix Conference Room - MR2
Tuesday, June 30th	10:00 - 12:00	TG Conference Room MR1-1
Wednesday, July 1st	2:00 - 5:00	DEC-10 Conference Room
Thursday, July 2nd	3:00 - 5:00	DEC-20 Conference Room
Monday, July 6th	1:00 - 3:00	DEC-10 Conference Room
Tuesday, July 7th	10:00 - 12:00	Lunar Conference Room MR1-2, Pole L7

NOTE: Please bring material for your particular box per the attached list to each meeting.

SD:pl

Attendees: See attached list

attachments

CHECKLIST OF THINGS TO BRING FOR THE REVIEW

- LIST OF BOX INTERFACE SIGNALS
- BLOCK DIAGRAMS THAT REPRESENT THE LOGIC
- MICROWORD DESCRIPTION OF FUNCTION
- A COPY OF THE BOX SPEC

SAS DURVASULA
18 JUNE 1981

VENUS DESIGN REVIEW

GOAL: FUNCTIONAL DESIGN RE V IEW OF THE SYSTEM TO IDENTIFY PROBLEMS AND CLOSE DESIGN.

PROCESS:

* PUBLISH LIST OF ATTENDEES TO COVER THE VARIOUS FUNCTIONS OF EACH BOX AND DIAGNOSTIC PROGRAMMERS.

* CONCENTRATE ON ONE BOX OR FUNCTION PER SESSION.

* A. KOTOK IS THE TECHNICAL MODERATOR AND TIE BREAKER.

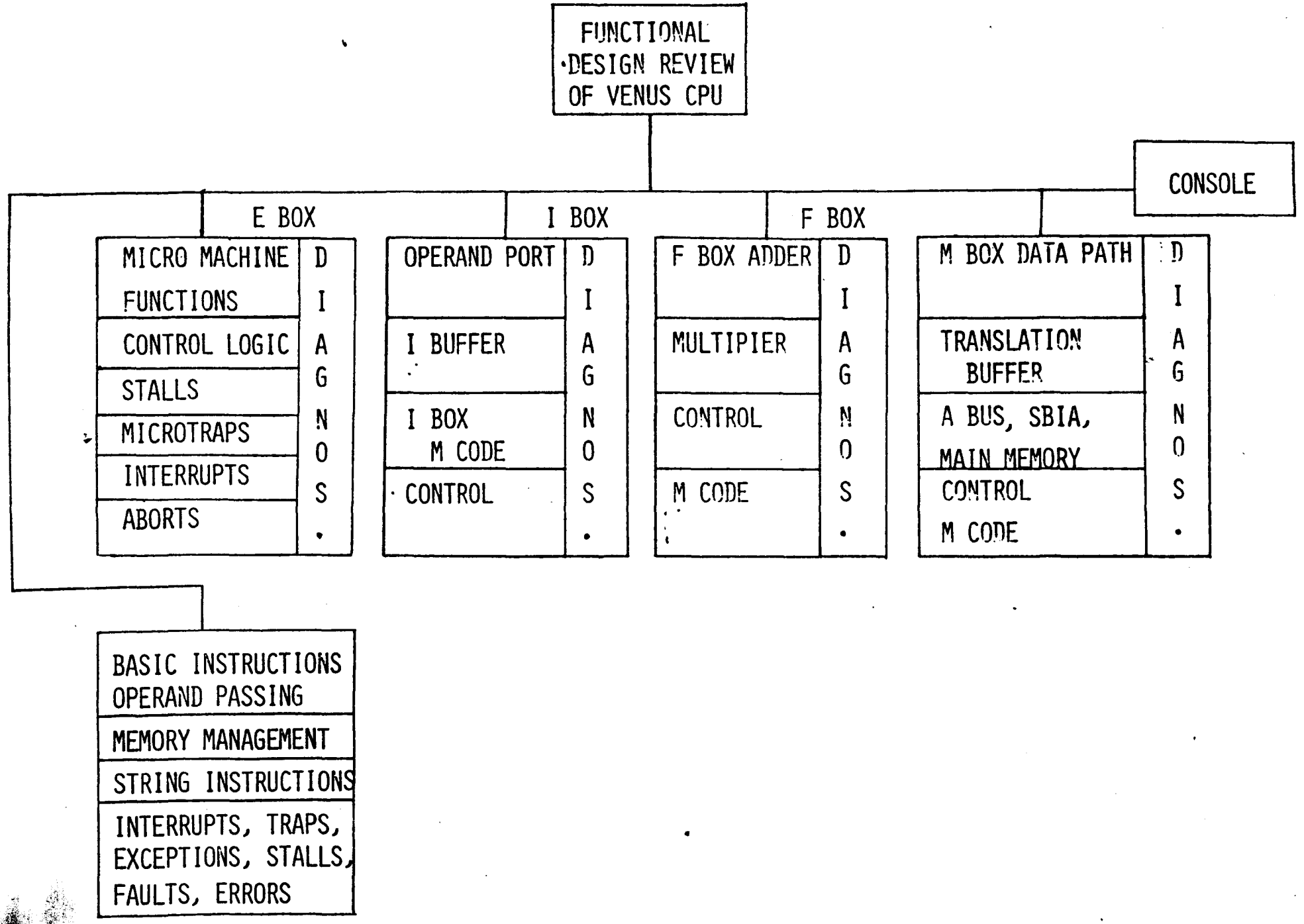
* HAVE A CHECK LIST FOR EACH BOX TO MEET THE REVIEW COMPLETION CRITERIA. (SEE ATTACHED CHECKLIST).

* FILL OUT PROCESS (PROBLEM SHEETS FOR CLOSURE).

* REVISIT SPECIFIC ISSUES FOR FINAL SOLUTIONS.

TANGIBLE OUTCOME:

* CLEAN DESIGN SPECS TO BE PLACED UNDER REV CONTROL.


FUNCTIONAL DESIGN REVIEW OF VENUS CPU
CONSOLE
E BOX
MICRO MACHINE FUNCTIONS
CONTROL LOGIC
STALLS
MICROTRAPS
INTERRUPTS
ABORTS
DIAGNOS.
I BOX
OPERAND PORT
I BUFFER
I BOX M CODE
CONTROL
DIAGNOS.
F BOX
F BOX ADDER
MULTIPIER
CONTROL
M CODE
DIAGNOS.
M BOX DATA PATH
TRANSLATION BUFFER
A BUS, SBIA, MAIN MEMORY
CONTROL
M CODE
DIAGNOS.
BASIC INSTRUCTIONS OPERAND PASSING
MEMORY MANAGEMENT
STRING INSTRUCTIONS
INTERRUPTS, TRAPS, EXCEPTIONS, STALLS, FAULTS, ERRORS

ATTENDEES:	ED ANTON	MR1-2/E47
	JOHN BLOEM	MR1-2/E47
	RAY BOUCHER	MR1-2/E47
	BILL BRUCKERT	MR1-2/E47
	AL DELLICICCHI	MR1-2/E47
	JOHN DEROSA	MR1-2/E47
	SAS DURVASULA	MR1-2/E47
	BOB ELKIND	MR1-2/E47
	ULF FAGERQUIST - F.Y.I.	MR1-2/E78
	BARRY FLAHIVE	MR1-2/E47
	TRYG FOSSUM	MR1-2/E47
	BILL GRUNDMANN	MR1-2/E47
	PAUL GUGLIELMI	MR1-2/E47
	AL HELENIUS	MR1-2/E47
	BILL HILLIARD	MR1-2/E47
	GEORGE HOFF - F.Y.I.	MR1-2/E47
	MIKE KAHAIYAN	MR1-2/E47
	TOM KNIGHT	MR1-2/E47
	ALAN KOTOK	MR1-2/E47
	JIM LACY	MR1-2/E47
	CLEM LIU	MR1-2/E47
	JOHN MANTON	MR1-2/E47
	JOHN MCALLEN	MR1-2/E47
	PETER RADO	MR1-2/E47
	EILEEN SAMBERG	MR1-2/E47
	MOHAMMED TYABUDDIN	MR1-2/E47
	TONY VEZZA	MR1-2/E47
	BILL ENGLISH	MR1-2/E47
	DIAGNOSTICS;	
	KATHY BAKER	MR1-2/E68
	JEFF BARRY	MR1-2/E68
	DICK BEAVEN	MR1-2/E68
	BILL DALE	MR1-2/E68
	RICH GLACKENMEYER	MR1-2/E68
	TOM MOORE	MR1-2/E68
	BOB PETTY	MR1-2/E68
	DAVE TIBBETTS	MR1-2/E68
	GORDON BELL - F.Y.I.	

JUL 7 1981 Vems

+---------------+
| d i g i t a l |
+---------------+

i n t e r o f f i c e
m e m o r a n d u m

TO: R. Casabona, J. Grady, D. Hooper, S. Jenkins, A. Kotok, R. Rezac
CC: See Distribution

DATE: 6 July 1981
FROM: Sam Fuller/Doug Clark
DEPT: Sys, Arch. & Tech.
DTN: 223-4562
LOC: ML2-2/H33

SUBJECT: CONCLUSIONS FROM THE 11/780 ECL STUDY

Attached are the viewgraphs that were presented at Bill Demmer's staff meeting on July 1, 1981. It is the result of a short term effort by Doug Clark, and myself, with help from others in Tewksbury and Marlboro.

Rick Casabona had collected similar information for the meeting and had come to similar conclusions.

The results of the discussion at the meeting was that if we decided to do a technology speedup of the 780 CPU, that standard 100K ECL was the technology to use.

Since this option continues to be considered as one alternative for a 780 midlife enhancement, comments and criticisms on the proposal are welcome.

/id
Att.
SF9:48

DISTRIBUTION

G. Bell
B. Demmer
J. Holz
P. Nist
J. O'Keefe
L. Portner
B. Strecker

CAN WE GET A FASTER VAX SOONER THAN VENUS?

- COMPETITORS HAVE 32 bit COMPUTERS FASTER THAN THE VAX-11/780 TODAY.

- UNCERTAINTY WITH RESPECT TO VENUS SCHEDULE.

CLARK &
FULLER
1 JULY 1981

ASSUMPTIONS

1. PRIMARY GOAL IS MINIMIZING TIME TO MARKET.

2. APPROACH WILL BE TO SPEED-UP CPU.

2A. OTHERS ARE LOOKING INTO OTHER ALTERNATIVES SUCH AS MULTIPROCESSOR CONFIGURATIONS, NEW COMM. OPTIONS, ...

3. WE NEED ~ 2X SPEEDUP, AT LEAST.

CONCLUSIONS

THE VAX-11/780 CPU TRANSLATED INTO ECL 100K ($ REST OF SYSTEM UNCHANGED).

PERFORMANCE : 2 TIMES 11/780

TRANSFER COST : $26K MORE THAN 11/780

SCHEDULE : 14 MONTHS TO RUNNING PROTOTYPE

(12 MORE MONTHS TO FRS?)

Technology – translation Alternatives

	7.80	750
F. A. S. T.	Too Slow	Much Too Slow
ECL 10K	Not as fast as 100K	Major re-design
ECL 100K		Major re-design
MCA's	Major redesign	Interesting! But takes too long to do

Design Ground Rules

1. Simple technology translation of 780
   NO "IMPROVEMENTS" TO DESIGN

2. No microcode changes

3. Board partitioning and backpanel connections stay the same

4. SBI stays the same
   (therefore small design changes to interface)

5. Cache and TB quadrupled
   (small amount of redesign)

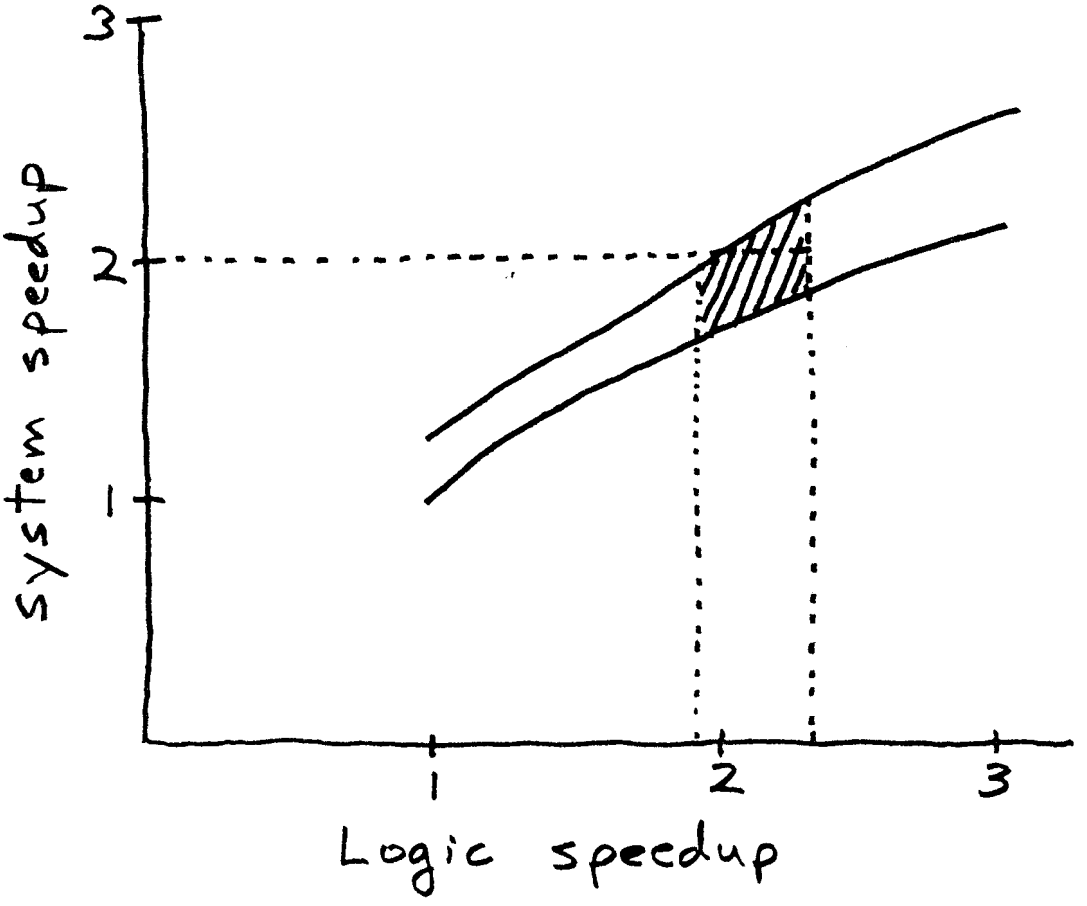
Approach: Use JUPITER technology

- Multiwire boards
- MPS power
- Corporate cabinets
- Controlled-impedance backplane
- ECL design rules
- Jupiter CAD
    - SUDS
    - Assortment of timing tools, checkers, etc.

Performance

- Based on close analysis of control-store loop (worst path in 780), we believe logic speedup of 1.9 - 2.3 X is attainable

- SBI speed remains constant. Quadrupling cache and TB lets system speedup follow logic speedup:



TRANSFER COST

ITEM	11/780	ECL 780	Cost Δ
ADDITIONAL SW CAB.			$950
POWER			
3 ECL "PAIRS"			2262
1 AC CONTROLLER			500
1 AIR MOVER			200
1 (MORE) CARD CAGE			400
BACKPANEL			
FPA			500
CPU (TTL → ECL)			500
POWER & PACKAGING			$5312
BOARDS (QTY: 25)	$150/bd	$300/bd	$3750
CHIPS (QTY: 2975)	$1/CHIP	$4/CHIP	$9000
RAMS (QTY: 275)	$8/CHIP	$38/CHIP	$8250
TOTAL			$26,000

COST OBSERVATIONS

1. MANY COST ESTIMATES ARE VERY SOFT

2. COST OF CPU BOX MORE THAN DOUBLED.

3. COST OF MOST VAX-11/780 SYSTEMS ABOUT $60K to $80K. HENCE:

   $\frac{1}{3}$ COST INCREASE $\Rightarrow$ 2X CPU SPEED

Engineering Schedule

	Man-months	Increment
1. Start-up		3 mos.
2. Design translation Assume 8 engineers	50	6 mos.
3. Multiwire layout Assume overlap with (2) Assume 4 people	25	1 mo.
4. Fabrication Assume overlap w/(2)+(3) Assume premium paid for fast turnaround of last bds.		1 mo.
5. Debug + re-work		3 mos.
TIME TO DEBUGGED PROTOTYPE		14 months

Engineering Manpower

8 Designers, including:

- 1 780 expert
- 1 ECL expert (at least)

4 Multiwire layout people

Risks and Uncertainties

- Chips per module is large
  - for Multiwire
  - for PC
- 100K chip costs could come down with volume purchases
- Opportunity cost of taking engineers from other projects
- Many cost estimates are soft

For: Jeff
From: Vic Ku

To: Gordon Bell JUL 20 1981
fyi - about the initial work on debug [zero defect] model - to base simulation and hardware debug on

VENUS DEBUG PROCESS

A. GOALS

B. DEBUG STRATEGY (REPLAN)

- CLASS, TYPE, SEVERITY OF ERRORS
- RANGE OF ERROR IN EACH CLASS
- OVERALL APPROACH IN APPLYING FILTERS
- TYPE OF FILTER & ESTIMATED EFFICIENCIES
- APPROACH IN GETTING BETTER FILTERS/ EFFICIENCIES
- ERROR COLLECTION & PREDICTION
- ERROR REPORTING SYSTEM & PREDICTION

C. KEY POINTS OF DEBUG STRATEGY

D. ACTIVITIES TO BE CONTINUED

E. PROPOSED SCHEDULE

VIC KU
7/14/81

GOALS: To design and implement a process to allow as many design/implementation errors as possible to be detected and eliminated as early as possible in the shortest possible time.

CLASS OF ERROR:

ERROR CLASS	RELATIVE SEVERITY	# ERRORS IN VENUS (Vic's Estimation)
1) System	H	2 - 5
2) Logic Design	H	500 - 1000
3) CAD/Simulation Tools	M	150 - 350
4) Microcode	L	200 - 600
5) VMS	M	30 - 100
6) Software Layered Products	M	6 - 20
7) Diagnostics	M	200 - 600
8) Circuit/Power System	H	5 - 20
9) Mechanical	H	5 - 20
10) Environmental	H	2 - 10
11) Construction/Fabrication	M	100 - 300
	Total Venus Error (Estimated)	1,200 - 3,025

APPENDIX TO ERROR DETECTION MATRIX:

1. System -

- Performance
- System Loading/Interaction
- Hydra Configuration
- Revision Control
- 2 SBI's Systems
- Specifications
- Bus Bandwidths
- Memory & Memory Expansion

2. Logic -

- Clock System
- Data Path
- Control Logic
- Timing
- Performance
- Interrupt Logic
- Memory Management Logic
- Error Detection Logic
- Error Correction/Recovery Logic
- Error Logging Logic
- Specification Discrepancies/Inconsistencies
- Margins (MCA, PC, BP)

3. CAD/Simulation Tools -

- Loading Rules
- Design Rules
- Libraries
- Delay Times
- Delay Analysis Tools
- Terminator Checking
- Simulation Tools
- Data Base Management

4. Microcode -

- Performance
- Specifications
- Functionality
- Loading of
- Error Detection
- Error Correction/Recovery
- Error Logging

5. & 6. VMS & Layered Products -

- Specifications
- 2 SBI's
- Console Handler
- Error Detection
- Error Correction/Recovery
- Error Logging
- Loading of
- Performance
- New I/O Handlers
- System Loading/Interaction
- Hydra Configuration

7. Diagnostic -

- Fault Coverage
- Performance
- 2 SBI's
- Functionality
- Error Recovery
- Error Logging
- Error Detection & isolation
- Human Engineering
  - Ease of use
  - Sequencing
  - Loading of
  - Output format capability
- Remote Diagnostic Capability
- I/O Diagnostic

8. Circuit/Power System

- Emulator (circiut)
- Circuit Rules
  - Loading
  - XOR
  - Crosstalk
  - Reflection
- Component Qualification/Spec/Screens
- Sourcing/Compatibility
- Design Specifications
- Functionality
- Failure Modes
- Sockets (electrical)
- MTBF of Components
- EMM
- Margins

9. Mechanical -

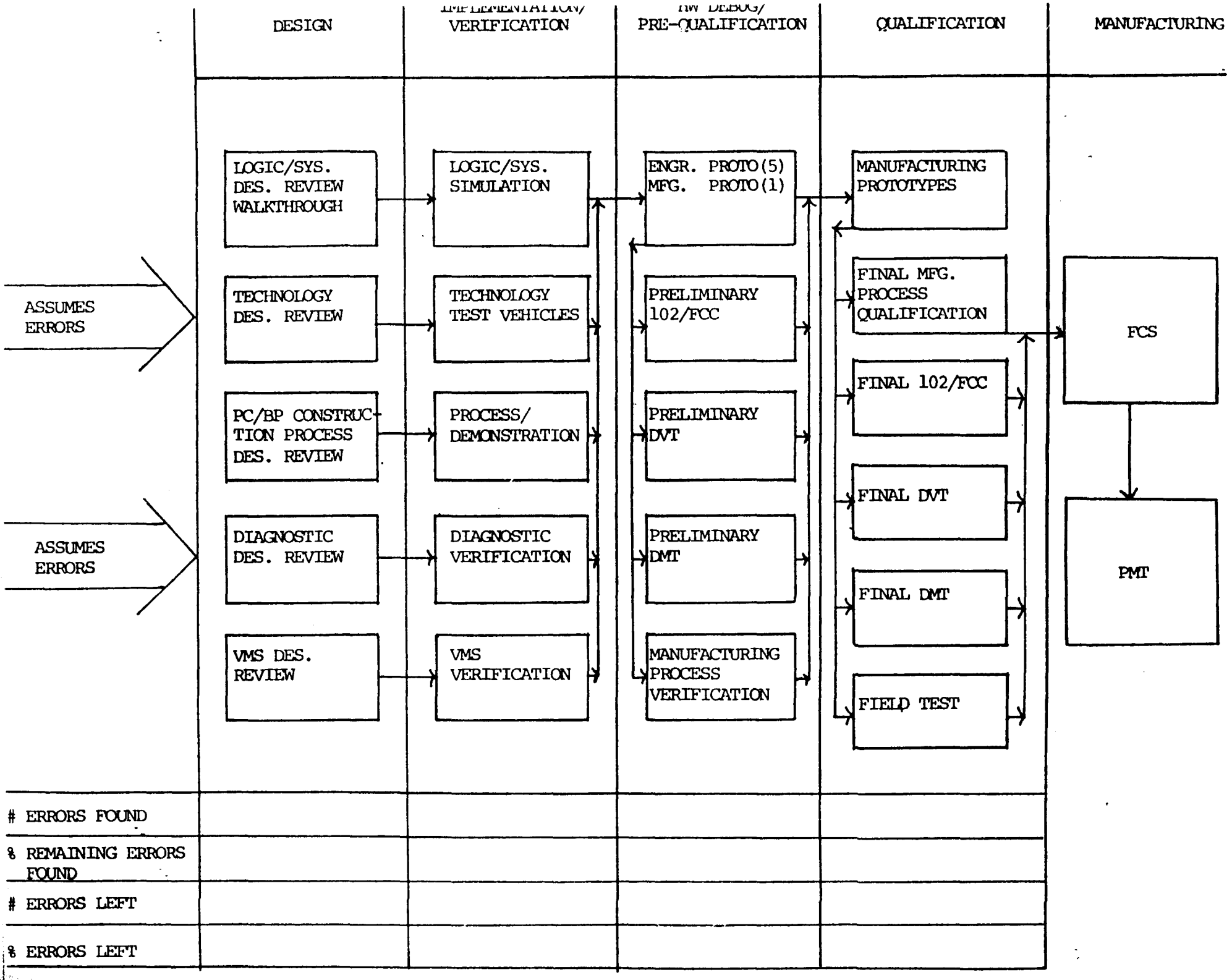
- Emulator (mechanical)
- Cooling System
- Mech Fab Design
- Connector Design
- Cable Design
- Sourcing
- Socket (mechanical)
- Margins

10. Environment -

- FCC
- 102
- 50 Hz
- Acoustics
- VDE

11. Construction/Fabrication -

- Open & Short Etches
- Bad Components
- Missing Terminators
- Components Mounted Wrong
- ECO Procedures
- Incoming Inspection Procedure & Tester for component, MCA, PC, BP, Mechanical Fab, & Cables.


DESIGN
IMPLEMENTATION/ VERIFICATION
HW DEBUG/ PRE-QUALIFICATION
QUALIFICATION
MANUFACTURING
ASSUMES ERRORS
ASSUMES ERRORS
LOGIC/SYS. DES. REVIEW WALKTHROUGH
TECHNOLOGY DES. REVIEW
PC/BP CONSTRUCTION PROCESS DES. REVIEW
DIAGNOSTIC DES. REVIEW
VMS DES. REVIEW
LOGIC/SYS. SIMULATION
TECHNOLOGY TEST VEHICLES
PROCESS/ DEMONSTRATION
DIAGNOSTIC VERIFICATION
VMS VERIFICATION
ENGR. PROTO(5) MFG. PROTO(1)
PRELIMINARY 102/FCC
PRELIMINARY DVT
PRELIMINARY DMT
MANUFACTURING PROCESS VERIFICATION
MANUFACTURING PROTOTYPES
FINAL MFG. PROCESS QUALIFICATION
FINAL 102/FCC
FINAL DVT
FINAL DMT
FIELD TEST
FCS
PMT
ERRORS FOUND
% REMAINING ERRORS FOUND
# ERRORS LEFT
% ERRORS LEFT

TYPE OF ERROR	ESTIMATED PERCENTAGE OF ALL ERRORS	FILTER TYPE & EFFECTIVENESS IN DETECTING ERROR																						
		Walkthrough	Test Vehicles	Simulation	Vote	Perf. Analysis	Timing Analsyis						ENGR. PROTOTYPE	PRELIM. 102/FCC	PRELIM. DVT	PRELIM. DMT	MFG. PROTOTYPES	FINAL 102/FCC	FINAL DVT	FINAL DMT	FIELD TEST	FCS	PMT
'stem		M	-	M	-	M	-						M	M	H	H	M	M	H	L	H	H	L
ɔgic		M	-	H	-	-	H						H	L	H	L	H	L	H	L	H	H	L
\D/Simulation Tools		M	H	H	-	-	-						M	L	L	L	M	L	L	L	L	L	L
1S/Layered Products		M	-	-	-	-	-						H	L	H	L	H	L	H	L	H	H	L
.agnostics		M	-	-	H	-	-						H	L	H	L	H	L	H	L	H	H	L
.rcuit/Power System		M	H	-	-	-	-						H	H	L	H	H	H	M	H	M	M	H
echanical		M	H	-	-	-	-						M	H	L	H	M	H	L	H	M	M	H
ıvironmental		M	M	-	-	-	-						M	H	L	M	L	H	L	L	M	M	M
ɔnstruction/Fabrication		M	H	-	-	-	-						H	M	L	H	H	M	L	H	L	M	H

VENUS:
ERROR DETECTION
EVALUATION

NAME:

FUNCTION:

DATE:

FILTER EFFECTIVENESS

O - NOT EFFECTIVE AT ALL

L - 10 - 40% EFFECTIVE

M - 40 - 80% EFFECTIVE

H - 80 - 100% EFFECTIVE

PLEASE EVALUATE FILTER EFFECTIVENESS:

O = 0 - 10% effective
L = 10 - 40% effective
M = 40 - 80% effective
H = 80 -100% effective

APPROACH IN GETTING BETTER FILTER & HIGHER EFFICIENCY

- Get Functional Groups to fill our individual "Filter Efficiency Form".

- Work with Functional Managers in deficient areas and on new filter designs/modifications to achieve desirable filter efficiencies.

    - To have at least one, highly effective, filter before Prototype Power-On.

	11/750	11/780	KL10	VENUS
# CPU Modules	5	25	51	18
Gate Equivalent	40K	68K	35K	41K
BB1 Logic Error	80*	77	120*	
BB2 Logic Error	43	18	18	
Prototype Logic Error	28	45	158	
PCO + ECO's	20	82	43	
Total Logic Error	171	222	339	(500-1000)*
Total Error	450	662	793	(1200-3000)*
Hitachi Model (1 Error in 20 to 50 gate)	800-2000	1360-3400	700-1750	1820-4500
Total Debug Time (BB & Proto)	50W	45W	78W	40-60W

*Estimated by Vic Ku

ERROR COLLECTION & CURVE FITTING

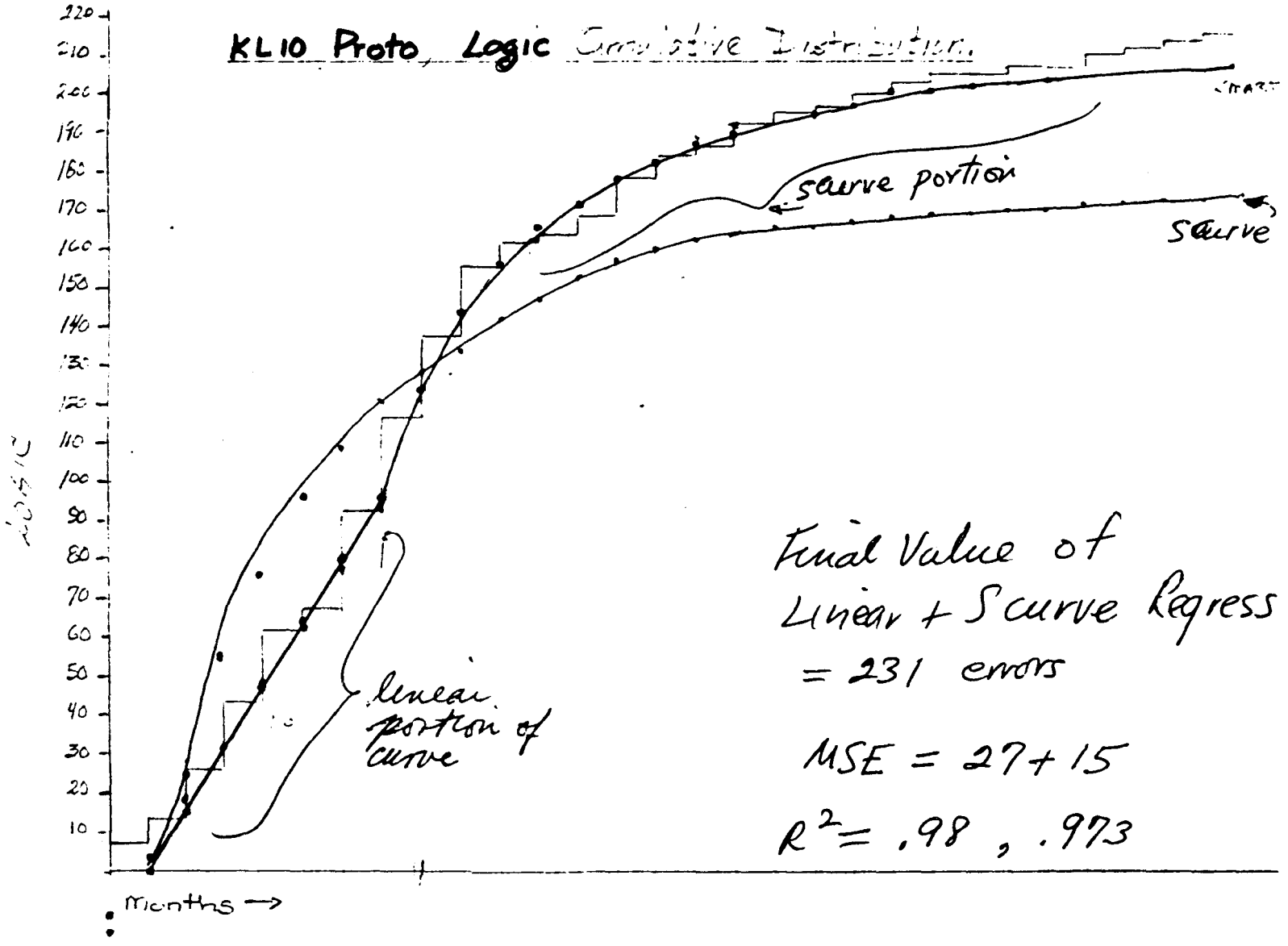
- Available error information on KL10, 11/750 and 11/780 was collected and analyzed.

- Mathematical models developed -

  - Straight Line
  - S Curve

- Resulting data pretty well suited to an asymptotic regression technique.

Exhibit #1



Final Value of Linear + S curve Regress
= 231 errors

MSE = 27 + 15

R^2 = .98 , .973

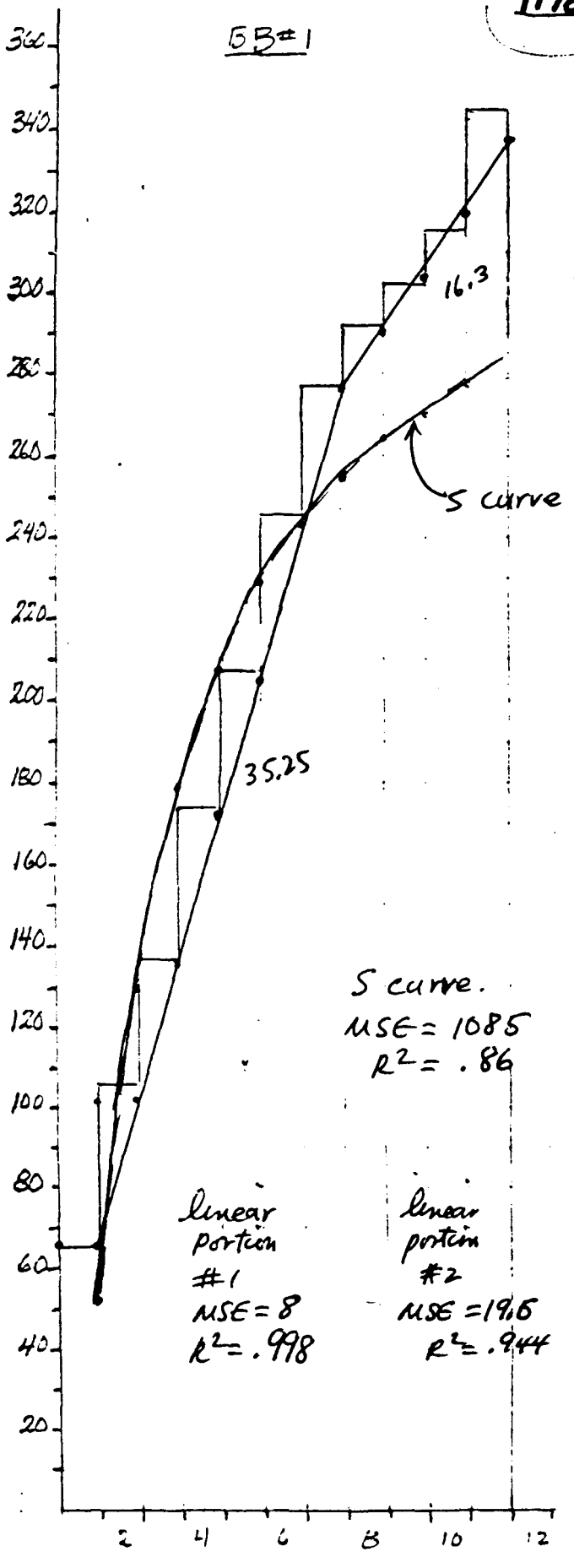
S curve only

MSE = 575

R^2 = .87

EXHIBIT 3

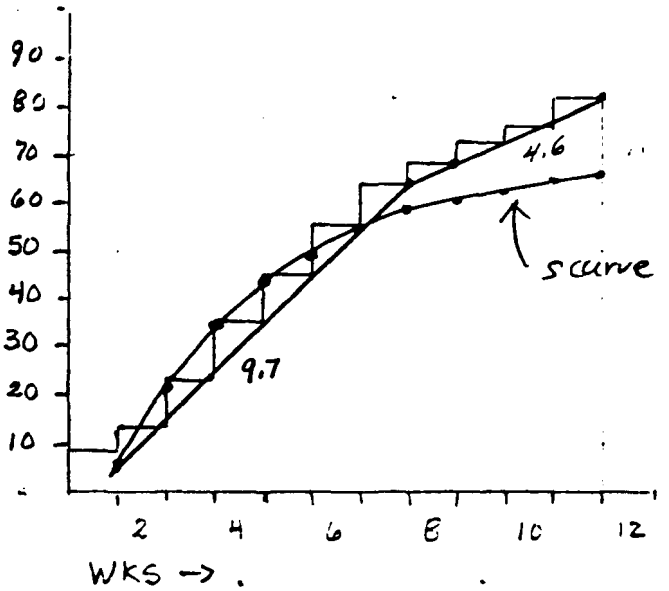
11780



S curve.
MSE = 1085
R^2 = .86

Linear portion #1
MSE = 8
R^2 = .998

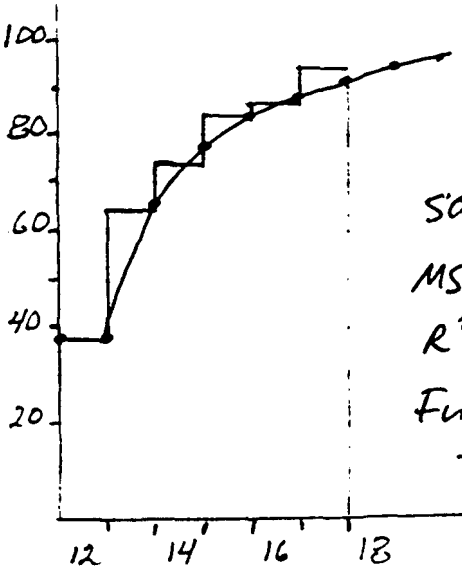
Linear portion #2
MSE = 19.6
R^2 = .944



S curve
MSE = 95
R^2 = .85

Linear Portion #1
MSE = 3.6
R^2 = .99

Linear Portion #2
MSE = .3
R^2 = .988



S curve
MSE = 3.5
R^2 = .989
Final Value = 108

Exhibit #6

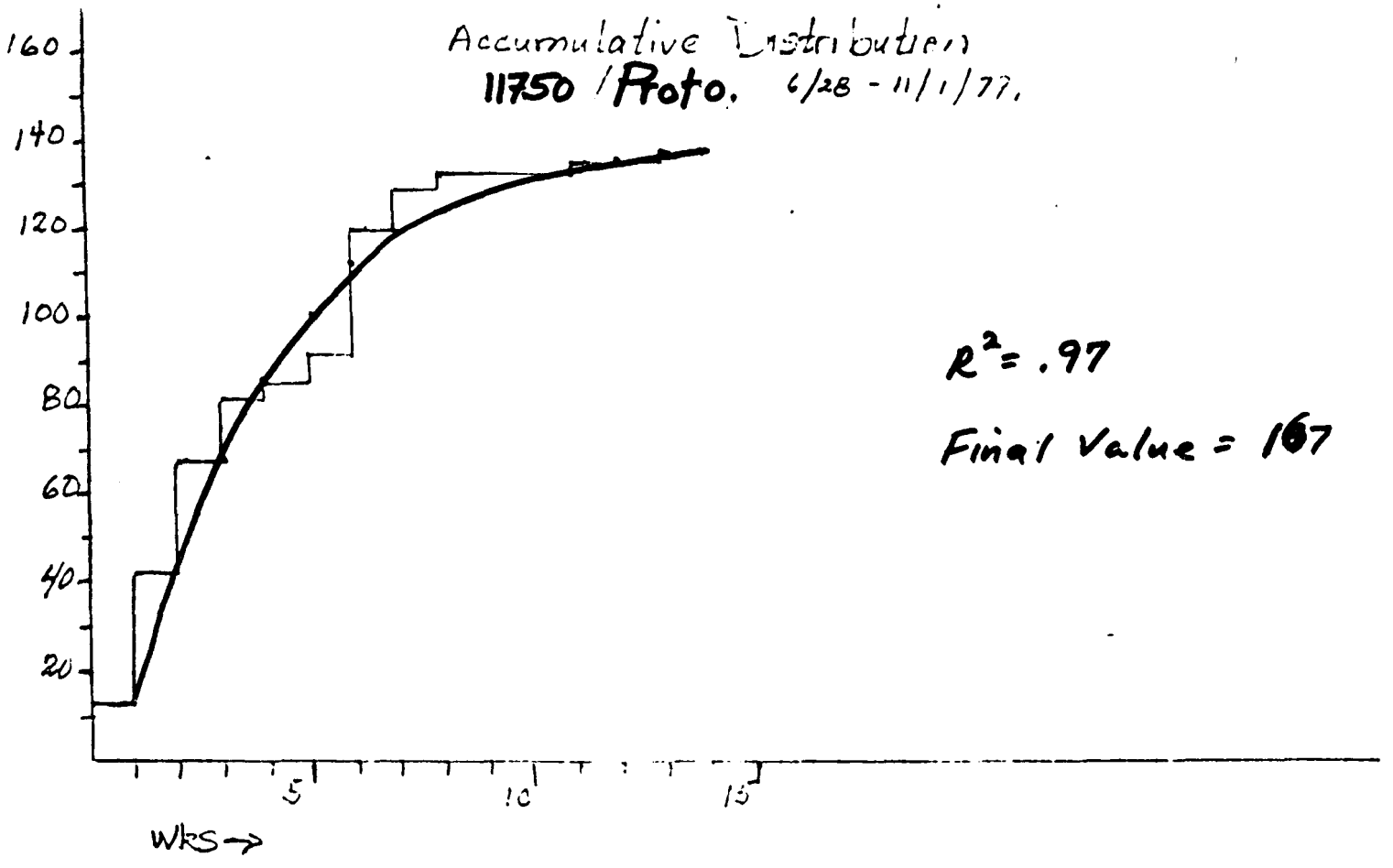

Accumulative Distribution
11750 / Proto. 6/28 - 11/1/77.
160
140
120
100
80
60
40
20
5
10
15
Wks →
$R^2 = .97$
Final Value = 167

Exhibit #7

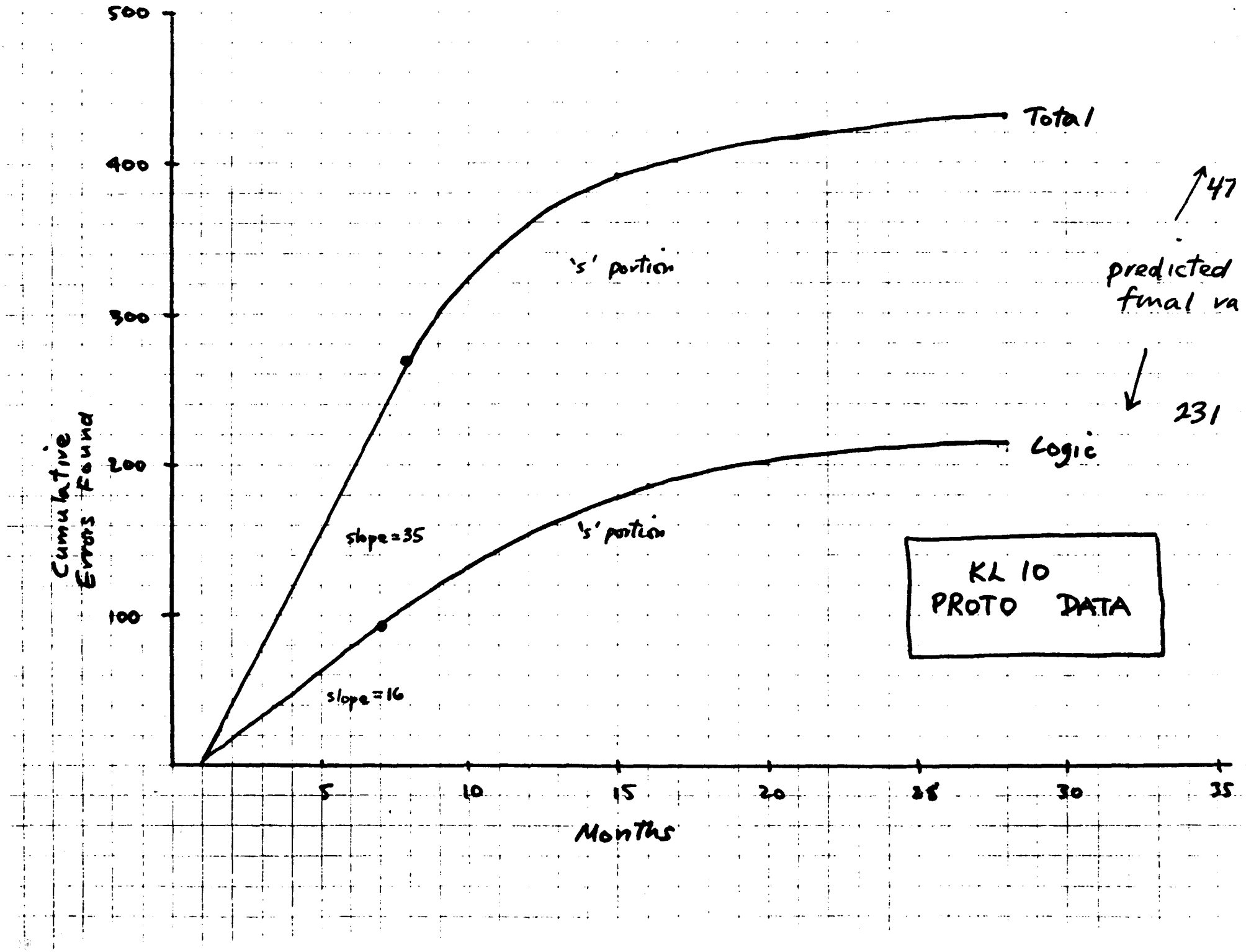
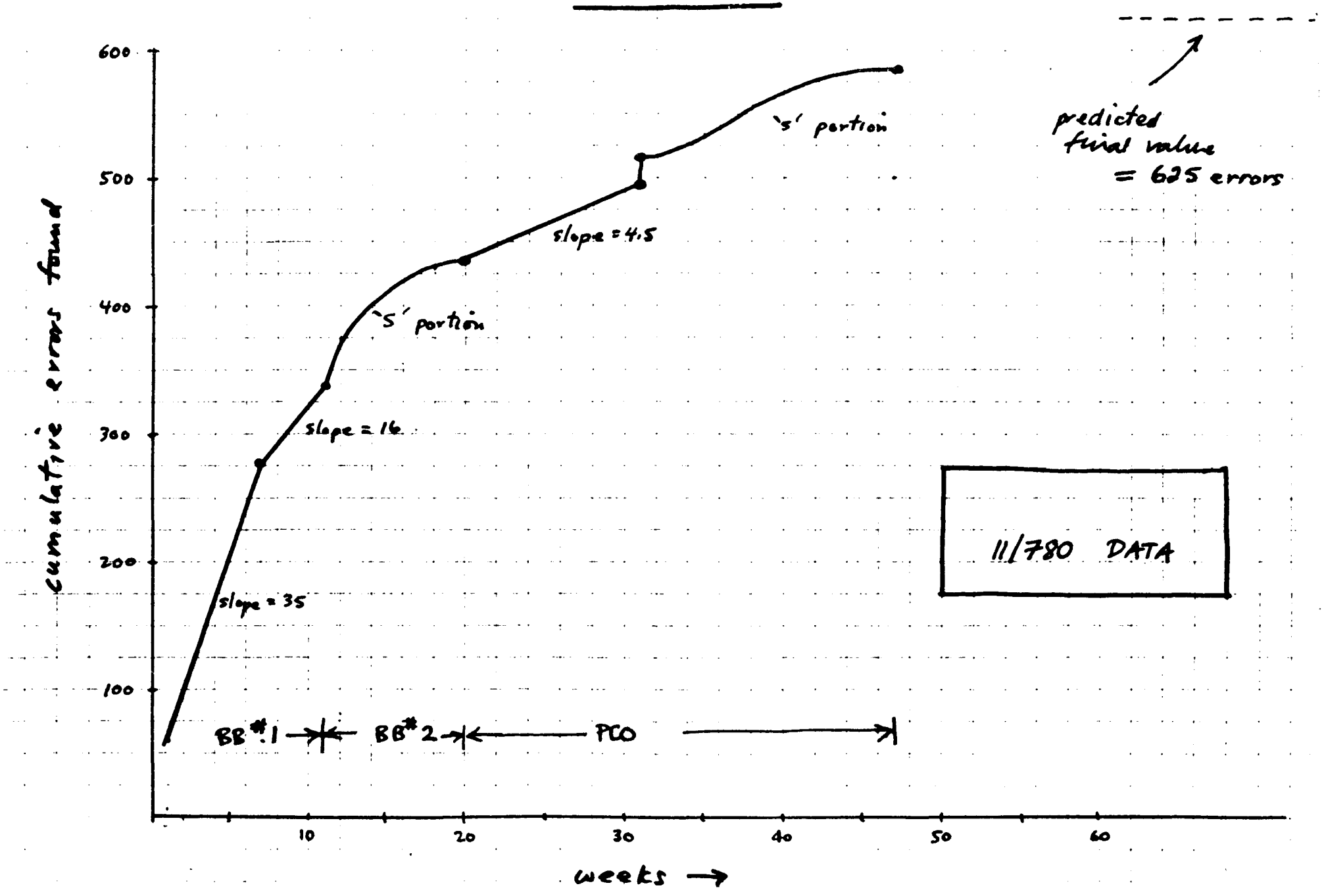

500
400
300
200
100
Cumulative Errors Found
Total
'S' portion
47
predicted final va
231
Logic
'S' portion
slope = 35
slope = 16
KL 10
PROTO DATA
5
10
15
20
25
30
35
Months

Exhibit #8


cumulative errors found
600
500
400
300
200
100
10
20
30
40
50
60
weeks →
slope = 35
slope = 16
'S' portion
slope = 4.5
'S' portion
predicted
final value
= 625 errors
11/780 DATA
BB#1
BB#2
PCO

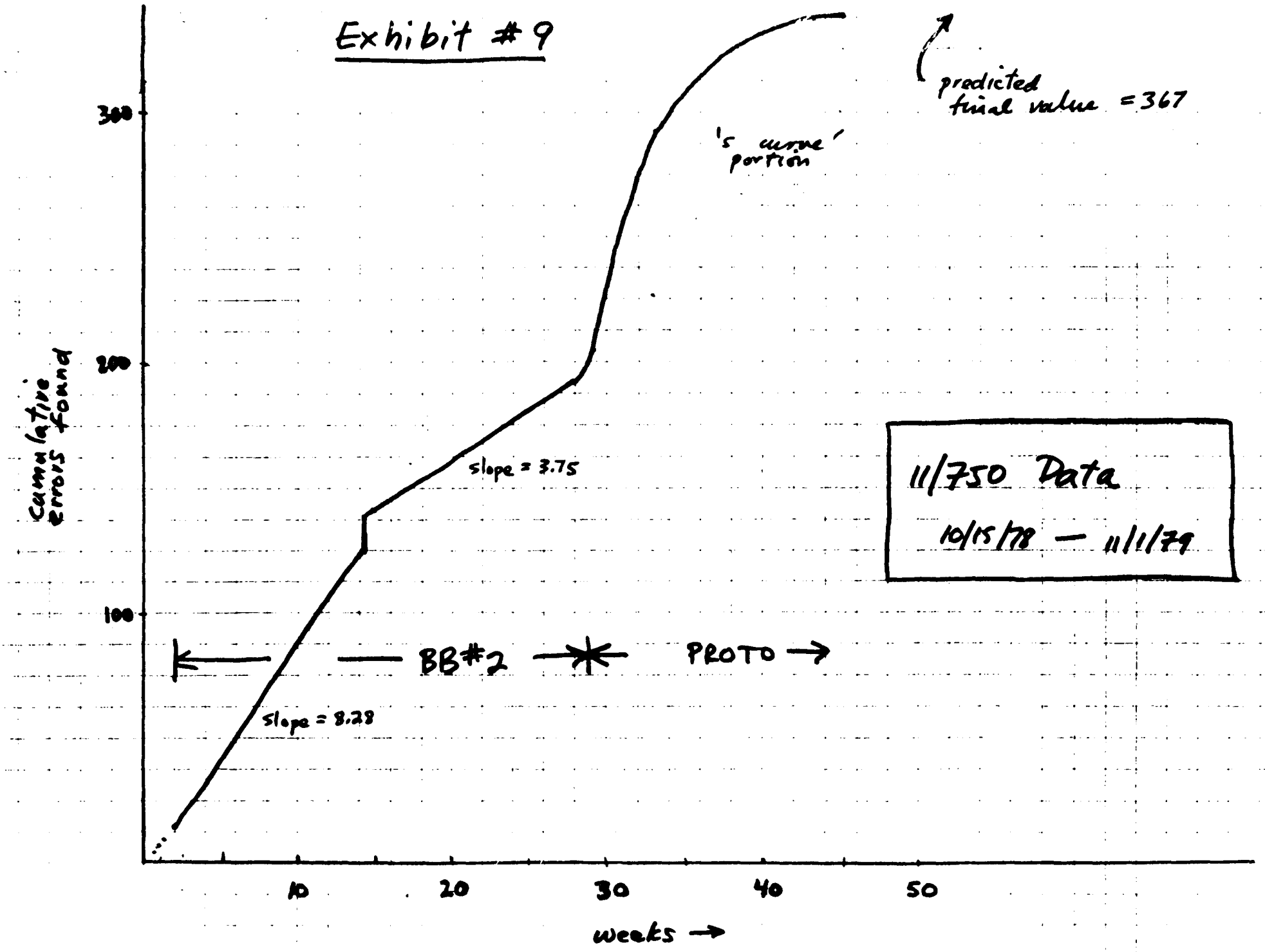


Exhibit #9
predicted final value = 367
'S' curve portion
slope = 3.75
slope = 8.28
11/750 Data
10/15/78 — 11/1/79
BB#2
PROTO
Cumulative errors found
300
200
100
10
20
30
40
50
weeks

ERROR REPORTING SYSTEM:

- Problem Administrator in place - Calvo
- Problem sheet format & process in final review. Expect to be in place by end of July.
- Computerized summary reports on Error Collection and status expected to be in place by early August.

VENUS PROBLEM SHEET NO. _____ Dependent PS _____ Page 1 of _____

Name _____ Date _____ Assigned to _____ Date _____

Area(s) Affected _____|_____|_____|_____

During what stage was the problem discovered ? _____

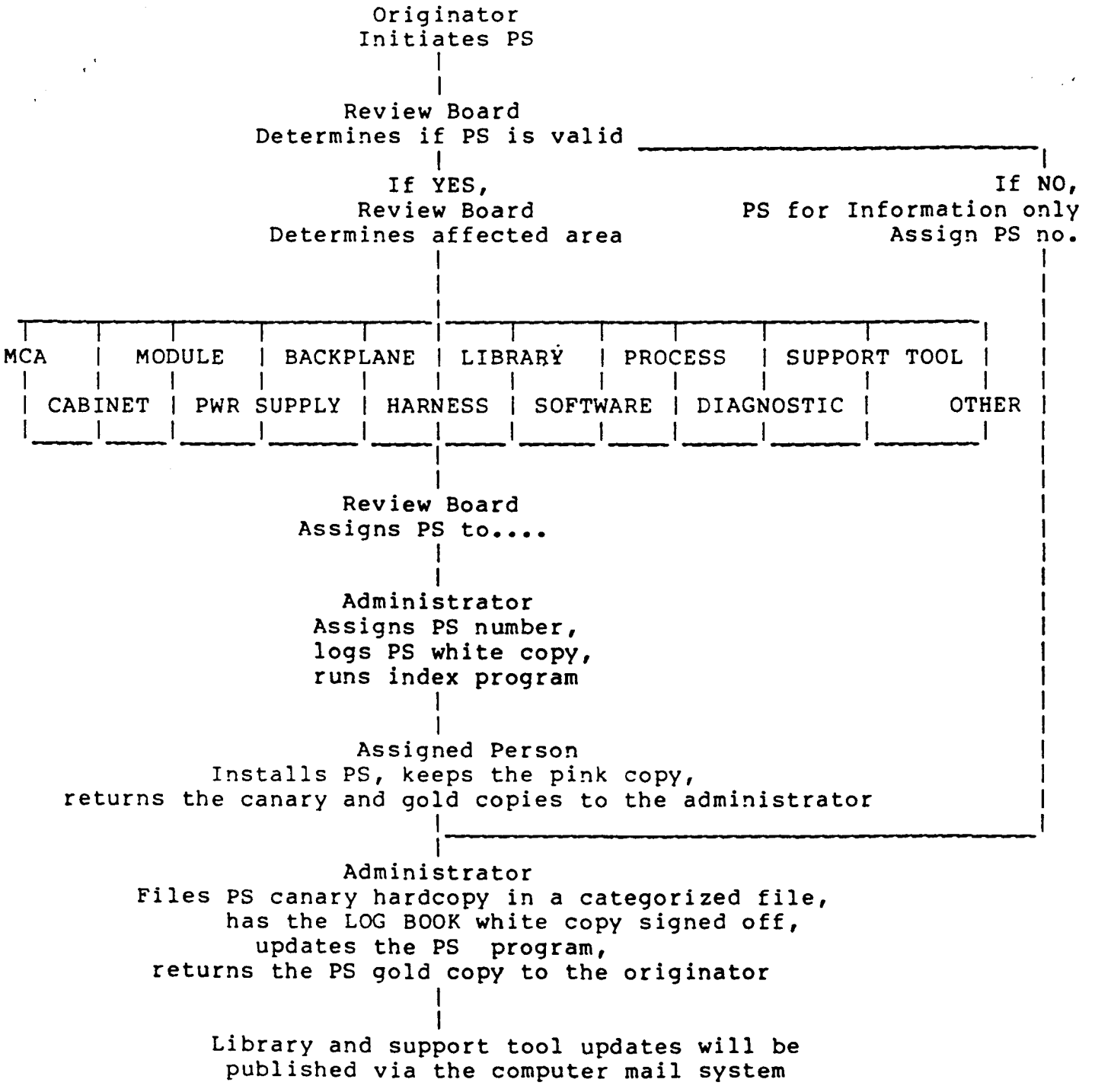
Documentation affected:	Old Rev	New Rev

PROBLEM / SOLUTION (continue on continuation sheet if necessary)

Solution by _____ Implemented by _____ Date _____

White→log book Canary→file Pink→assigned person Gold→originator

3. Condensed PROBLEM SHEET Process



KEY POINT IN DEBUG STRATEGY:

- Errors found in later stages cost much more and take much longer to fix.
- "Zero Defect" attitude must be installed in the Design Team -
  - reduce total error
  - reduce debug time
- MCA logic errors must be less than 50 by Prototype Power-On (no more than 16 emulators can be used) for us to have a high enough confidence in a "functional", if not a full-speed quality prototype.
- Most sever error types are -

- MCA Technology	-	performance, schedule
- MCA Logic	-	schedule
- System	-	performance, schedule
- Environmental	-	standards
- Manufacturing Process	-	Ramp-up, after cost

- Simulation must be effective (95 - 99%) for program to succeed.
- Turnaround times (PC/BP/MCA) must be significantly reduced to reduce schedule and ECO risks.

ACTIVITIES TO BE CONTINUED:

1) Get Debug Strategy finalized and approved.

2) Put Error Reporting System in Operation and start collecting and predicting errors.

3) Walton's "Box Test" scenario needs to be addressed.

4) Work out an optimum strategy in releasing MCA/PC to Layout in the midst of simulation.

5) Work to define and get commitments/approvals on -

   - new filter designs
   - filter ownerships
   - filter entry/exit criteria
   - time estimations on going through filters.

   Overall Debug schedule estimation pending on the above 5 items.

****
* d i g i t a l *
****

TO: *GORDON BELL

cc: ULF FAGERQUIST
GEORGE HOFF

DATE: FRI 31 JUL 1981 10:37 EST
FROM: ALAN KOTOK
DEPT: LARGE VAX DEV.
EXT: 231-7391
LOC/MAIL STOP: MR1-2/E47

SUBJECT: RE: SPECIFYING & TESTING MOD. VS SYS. & THE VENUS M/E TESTER

I would like to get an outside perspective on testing chips, MCAs and modules. I have a rather sick feeling in my stomach about our capabilities. For instance, the Hudson people tell us that they can't test a chip they are making for us to within 2 ns on an 8ns spec, because our testers are so inaccurate. Another for instance is that nobody seems to have a good story how Venus boards which have been isolated as non-working got fixed. I can't believe we'll throw them away.

I would be happy to meet with Mr. D'Arbeloff. I can't say I know anything about the subject. On the other hand, I have concerns with those who supposedly do, but leave us in the position I see us in.

****
* d i g i t a l *
****

TO: see "TO" DISTRIBUTION

cc: JOHN HOLMAN

DATE: WED 5 AUG 1981 13:17 EST
FROM: WARREN MONCSKO
DEPT: TECHNICAL OPERATIONS
EXT: 223-4080
LOC/MAIL STOP: ML21-3/T40

SUBJECT: ENGINEERING PRODUCTION CONTROL SYSTEM

Gordon, you are right; we do need an engineering shop floor control system. We have looked at some shop floor control systems and have determined that the current METRICS System (see the red "ES METRICS User Guide") can fulfill the engineering shop floor control system needs.

There are two basic problems with this system which I propose to change:

1. Currently the system is limited to P.C. design and prototype board build.

2. Within the engineering groups there does not appear to be a high degree of understanding or acceptance of this system.

Consequently, this system is not being used to its capability.

I propose several things be done to remedy these ills. First, I would like to meet with all of you to discuss your concerns and to present a set of objectives toward expanding this shop control system. (I have asked my secretary to set this up - let me know if you disapprove of this meeting.)

Second, the outcome of the meeting should be a list of enhancements that will meet the objectives of improving process flow. These results should be further understood, accepted and modified by a presentation for the PEG members.

Third, projects should be spawned to meet the objectives outlined by Engineering Staff.

One of my concerns in attempting to do this is that the workstations that information is flowing between crosses many areas of Engineering and Manufacturing. Changes will be necessary in all areas, we need to find and work on the big holes first and also to get all concerned pulling toward the same goals.

"TO" DISTRIBUTION:

*GORDON BELL LARRY PORTNER PETER VAN ROEKENS

ATTACHED: MEMO;21

****
* d i g i t a l *
****

TO: WARREN MONDSKO
LARRY PORTNER
PETER VAN ROEKENS

DATE: MON 3 AUG 1981 14:13 EST
FROM: GORDON BELL
DEPT: ENG STAFF
EXT: 223-2236
LOC/MAIL STOP: ML12-1/A51

SUBJECT: RE: ENGINEERING PROCESS IMPROVEMENT PROGRAM

Could you go out and look for some on line shop floor control systems we might use?

We need the type a job shop might use. Let's look at some, select, and try one now!

~~for Eng.~~ + MF

Andrew Could you go out and look for some on line shop floor control systems we might use? Sent

pvr, LP

****
* d i g i t a l *
****

TO: see "TO" DISTRIBUTION gp, LP, PVR

cc: ENG STAFF:

DATE: THU 30 JUL 1981 7:59 EST
FROM: WARREN MONCSKO
DEPT: ENGINEERING SYSTEMS
EXT: 223-4080
LOC/MAIL STOP: ML21-3/T40

SUBJECT: ENGINEERING PROCESS IMPROVEMENT PROGRAM

We need the type a job shop might use. Let's look at some, ~~one, bring~~ select, and try one now!

In recent weeks there has been much talk about improving the engineering design cycle along with the concept of the engineering factory. I would like to point out some thoughts that I have had related to this area and offer my help in making the engineering factory a reality.

I recognize that there are two primary thrusts to deal with: (1) Improved time to market (productivity); and (2) Improved quality. I believe that the engineering factory, as outlined by Gordon, is capable of achieving these objectives.

When I think of an engineering factory I like the analogy with a production or manufacturing factory. Two things are present in a production environment: (1) Parts which stop at points and have value added to the part; and (2) a conveyor belt which moves the parts from workstation to workstation. We have the same situation for an engineering factory, we move a concept from workstation to workstation adding value to the end product. The engineering conveyor belt is the information transmission vehicle. Sometimes, this is computer files, sometimes this is drawings, parts lists, BOMs and sometimes this is modules, backplanes, chips, etc.

It is my opinion that the engineering information flow holds the key to obtaining significant engineering process improvements. To date a lot of the process improvement appears to be focused on improving the time that is spent at the workstations. Recently except within the TOPS staff I have not heard much talk about improving the engineering conveyor belt or information flow process.

The concept of an Engineering Production Control System is what I have been trying to evolve out of the TOPS focus into Information Management. We (TOPS) Staff) do not feel the pull necessary for a production control system. The fact that it takes several weeks for an engineering released drawing to become a manufacturing accepted drawing is an Information Management problem. CAD libraries taking several weeks to be updated with a new "footprint" so that a SUDS output can be run in an IDEA system is an Information Management problem. I'm sure we can name more.

I believe that an equal focus needs to be placed on improving the production control (Information Management) systems as on improving the workstations (technology) available. I also believe that the proposed charter for technical information management is to focus on making the engineering groups more productive with improved quality via Information Management.

TO: Distribution

DATE: 20 July 1981
FROM: Vic Ku,
Barbara Watterson
DEPT: L.S.E.G.
DTN: 231-6202, 231-7527
MAIL STOP: MR1-2/E47

SUBJ: VENUS - Error Detection

We are designing a form to document filter type and effectiveness in detecting errors in the VENUS system. Our goal is three-fold: to forecast errors as early as possible, to engender a zero-defect mind-set, and to design and implement filters that will detect most, if not all, errors before "Engineering Prototype" power-on.

Enclosed are several items for your information, including a sample of the matrix we want you to complete. Please review everything and then complete the blank matrix as follows:

1. Fill in the section giving your name, function, etc.

2. Note any comments about errors relevant to your area.

3. Under "Type of Error", list all relevant error categories.

4. For each category, estimate it as a percentage of all errors and indicate this percentage in the space provided.

5. List any filters not mentioned on the form and evaluate the effectiveness of each, using the code at the bottom of the form.

6. Evaluate the filters listed on the form, using the same code.

7. On a separate sheet, specify which error type(s) each filter is designed primarily to detect.

We will arrange for you to meet with us by the end of July so that you can explain your findings and suggestions.

If you have any questions in the meantime, call either of us.

Thank you for your support.

FUNCTION	RESPONSIBLE PERSON
System	George Hoff
I Box E Box M Box F Box SBIA Console Microcode	Alan Kotok
Mechanical Circuit Power System Environmental Memory Array	Jim McElroy
Construction/Fabrication	Joe McMullen
Manufacturing/Process	John Grose
CAD Tools Simulation Tools	Vehbi Tasar
Simulation (general)	Roy Rezac
VMS	Chuck Samuelson
Software Layered Products	Peter Ross
Diagnostic (CPU) Diagnostic (I/O)	Dick Beaven
MCA/PC Layout Tools MCA Process Loops PC Process Loops	Nick Cappello

TYPE OF ERROR	ESTIMATED PERCENTAGE OF ALL ERRORS	FILTER TYPE & EFFECTIVENESS IN DETECTING ERROR											ENGR. PROTOTYPE	PRELIM. 102/FCC	PRELIM. DVT	PRELIM. DMT	MFG. PROTOTYPES	FINAL 102/FCC	FINAL DVT	FINAL DMT	FIELD TEST	FCS PREPARATION	PMT

VENUS:
ERROR DETECTION
EVALUATION

NAME:

FUNCTION:

DATE:

PAGE___OF___

COMMENTS:

PLEASE EVALUATE FILTER EFFECTIVENESS:

O = 0 - 10% effective
L = 10 - 40% effective
M = 40 - 80% effective
H = 80 - 95% effective
VH = 95 -100% effective

FILTER TYPE & EFFECTIVENESS IN DETECTING ERROR

VENUS: ERROR DETECTION EVALUATION

NAME: John Doe

FUNCTION: IBox Logic

DATE: 7/20/81

PAGE 1 OF 1

TYPE OF ERROR	ESTIMATED PERCENTAGE OF ALL ERRORS	Design Walkthrough	MCA SAGE Simulation	IBox TCMS Simulation	IBox Subfunction Simulation	IBox Simulation	I/E/M IBox Simulation						ENGR. PROTOTYPE	PRELIM. 102/FCC	PRELIM. DVT	PRELIM. DMT	MFG. PROTOTYPES	FINAL 102/FCC	FINAL DVT	FINAL DMT	FIELD TEST	FCS PREPARATION	PMT
Data Path	5%	M	M	H	M	H	VH						VH	L	VH	M	VH	L	VH	H	VH	VH	L
Medium Control Logic	25%	M	M	L	M	H	VH						H	L	VH	M	H	L	VH	H	VH	VH	L
Heavy Control Logic	50%	M	M	L	M	H	VH						H	L	VH	M	H	L	VH	H	VH	VH	L
Error Detection Logic	5%	M	M	L	M	M	H						H	L	VH	M	H	L	VH	H	VH	VH	L
MBox Interface Logic	10%	M	L	L	L	M	VH						VH	L	VH	M	VH	L	VH	H	VH	VH	L
SBox Interface Logic	5%	M	L	L	L	M	VH						VH	L	VH	M	VH	L	VH	H	VH	VH	L

COMMENTS:

Additional filter needs to be worked out to catch more "Error Detection" errors before Prototype power-on

PLEASE EVALUATE FILTER EFFECTIVENESS:

O = 0 - 10% effective
L = 10 - 40% effective
M = 40 - 80% effective
H = 80 - 95% effective
VH = 95 -100% effective

APPENDIX TO ERROR DETECTION MATRIX:

1. System -

- Performance
- System Loading/Interaction
- Hydra Configuration
- Revision Control
- 2 SBI's Systems
- Specifications
- Bus Bandwidths
- Memory & Memory Expansion

2. Logic -

- Clock System
- Data Path
- Control Logic
- Timing
- Performance
- Interrupt Logic
- Memory Management Logic
- Error Detection Logic
- Error Correction/Recovery Logic
- Error Logging Logic
- Specification Discrepancies/Inconsistencies
- Margins (MCA, PC, BP)

3. CAD/Simulation Tools -

- Loading Rules
- Design Rules
- Libraries
- Delay Times
- Delay Analysis Tools
- Terminator Checking
- Simulation Tools
- Data Base Management

4. Microcode -

- Performance
- Specifications
- Functionality
- Loading of
- Error Detection
- Error Correction/Recovery
- Error Logging

5. & 6. VMS & Layered Products -

- Specifications
- 2 SBI's
- Console Handler
- Error Detection
- Error Correction/Recovery
- Error Logging
- Loading of
- Performance
- New I/O Handlers
- System Loading/Interaction
- Hydra Configuration

7. Diagnostic -

- Fault Coverage
- Performance
- 2 SBI's
- Functionality
- Error Recovery
- Error Logging
- Error Detection & isolation
- Human Engineering
  - Ease of use
  - Sequencing
  - Loading of
  - Output format capability
- Remote Diagnostic Capability
- I/O Diagnostic

8. Circuit/Power System

- Emulator (circiut)
- Circuit Rules
  - Loading
  - XOR
  - Crosstalk
  - Reflection
- Component Qualification/Spec/Screens
- Sourcing/Compatibility
- Design Specifications
- Functionality
- Failure Modes
- Sockets (electrical)
- MTBF of Components
- EMM
- Margins

9. Mechanical -

- Emulator (mechanical)
- Cooling System
- Mech Fab Design
- Connector Design
- Cable Design
- Sourcing
- Socket (mechanical)
- Margins

10. Environment -

- FCC
- 102
- 50 Hz
- Acoustics
- VDE

11. Construction/Fabrication -

- Open & Short Etches
- Bad Components
- Missing Terminators
- Components Mounted Wrong
- ECO Procedures
- Incoming Inspection Procedure & Tester for component, MCA, PC, BP, Mechanical Fab, & Cables.

For: McJannis, Croron, Bob Stewart,

****
d i g i t a l *
****

some thought on Testers and modules

FYI — no rambles

)): see "TO" DISTRIBUTION

DATE: TUE 28 JUL 1981
FROM: GORDON BELL
DEPT: ENG STAFF
EXT: 223-2236
LOC/MAIL STOP: ML12-1/A51

:: ENG STAFF:

JBJECT: SPECIFYING & TESTING MOD. VS SYS. & THE VENUS M/E TESTER

Alex D'Arbeloff, President of Teradyne, asked me if we intend to /olve to completely specified modules, versus ones which are ufficient to operate in a system. Apparently IBM and Western lectric do this and obtain a field failure rate of 2.5% modules/year ersus 7% for us (Is size a factor?). Steve Davis, are we making the rong design trade-off?

As a seperate, but highly related issue, I asked for a logic estor that would test MCA's, boards and boxes at speed so that we ould systematically test and integrate the parts of VENUS for ngineering. This would also become the manufacturing test.

He and some of his staff would like to visit us regarding nese... especially Venus.

TO" DISTRIBUTION:

TEVE DAVIS ULF FAGERQUIST GEORGE HOFF
LAN KOTOK

****
* d i g i t a l *
****

Answer cc: Ulf

Can you identify alternative resources & so this isn't a meeting to

TO: see "TO" DISTRIBUTION

cc: JIM CUDMORE

DATE: FRI 31 JUL 1981 13:29 EST
FROM: ROY REZAC
DEPT: MR SITE ENG'G MANAGER
EXT: 231-4140
LOC/MAIL STOP: MR1-2/E18

SUBJECT: REVIEW OF SIMULATION STRATEGY

Grab & Scrapes's ?

On August 11, I scheduled with you a review of the VENUS Simulation Strategy. The purposes of this meeting are as follows:

A. To review with you our Simulation Strategy for your technical and management review.

B. To gain your support for this effort, and/or identify key issues that you see.

C. To decide on the scenario for obtaining the key simulation resources that are needed. (Assignment of experienced Simulation Process Engineers from current projects to VENUS for periods of 3 - 6 months.)

My approach to process this meeting is as follows:

A. We at Marlboro will have a dry run prior to the actual presentation to you to make sure we've resolved all of our questions and can adequately explain them.

B. Since it is very important to get an overview of the proposal, I will ask that we take clarifying questions during the presentation, and hold more in-depth questions until the end.

C. I will attempt to present the six key items that are critical to the Simulation, with several slides as backup material.

The six key items that I will address will be as follows:

1. Overall Debug Strategy Summary
2. Simulation Goals and Strategy
3. Hardware/Software Debug Comparisons
4. Key Simulation Metrics
5. Proposed Schedules and Key Resource Needs
6. Key Open Issues

We will, of course, be able to respond to more detailed questions in any given area, and we'll have the technical people there to answer any questions you may have.

After approximately 1-1/2 hours on this overview, I would like to adjourn to Ulf's office where we will make the final decision to which scenario we will use for obtaining the Simulation resources. Our best estimate is as follows:

A. SCENARIO I

If the key resources identified in my June 26 EMS are assigned by September 1, then the Simulation Project will be on track with the replanned Logic Design.

B. SCENARIO II

If the approach is to use "good programmers with hardware background" to develop the process, then the Simulation effort will lag the Logic Design by 1-1/2 - 3 months and the probability of project success is reduced from 90% to 70%.

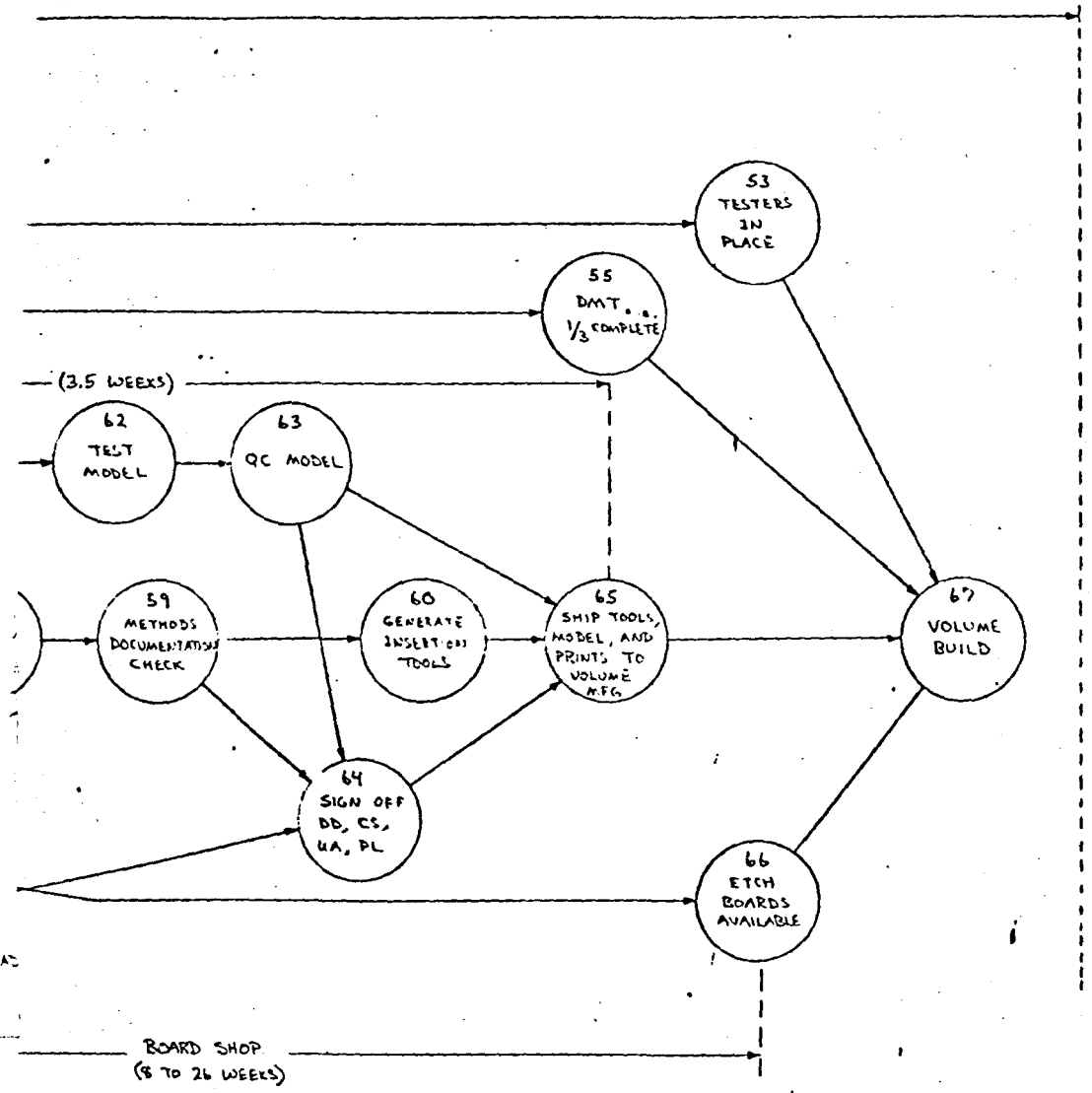
After this decision, I will implement the agreed upon scenario.

I'm sending you this process approach and agenda ahead of time, so you can review it, and comment. I need your help to make sure that the process in the meeting is effective, and will allow us to present the information, get the important questions answered, and speed communication. Do you have any other comments on the plan for this meeting?

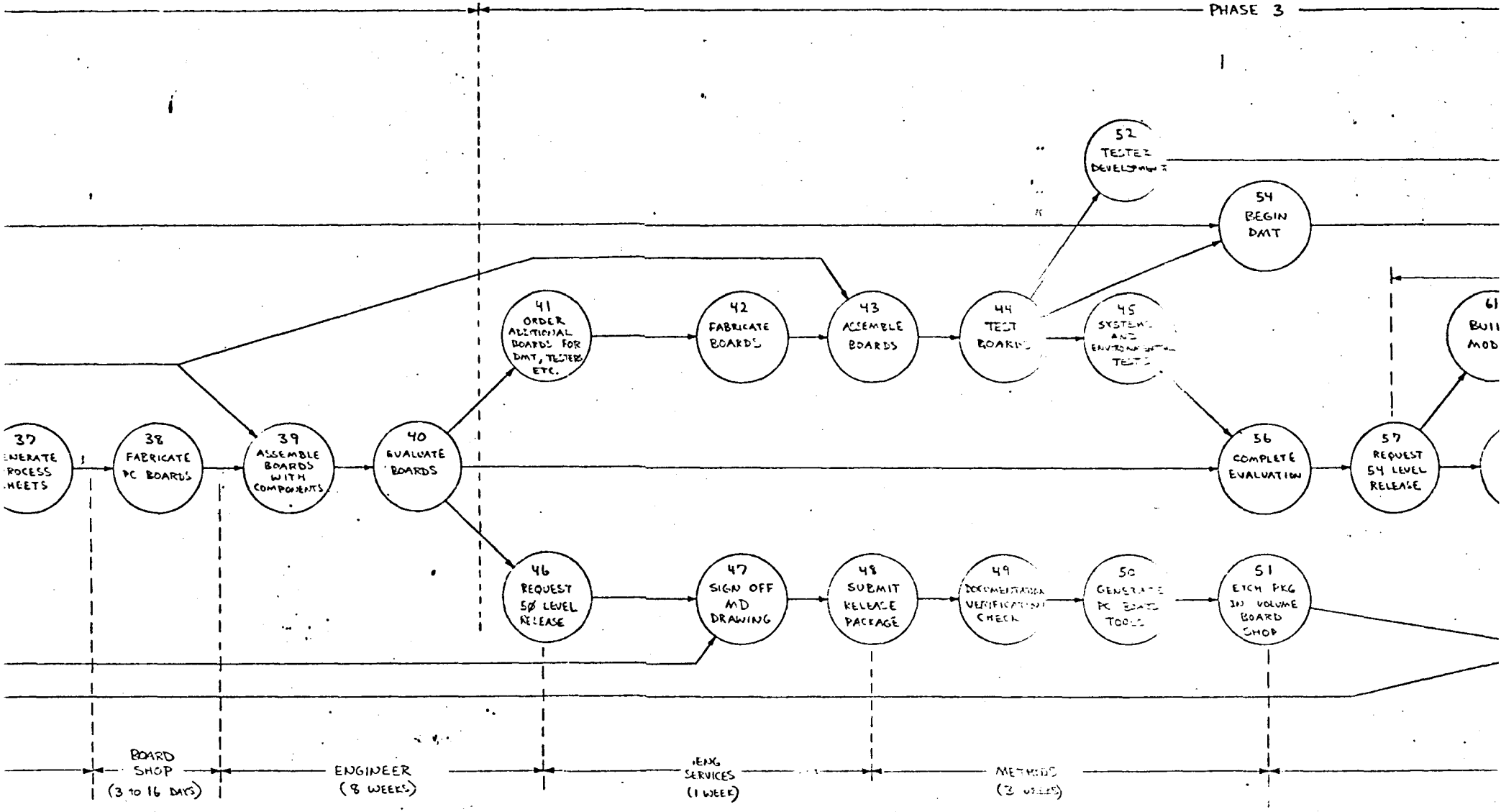
RF:sm

"TO" DISTRIBUTION:

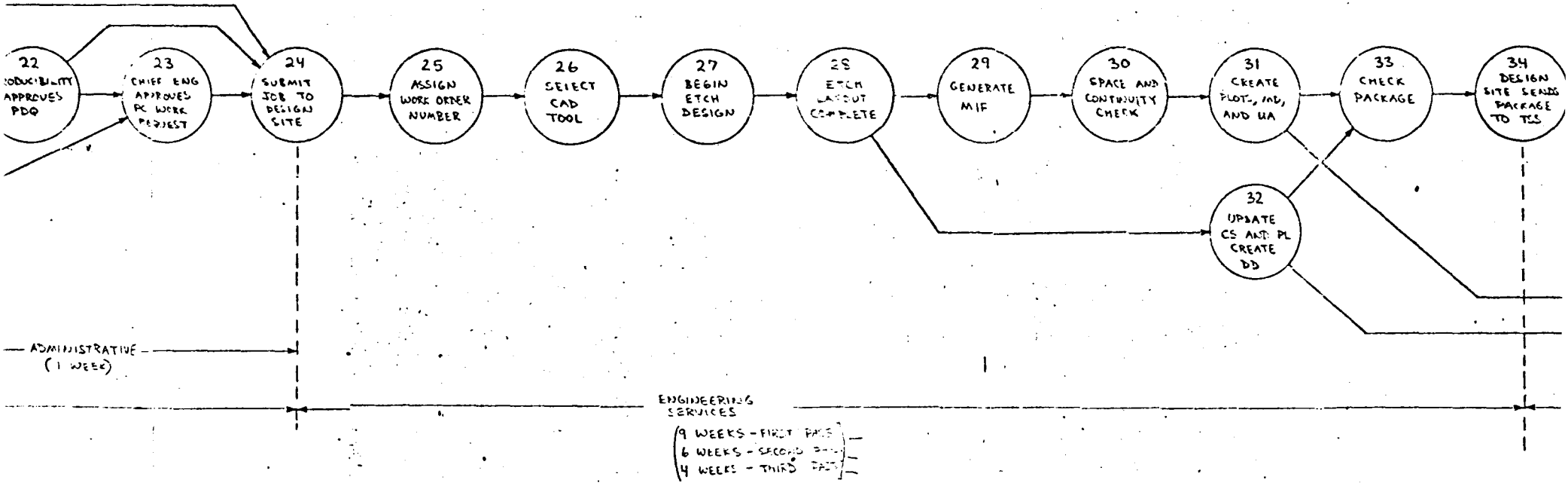
%GORDON BELL ULF FAGERQUIST GEORGE HOFF
ALAN KOTOK

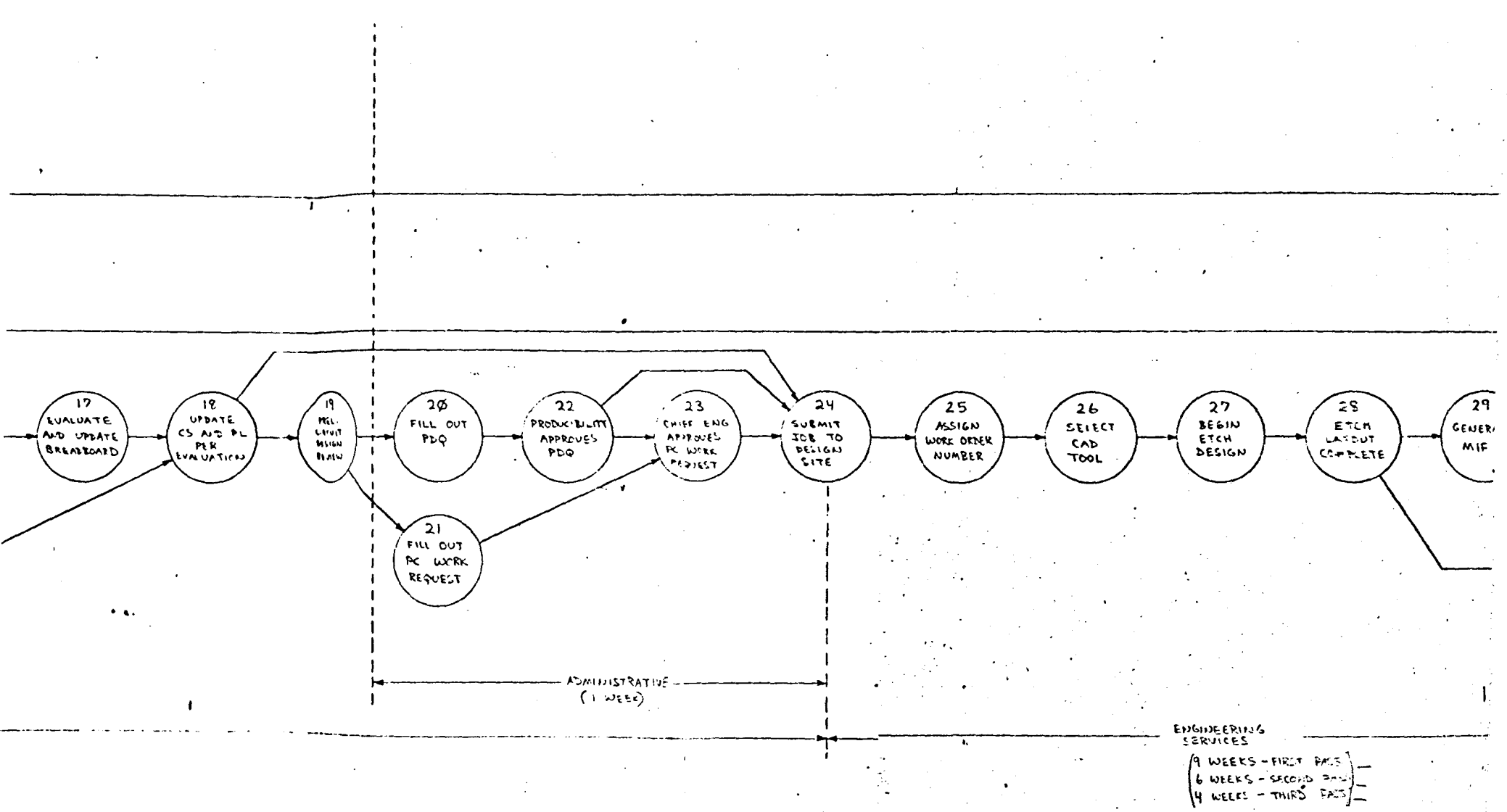


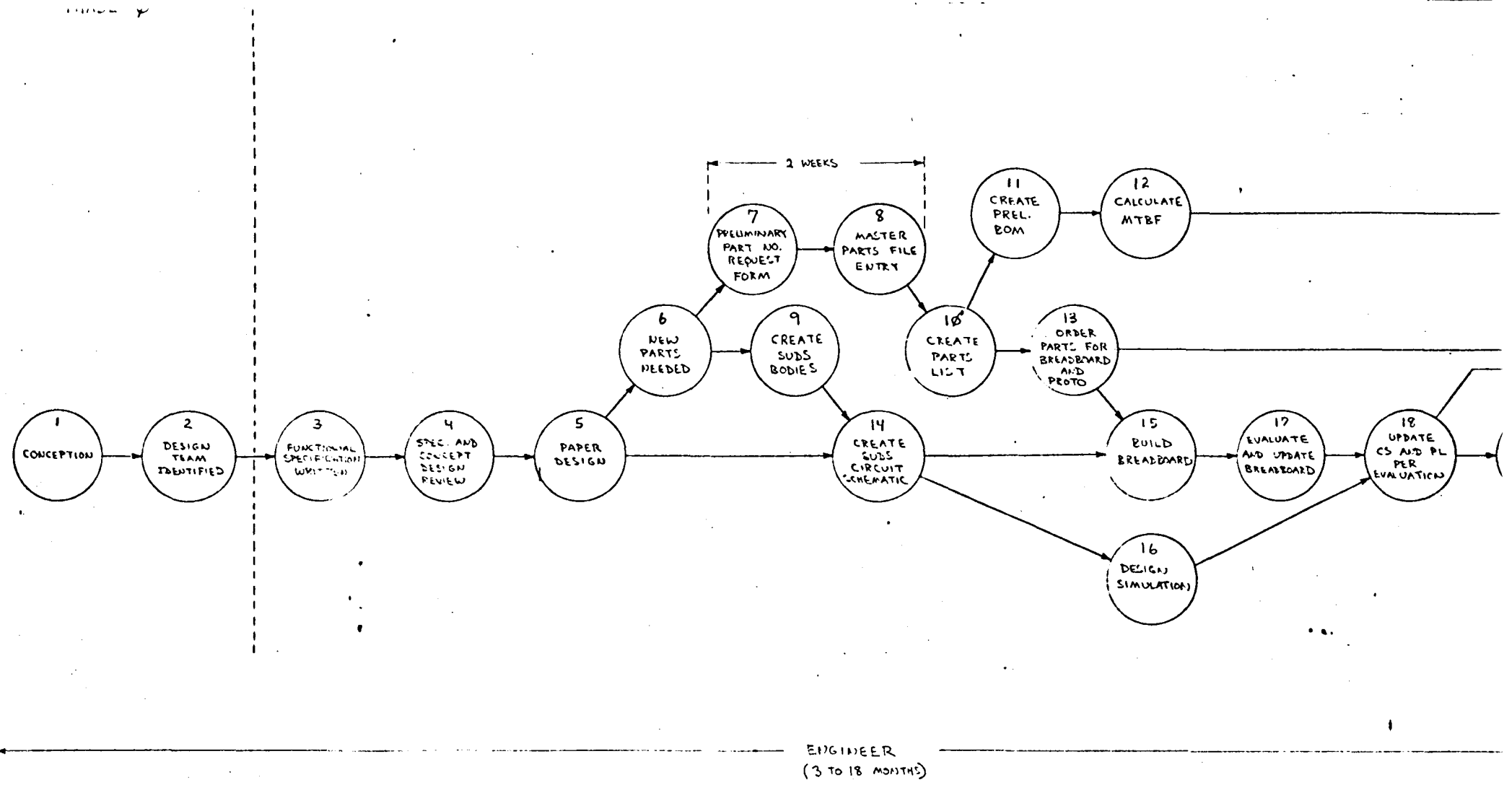
MODULE DESIGN DESIGN


PHASE 3
37 GENERATE PROCESS SHEETS
38 FABRICATE PC BOARDS
39 ASSEMBLE BOARDS WITH COMPONENTS
40 EVALUATE BOARDS
41 ORDER ADDITIONAL BOARDS FOR DMT, TESTERS ETC.
42 FABRICATE BOARDS
43 ASSEMBLE BOARDS
44 TEST BOARDS
45 SYSTEMS AND ENVIRONMENTAL TESTS
46 REQUEST 50 LEVEL RELEASE
47 SIGN OFF MD DRAWING
48 SUBMIT RELEASE PACKAGE
49 DOCUMENTATION VERIFICATION CHECK
50 GENERATE PC BOARD TOOLS
51 ETCH PKG IN VOLUME BOARD SHOP
52 TESTER DEVELOPMENT
54 BEGIN DMT
56 COMPLETE EVALUATION
57 REQUEST 54 LEVEL RELEASE
BOARD SHOP (3 TO 16 DAYS)
ENGINEER (8 WEEKS)
ENG SERVICES (1 WEEK)
METHODS (3 WEEKS)

PHASE 2


22
APPROVES PDQ
23
CHIEF ENG APPROVES PC WORK REQUEST
24
SUBMIT JOB TO DESIGN SITE
25
ASSIGN WORK ORDER NUMBER
26
SELECT CAD TOOL
27
BEGIN ETCH DESIGN
28
ETCH LAYOUT COMPLETE
29
GENERATE MIF
30
SPACE AND CONTINUITY CHECK
31
CREATE PLOT-, IND, AND UA
32
UPDATE CS AND PL CREATE DD
33
CHECK PACKAGE
34
DESIGN SITE SENDS PACKAGE TO TSS
ADMINISTRATIVE
(1 WEEK)
ENGINEERING SERVICES
9 WEEKS - FIRST PASS
6 WEEKS - SECOND PASS
4 WEEKS - THIRD PASS


17
EVALUATE AND UPDATE BREADBOARD
18
UPDATE CS AND PL PER EVALUATION
19
20
FILL OUT PDQ
21
FILL OUT PC WORK REQUEST
22
PRODUCIBILITY APPROVES PDQ
23
CHIEF ENG APPROVES PC WORK REQUEST
24
SUBMIT JOB TO DESIGN SITE
25
ASSIGN WORK ORDER NUMBER
26
SELECT CAD TOOL
27
BEGIN ETCH DESIGN
28
ETCH LAYOUT COMPLETE
29
ADMINISTRATIVE
(1 WEEK)
ENGINEERING SERVICES
9 WEEKS - FIRST PASS
6 WEEKS - SECOND PASS
4 WEEKS - THIRD PASS


2 WEEKS
1 CONCEPTION
2 DESIGN TEAM IDENTIFIED
3 FUNCTIONAL SPECIFICATION WRITTEN
4 SPEC. AND CONCEPT DESIGN REVIEW
5 PAPER DESIGN
6 NEW PARTS NEEDED
7 PRELIMINARY PART NO. REQUEST FORM
8 MASTER PARTS FILE ENTRY
9 CREATE SUBS BODIES
10 CREATE PARTS LIST
11 CREATE PREL. BOM
12 CALCULATE MTBF
13 ORDER PARTS FOR BREADBOARD AND PROTO
14 CREATE SUBS CIRCUIT SCHEMATIC
15 BUILD BREADBOARD
16 DESIGN SIMULATION
17 EVALUATE AND UPDATE BREADBOARD
18 UPDATE CS AND PL PER EVALUATION
ENGINEER
(3 TO 18 MONTHS)

AUG 8 1980

digital

INTEROFFICE MEMORANDUM

TO: DISTRIBUTION

DATE: AUGUST 5,1980
FROM: Ken Brabitz
DEPT: LSI Program Managment
EXT: 225-5059
LOC/MAIL STOP: HL-M05

SUBJECT: VENUS July Monthly Report

To: Russ Doane
Dick Clayton
ULF Fagerquist
Steve Teicher
Bill Long
Gordon Bell
Jim Cudmore
Bill Demer
Bill Green
Alan Hanover
Roy Moffa
Larry Portner
Jack Smith
Will Thompson
Peg Wesley
Joe Zeh
Joe Mangiafico
Yic Ku
Charlie Bradshaw

Enclosed is the second of two Russ Doane Memo's and an extract from my Monthly Report. The first Memo from Russ shows the allocations for all LSI FY81 Moneys. The time has come to feed or shoot. If we are in the LSI Business lets Feed the projects. If not pick someone to shoot one or more but not VENUS! If some Soloman cannot choose, feed them all before one or more of them starve to death.

- FUNDING ISSUES

By now most everyone has seen their FY81 budgets. I am sure most

everyone is worrying about their own lack of funds. I also have problems. The LSI VENUS program is in serious trouble. The program has two major tasks to accomplish. 1) Qualify the MCA. 2) Qualify Hudson as the second source for MCA's. Both of these tasks accomplished prior to product announcement.

The Corporation has chosen VENUS as the VAX replacement for mid 80's FCS. The bundled system dollar revenue from VENUS over its product life should be no less than $12 Billion and more like $15 Billion. To make this happen, the MCA has to be qualified and a second source has to be qualified.

The LSI operation in Hudson is key to the MCA. The MCA and Hudson's ability to produce it are the forerunner of another corporate product, ie. NAUTILUS. In addition, the long term corporate strategy calls for an LSI Engineering and Manufacturing operation. Failure to develop this LSI capability will leave DEC in a very vulnerable, possibly even a comatose non-competative, position by the late 80's. The Hudson operation has been identified as the prime mover in DEC's LSI strategy. For Hudson to become viable they need a certain critical mass. This critical mass is supported by a certain level of production. Being the second source for the VENUS MCA's will help build this critical mass. This critical mass is built up of highly motivated highly skilled people. To attract and or keep these people in the current and future job market, the message has to be absolutely clear. The message has to be: DEC IS IN THE LSI BUSINESS AND WE INTEND TO STAY THERE IF NOT BE A LEADER. The message today is not only not clear, it is practically invisible it the LSI budget is used as the measure.

Specifically Hudson should be entitled to funding from three sources for VENUS. The three sources are E69-New Product Start Up, E97-Adavanced Development for Manufacturing, and E96-Second Source. The initial Hudson budget request for VENUS LSI was $1.32M. E97 was asked to provide $1.26M. E97 was able to provide $774K. Another $50K was utilized from FY80's to pay $50K of the $150K to Motorola for the technology transfer. (See Attachment C, Russ Doane's memos of 15 and 22 July.) This leaves a net shortfall of $536K for FY81.

This shortfall of $536K has a divided impact. $350K was earmarked

for a monitering and evaluation study. This study has little or no impact on VENUS itself, but could be of value to DEC. The objective of the study was to moniter the MCA qualification and the Hudson start-up. At the end points of both these activities a postmortum would be held. The outcome of the postmortum would be a comprehensive list of the things that went right and wrong during the program and the reasons. This list could form the background for future program plans, business plans, engineering specifications and manufacturing specifications. These plans and specifications could prove valuable for all future LSI programs. Removing this $350K from the $536K shortfall leaves an actual impact on VENUS of $186.

What really happens to VENUS without this $186K? The number of wafer starts gets cut by 50%. This is a new process involving new equipment and technology. By cutting the number of wafer starts in half the process maturity will take longer and less different MCA options will go through the process. There are 36 different MCA's in the base VENUS system. At the same time process engineering will be cut. There are two ways to bring in a new process. It can be done by trial and error or it can be engineered. The two are not mutually exclusive. In trial and error it takes more wafer starts to learn how the process should be run. It also takes failure analysis which implies the use of process engineers. By using process engineering up front it may be possible to start less wafers. To try to cut both on the same project is to the toll the bell of failure before the project even starts. By cutting the number of wafer starts there will be an incidental manpower savings in FY81. This could prove to be a false savings because a base of trained manpower has to be in place to produce product. When LCG needs MCA's for shipments does not seem to be the time for training new manpower. The $186K shortfall will in all probability delay VENUS announcement by one quarter.

I spoke once before of the need for highly trained highly motivated people. Joe Mangiafico is one of these people. He has been through the COMET start up and is online for VENUS. To devote himself and his organization full time to VENUS he needs $60K more than he has for FY81. This $60K was not sought from E97 and should probably come from either E69 or E96 or a combination of the two.

To make VENUS an LSI reality and to meet FCS an additional $246K has to funded NOW! The additional investment of $246K in FY81 against the future payback of $12 or $15 BILLION in VENUS revenue appears to be the only path of sanity. $246 of additional funding now gives Hudson a solid footing to do VENUS right and on time.

To This page only + ref.

Who's going to do our thinking? Planning Strategy here? Process? Gordon

JUL 2 3 1980

digital

INTEROFFICE MEMORANDUM

TO: Dick Clayton
Ulf Fagerquist
George Hoff
Steve Teicher

DATE: 22 July 1980
FROM: Russ Doane
DEPT: LSI Mfg. & Eng.
EXT: 223-4482
LOC/MAIL STOP: ML1-4/B34

CC: Distribution list
SUBJECT: WHY WE MUST STOP ONE MAJOR PRODUCT

As you can see from the enclosed summary of E97 (Mfg. Process Development) Funding for the Seminconductor Cluster, we have no funding for further MOS process development after HMOS.

Within the next few weeks, our advanced process development people will begin to find other employment. And not just the MOS folks: our bipolar people are going to see the following handwriting on the wall too:

"DEC has decided to back out of LSI Manufacturing by the mid-80s".

SCORPIO will not be sourced internally. The MOSAIC I process for VENUS could be crippled too, by attrition due to our evident lack of committment.

Since the vendor base is inexorably shrinking, we're not just going out of the LSI business. We're capping off DEC.

I think we should stop a major product development instead, so we'd have the money to stay in the LSI ball game.

This lumps the whole mess together (How're we going to segment?)

Delagi
Tomasic

Distribution List

Gordon Bell
Ken Brabitz
Jim Cudmore
Bill Demer
Bill Green
Alan Hanover
Roy Moffa
Larry Portner
Jack Smith
Will Thompson
Peggy Wesley
Joe Zeh

Ken Thompson Infinite

My continued questioning of our overall Semiconductor strategy is absolutely born out by this memo. It only addresses one year. We cannot continue to exist in this hand-to-mouth fashion on a year-to-year basis. The whole issue of plant equipment totally swamps these costs. Also our difficulty with proprietary parts levers mfg. even more than these costs!

Do we have a major strategic set of questions: how many technologies can we mfg.? develop.?? MOS or Bipolar/ECL ?

↑ all the effort, main competitor, "the future"

↑ all the revenue

Should we buysout [illegible] (or give up low end etc)?

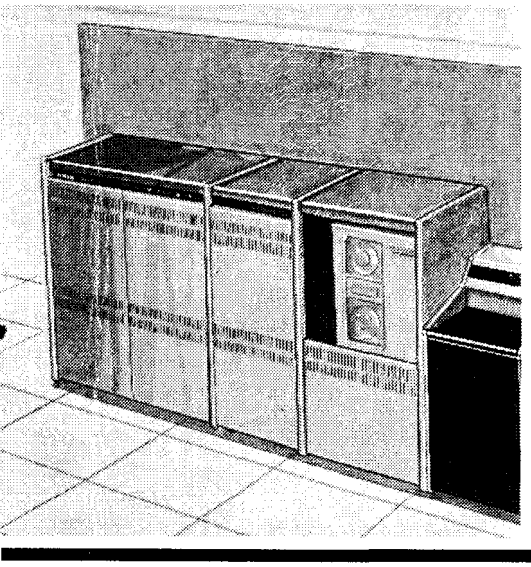
VAX-11/VENUS

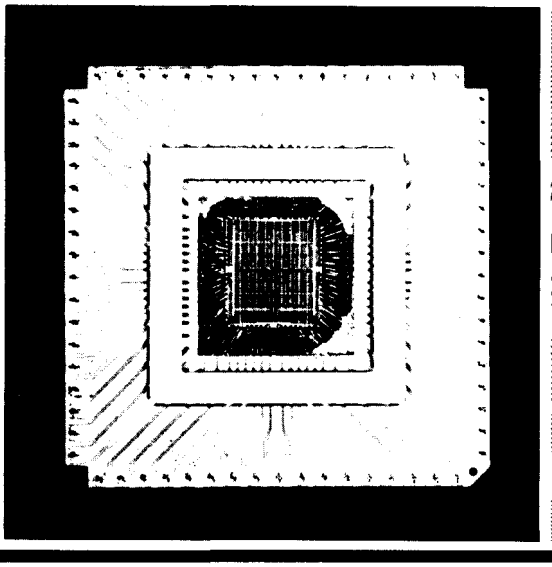
VENUS TASK FORCE

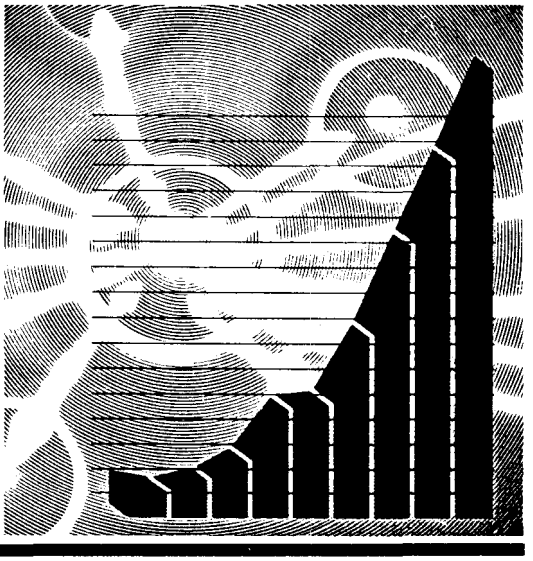
FINAL REPORT

Vol. 1 of 2

June 26, 1980







COMPANY CONFIDENTIAL

digital

VENUS TASK FORCE

FINAL REPORT

TABLE OF CONTENTS

VOLUME 1

++++++++++++++
!D I G I T A L!
++++++++++++++

INTEROFFICE MEMORANDUM

TO: Venus Task Force

CC: VTFWG

DATE: June 17, 1980
FROM: Len Kreidermacher
DEPT: LSEG
LOC: MR1-2/E18
EXT: 231-6617

SUBJ: VENUS TASK FORCE FINAL REPORT

ASSIGNED TASK

The task was assigned by a Win Hindle memo dated July 10, 1979:

> "Study the total life cost of Engineering, manufacturing, Installation, Warranty, and Service for VENUS systems. Develop alternatives for spending to minimize total costs and maximize customer satisfaction for this product, given IBM as chief competition."

RECOMMENDATIONS

The VENUS Task Force makes the following recommendations. The appendix contains additional detail.

1. LCM (Lifecycle model) definition and requirements

   Automate LCM requirements. Business Planning tool would have both "tradeoff" and "business plan" (BURP) modes.

   The requirements are:

   1) Automated assimilation of data from current and additional functional planning models (eg., LCBM, selling cost model)

   2) Automated sensistivity analysis showing:

      a) changes in strategic variable with offsetting effects on NPV
      (if Δ Time to market = 2Q's;
      what Δ Transfer cost offsets this?)

      b) risk assessment of design alternative
      (Δ failure rate makes Δ NPV equals 0)

These recommendations are consistent with Corporate Product Management Plans.

2. Socket Evaluation using LCM

Complete technical investigation and make socket use decision based on financial and technical data.

3. Discount Rate used in LCM

Risk Adjustment Alternative

Product specific risk should be reflected in the cost and revenue data for the product (best case, worst case, most likely)

Non-product specific risk should be applied through the discount rate but should be determined for some number of product or market sub-sets (terminals, large systems, mid-range)

Non risk adjusted discount rate. Use sensitivity analysis to evaluate risk.

Accounting Adjustment for Assets

Should be removed from discount rate and replaced by an algorithm that would allocate an appropriate charge for plant and equipment not otherwise accounted for.

4. Manufacturing Learning Curve

Purchasing to obtain the lowest possible cost of material (both in-house and outside DEC). (Machine 72% material)

Have a Dock Mergeable System. Potential savings (over life of the product) between Package System and Dock Merge is $40 million. The potential savings (over life of the product) between Complex System and Package System is $70 million.

Pursue the creation of a life cycle inventory model to be included within the life cycle cost model.

5. Customer Service Learning Curve

Continue VAX 11/780 strategy of initial installation and service of VENUS with experienced people and evolving over time to use less experienced people and more mature tools. Consistent with customer services strategy.

Accelerate development of tools (diagnostic, SPEAR, etc.); Maintain high quality level of spares to take advantage of learning curve.

6. Market Targeting

Recommendations:

Exploit Momentum - new product/existing markets
- channel efficiencies
- market share if selling cost benefit
- 3 of top 5 are scientific computation

Opportunities - new product/new markets
- product line's add application value
- products necessary, but not sufficient
- market growth makes it easier

7. Configuration Variation

Generate data for sensitivity testing.
Develop methodology for accommodating change.

8. Customer Satisfaction

Retain strengths: Thruput; interactive software; ease of programming.

Close IBM gap: Compatibility; software support; sales

9. Business Planning

Provide Product Managers with a decision support tool with the following characteristics:

- uses LCM
- conceptually different from P/L statement
- eliminate cumbersome traditions:
    - FA & T (% of MLP)
    - I & W (% of MLP)
    - cost-based pricing
    - mark-up goals
- Allows experimentation with revenue stream/cost structure alterations

SUMMARY

1. The use of a Business Planning tool that includes LCM requires changes to the Product Management and Product Development team responsibilities and tasks.

2. An appropriate discount rate must be used in the LCM to obtain meaningful results.

3. Managing programs to a goal of maximizing life cycle profitability requires an environment that is receptive to proposals that alter the revenue streams and cost structures within the corporation. For example:

    a. Allocate funds to engineering to reduce FA&T and service costs.

    b. Allocate funds to engineering to increase sales revenues and decrease service cost.

    c. Allocate funds to engineering to increase transfer cost (reducing sales profitability) and decrease service cost (increasing service profitability).

4. Customer satisfaction parameters include some that are product determined (ease of programming) and some that are organization dependent (software support). DEC's current customers buy our products because of our strengths. To attract IBM customers requires developing new strengths without abandoning current strengths.

VENUS TASK FORCE

FINAL REPORT

JUNE 26, 1980

VENUS TASK FORCE

SPONSORS: Andy Knowles and Win Hindle

DATE: July 10, 1979

MEMBERS: Ulf Fagerquist
Walt Manter
Dave Thorpe

TASK: Study the total life cost of Engineering, Manufacturing, Installation, Warranty, and Service for VENUS systems. Develop alternatives for spending to minimize total costs and maximize customer satisfaction for this product, given IBM as chief competition.

June 26, 1980

VENUS TASK FORCE

- NINE SUBTASKS

  1. LCM DEFINITION AND REQUIREMENTS : ROLF MCCLELLAN
  2. SOCKET EVALUATION USING LCM : REG BURGESS/VIC KU
  3. DISCOUNT RATE USED IN LCM : DAVE WETHERBEE
  4. MANUFACTURING LEARNING CURVE : JOHN GROSE
  5. CUSTOMER SERVICES LEARNING CURVE : REG BURGESS
  6. MARKET TARGETTING : CARL GIBSON
  7. CONFIGURATION VARIATION: CARL GIBSON
  8. CUSTOMER SATISFACTION : CARL GIBSON
  9. BUSINESS PLANNING : CARL GIBSON

- USE VENUS AS VEHICLE
- LINK TO VAX 11/780 TASK FORCE
- LINK TO COST-OF-OWNERSHIP TASK FORCE

JUNE 26, 1980

VENUS TASK FORCE

PARTICIPANTS:

CUSTOMER SERVICES

REG BURGESS
ROB HILBRINK
BOB LEVASSEUR
ROLF MCCLELLAN
MIKE ROBEY
BOB TABOR

MANUFACTURING

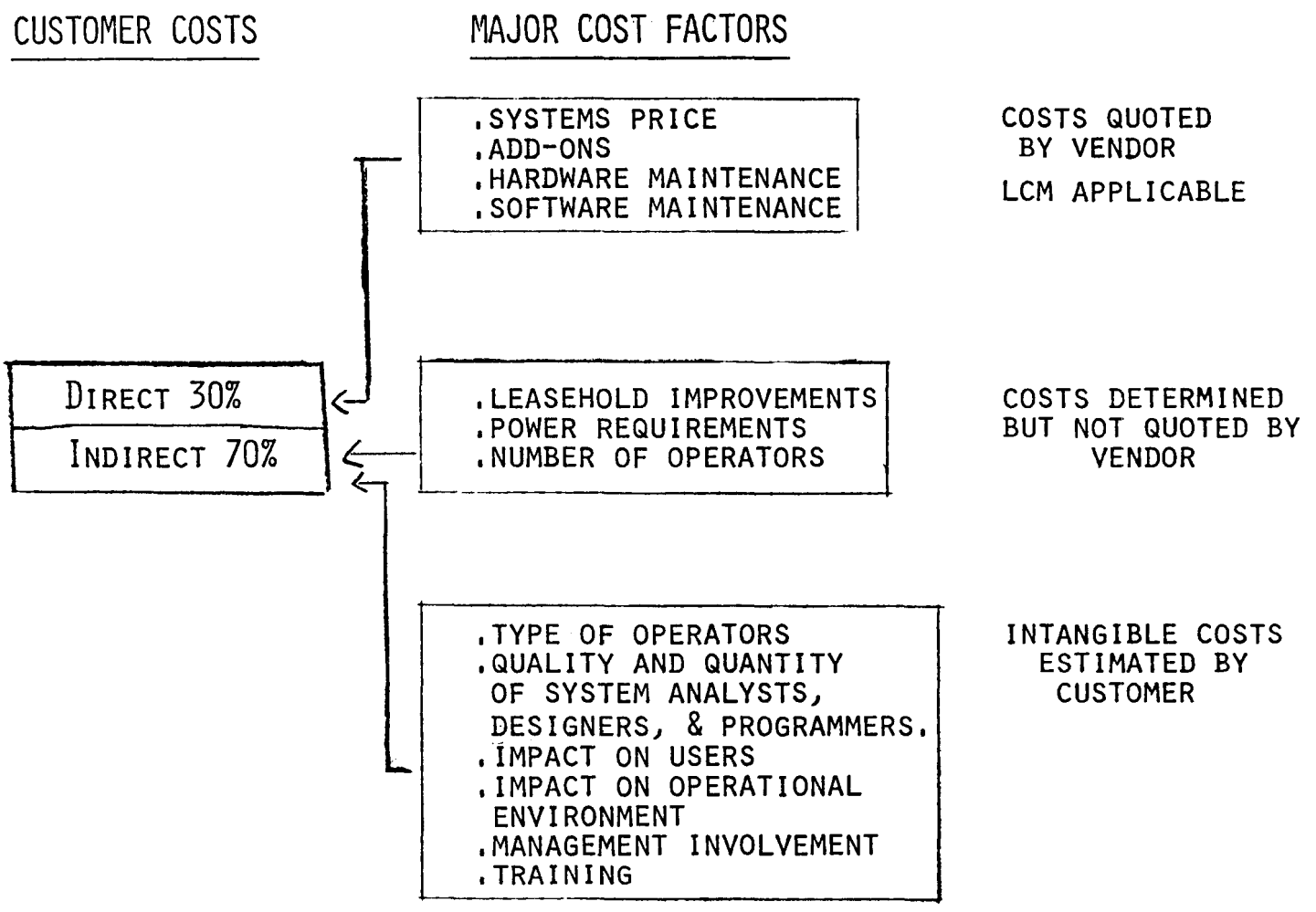
DAVE BEVERIDGE
JOHN GROSE
BILL MARTEL

ENGINEERING

BRUCE CAMPELIA
SAS DURVASALA
CARL GIBSON
PER HJERPPE
GEORGE HOFF
LEN KREIDERMACHER
VIC KU
DAVE WETHERBEE

JUNE 26, 1980

VENUS TASK FORCE



VENUS TASK FORCE

- AGENDA

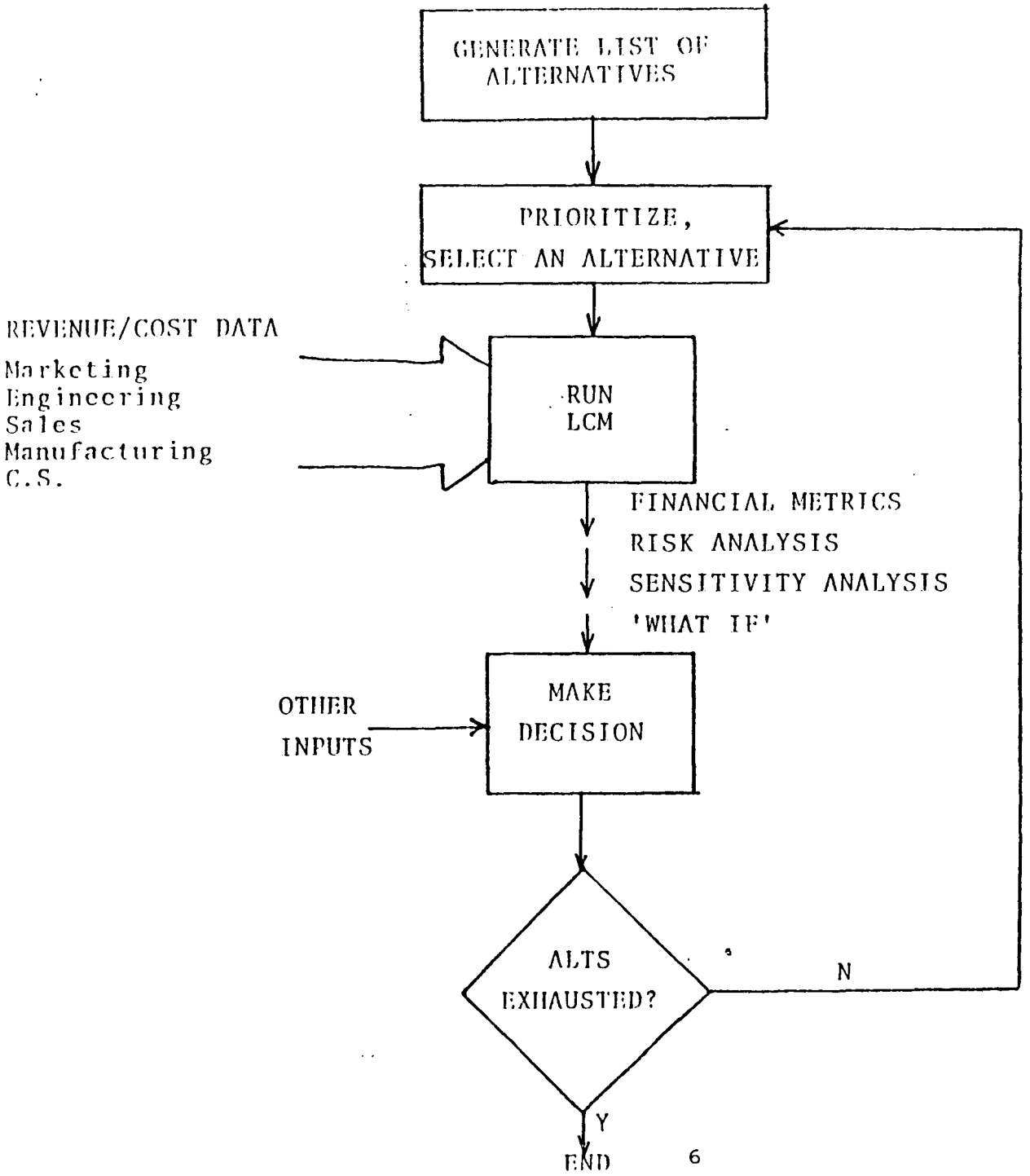
1. NINE SUBTASKS

   RECOMMENDATIONS

   BACK UP IN VOLUME 2

2. SUMMARY

USING LCM


GENERATE LIST OF ALTERNATIVES
PRIORITIZE, SELECT AN ALTERNATIVE
REVENUE/COST DATA
Marketing
Engineering
Sales
Manufacturing
C.S.
RUN LCM
FINANCIAL METRICS
RISK ANALYSIS
SENSITIVITY ANALYSIS
'WHAT IF'
OTHER INPUTS
MAKE DECISION
ALTS EXHAUSTED?
N
Y
END

VENUS TASK FORCE

1. LCM DEFINITION AND REQUIREMENTS

- STATUS

  MANUAL VERSION EXISTS AND USED BY VENUS AND 11/780 TASK FORCES.

- RECOMMENDATIONS

  AUTOMATE LCM REQUIREMENTS.
  BUSINESS PLANNING TOOL WOULD HAVE BOTH "TRADEOFF" AND "BUSINESS PLAN" (BURP) MODES.

  REQUIREMENTS:

  1) AUTOMATED ASSIMILATION OF DATA FROM CURRENT AND ADDITIONAL FUNCTIONAL PLANNING MODELS (E.G. LCBM, SELLING COST MODEL)

  2) AUTOMATED SENSITIVITY ANALYSIS SHOWING:

     A) CHANGES IN STRATEGIC VARIABLES WITH OFFSETTING EFFECTS ON NPV (IF ΔTIME TO MARKET = 2Q'S; WHAT Δ TRANSFER COST OFFSETS THIS?)

     B) RISK ASSESSMENT OF DESIGN ALTERNATIVE (ΔFAILURE RATE MAKES ΔNPV EQUALS 0)

  RECOMMENDATIONS ARE CONSISTENT WITH CORPORATE PRODUCT MANAGEMENT PLANS.

VENUS TASK FORCE

2. SOCKET EVALUATION USING LCM

- BEFORE EVALUATION
  - MCA SOCKETS BEING INVESTIGATED
- EVALUATION (10 YEAR LIFE CYCLE BEGINNING FRS; PRODUCT VOLUME 12,500 SYSTEMS)
  - POTENTIAL $4.5M SAVINGS (MFG AND SERVICE) BY USING MCA SOCKETS
  - POTENTIAL $10.8M SAVINGS (MFG AND SERVICE) BY USING RAM SOCKETS
- SENSITIVITY ANALYSIS: ADDITIONAL INTERMITTENT FAILURE RATES OF SOCKETS SO Δ NPV EQUALS 0
  - MCA SOCKETS: 2.35 TIMES ORIGINAL FAILURE RATE
  - RAM SOCKETS: 23.13 TIMES ORIGINAL FAILURE RATE
- AFTER EVALUATION

  USE OF LCM CONVINCED ENGINEERING THAT MCA AND RAM SOCKETS HAVE LIFE CYCLE COST BENEFIT
  - ENGINEERING AGRESSIVELY SEEKING SOLUTIONS TO TECHNICAL PROBLEMS WITH SOCKETS FOR MCA AND RAM
  - CONTROL RISK BY USING SAME PC FOOT-PRINT FOR SOCKET AND NON-SOCKET.
- RECOMMENDATION
  - COMPLETE TECHNICAL INVESTIGATION AND MAKE SOCKET USE DECISION BASED ON FINANCIAL AND TECHNICAL DATA

- NOTE
  - USE OF LCM UNCOVERED THE FINANCIAL BENEFIT OF RAM SOCKETS

JUNE 26, 1980

VENUS TASK FORCE

3. DISCOUNT RATE USED IN LCM

- USED IN DCF ANALYSIS TO DETERMINE WHETHER OR NOT AN INVESTMENT MEETS MINIMUM RETURN REQUIREMENTS

- IT DESCRIBES THE COMPOUND RATE OF RETURN REQUIRED TO SATISFY THE INVESTOR'S EXPECTATION FOR AN ASSUMED LEVEL OF RISK

- CURRENT DEFINITION -

- ROA GOAL	16%
- ADJ. FOR ASSETS	10%
- RISK	14%
	40%

- CONCERNS

ERROR COMPOUNDED

UNIVERSALLY APPLIED

JUNE 26, 1980

3. DISCOUNT RATE USED IN LCM

VENUS CPU LIFE CYCLE COSTS

	RAW COST		NPV @ 20%		NPV @ 40%	
ENG	$25M	5.0%	$17M	12%	$13M	17%
MFG	$320M	70.0%	$101M	70%	$58M	72%
F/S	$115M	25.0%	$26M	18%	$9M	11%
TOTAL	$460M	100%	$144M	100%	$80M	100%

JUNE 26, 1980

3. DISCOUNT RATE USED IN LCM

- RECOMMENDATIONS

RISK ADJUSTMENT ALTERNATIVES

- PRODUCT SPECIFIC RISK SHOULD BE REFLECTED IN THE COST AND REVENUE DATA FOR THE PRODUCT (BEST CASE, WORST CASE, MOST LIKELY)

- NON-PRODUCT SPECIFIC RISK SHOULD BE APPLIED THROUGH THE DISCOUNT RATE BUT SHOULD BE DETERMINED FOR SOME NUMBER OF PRODUCT OR MARKET SUB-SETS (TERMINALS, LARGE SYSTEMS, MID-RANGE)

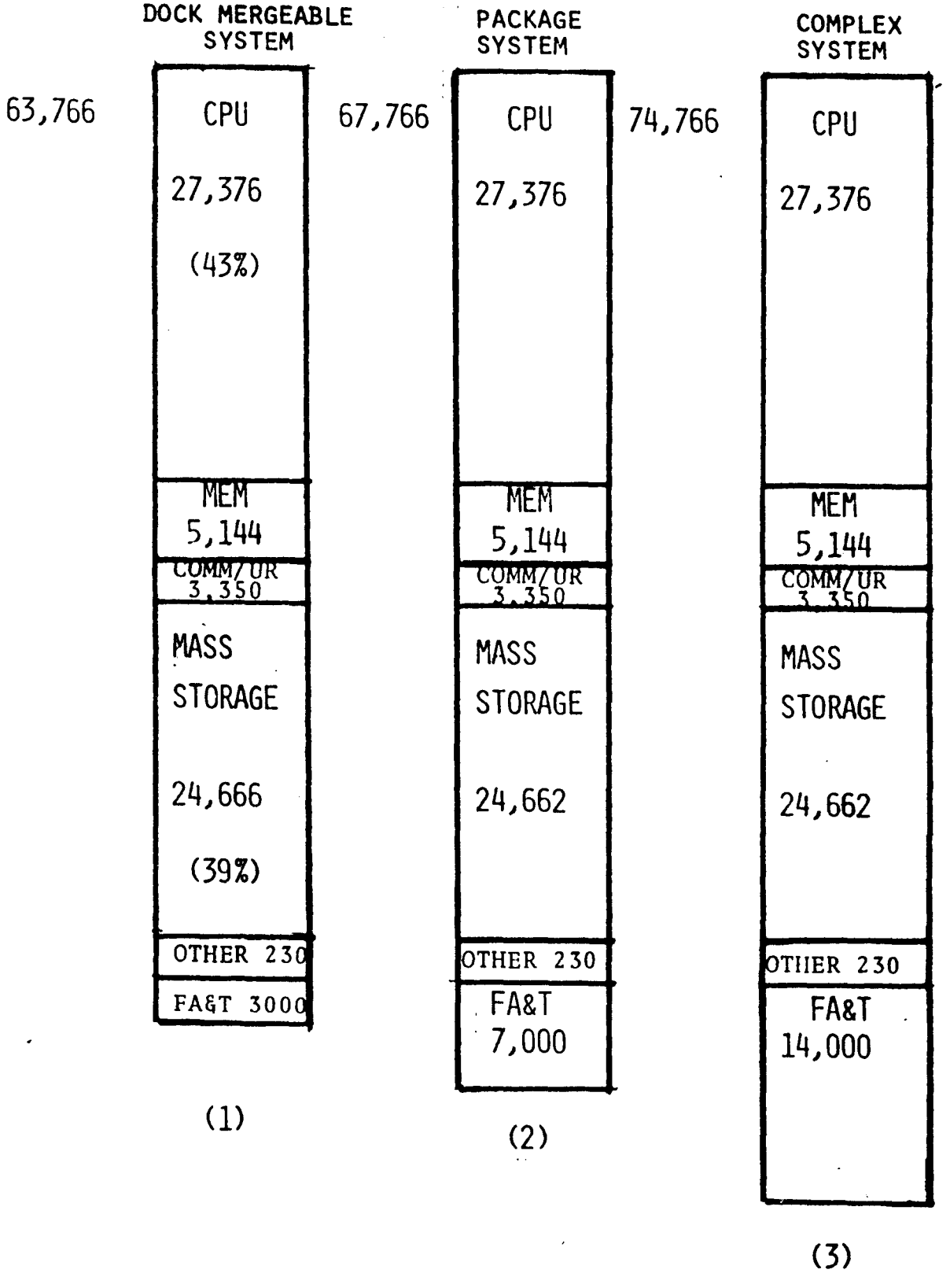
- NON RISK ADJUSTED DISCOUNT RATE. USE SENSITIVITY ANALYSIS TO EVALUATE RISK.

ACCOUNTING ADJUSTMENT FOR ASSETS

- SHOULD BE REMOVED FROM DISCOUNT RATE AND REPLACED BY AN ALGORITHM THAT WOULD ALLOCATE AN APPROPRIATE CHARGE FOR PLANT AND EQUIPMENT NOT OTHERWISE ACCOUNTED FOR

JUNE 26, 1980

BASIC SYSTEM COST
STEADY STATE



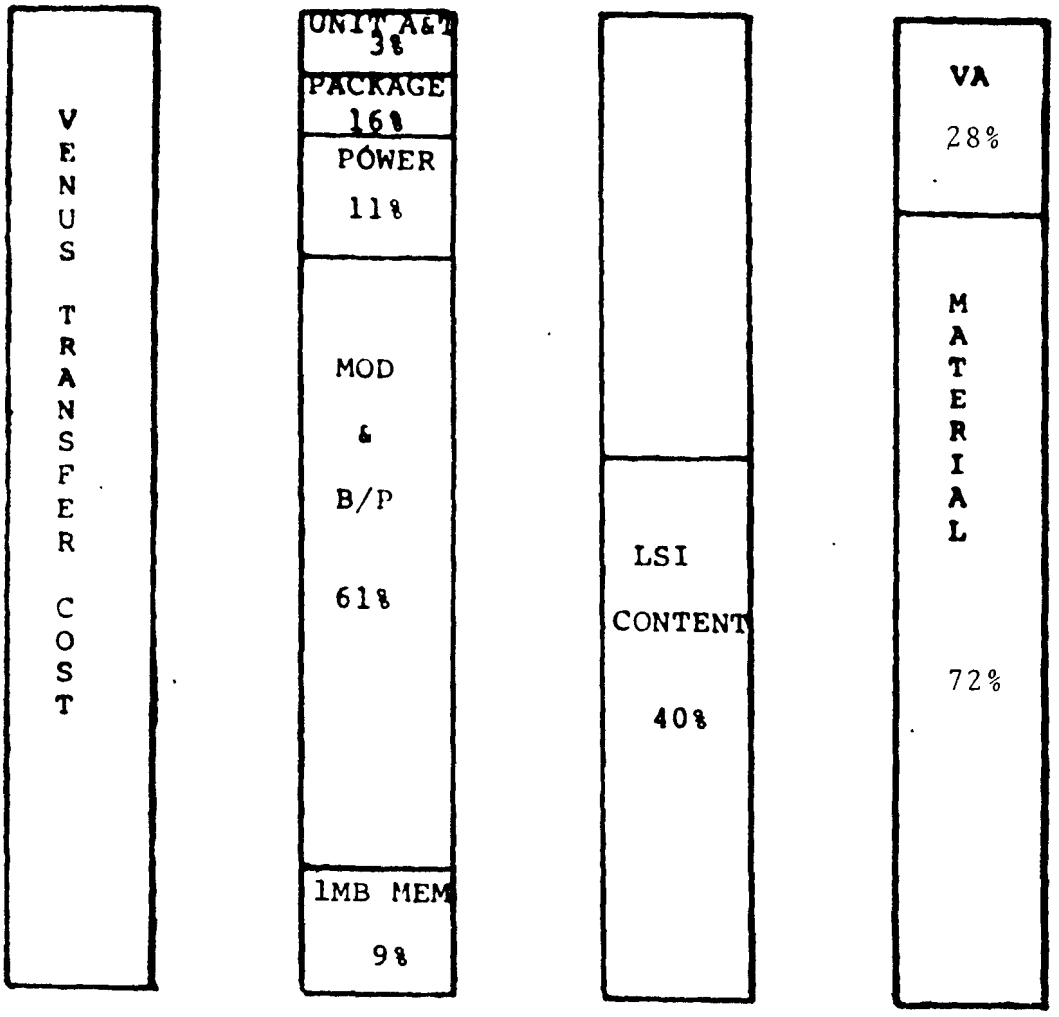
4. MANUFACTURING LEARNING CURVE

STEADY STATE

VENUS CONCEPTUAL COST ESTIMATES

COST STRATIFICATION

51148*



*INCLUDES CPU, P.S. CAB, MEM (1MB), CONSOLE, AND SBIA

6-20-80
John Grose

VENUS TASK FORCE

4. Manufacturing Learning Curve

Recommendations:

1. Purchasing to obtain the lowest possible cost of material (both in-house and outside DEC). (Machine 72% material)

2. Have a Dock Mergeable System.

   Potential savings (over life of the product) -- 10,000 systems between Package System and Dock Merge

   $40 million

   between Complex System and Package System

   $70 million

3. Pursue the creation of a life cycle inventory model to be included within the life cycle cost model.

June 24, 1980

John Grose

VENUS TASK FORCE

5. Customer Services Learning Curve

- VAX 11/780 experience
  - Installation: 35 - 50% reduction during first year
  - Service for

	System	CPU	RP06	
MTTR:	19%	12%	26%	reduction over 6 mo.
MLH:	13%	9%	29%	reduction over 6 mo.

- Recommendation
  - Continue VAX 11/780 strategy of initial installation and service of VENUS with experienced people and evolving over time to use less experienced people and more mature tools. Consistent with customer services strategy.
  - Accelerate development of tools (diagnostics, SPEAR, etc.); maintain high quality level of spares to take advantage of learning curve.

June 26, 1980

VENUS TASK FORCE

6. Market Targetting

- ROI / Market Share Correlations
  - High correlation for infrequently purchased products
  - Only dominant vendors can profitably obtain premium prices
  - Research and Development yields more profit if market share is high
  - Market share most often related to profit in companies over $1.5B annual sales
  - New products help dominant vendor least
- WHY
  - Influence pricing policy structure
  - Strong bargaining posture

June 26, 1980

6. MARKET TARGETTING

CONCLUSIONS

MOST LIKELY TO BE ABLE TO BEAT IBM IN

1)	FED GOVERNMENT	9.5% OF DEC
2)	EDUCATION	5.8% OF DEC
		15.3%

AND, MAYBE

3)	TRANS/UTIL	10.5% OF DEC
		25.8%

BUT,

- o FED GOVT REQUIRES 355% GROWTH IN A MARKET GROWING 6% / YR.
- o EDUCATION REQUIRES 702% GROWTH IN A MARKET GROWING AT 10% / YR.
- o TRANS/UTIL REQUIRES 740% GROWTH IN A MARKET GROWING AT 13% / YR.

JUNE 26, 1980

6. MARKET TARGETTING

CHOOSING CHANNELS

FOR

PROFIT

(IN THE SHORT TERM)

IF A RESOURCE IS SCARCE,

APPLY IT TO THE CHANNELS

THAT ARE MOST EFFICIENT

PROFIT GENERATORS

6. MARKET TARGETTING

1ST YEAR VOLUME CAPACITY IS SCARCEST RESOURCE

WHICH CHANNEL USES IT MOST EFFICIENTLY?

PRODUCT GROUP	PBT / TC *
TIG	.80
TOEM	.71
ESG	.68
MSG	.63
LDP	.61
CSI	.59
COEM	.42
GSG	.42
ECS	.34
MDC	.23
G/A	.05

*PER Q1, Q2 FY80

JUNE 26, 1980

6. MARKET TARGETTING

RECOMMENDATIONS:

- EXPLOIT MOMENTUM - NEW PRODUCT/EXISTING MARKETS
  - CHANNEL EFFICIENCIES
  - MARKET SHARE IF SELLING COST BENEFIT
  - 3 OF TOP 5 ARE SCIENTIFIC COMPUTATION

- OPPORTUNITIES - NEW PRODUCT/NEW MARKETS
  - PRODUCT LINE'S ADD APPLICATION VALUE
  - PRODUCTS NECESSARY, BUT NOT SUFFICIENT
  - MARKET GROWTH MAKES IT EASIER

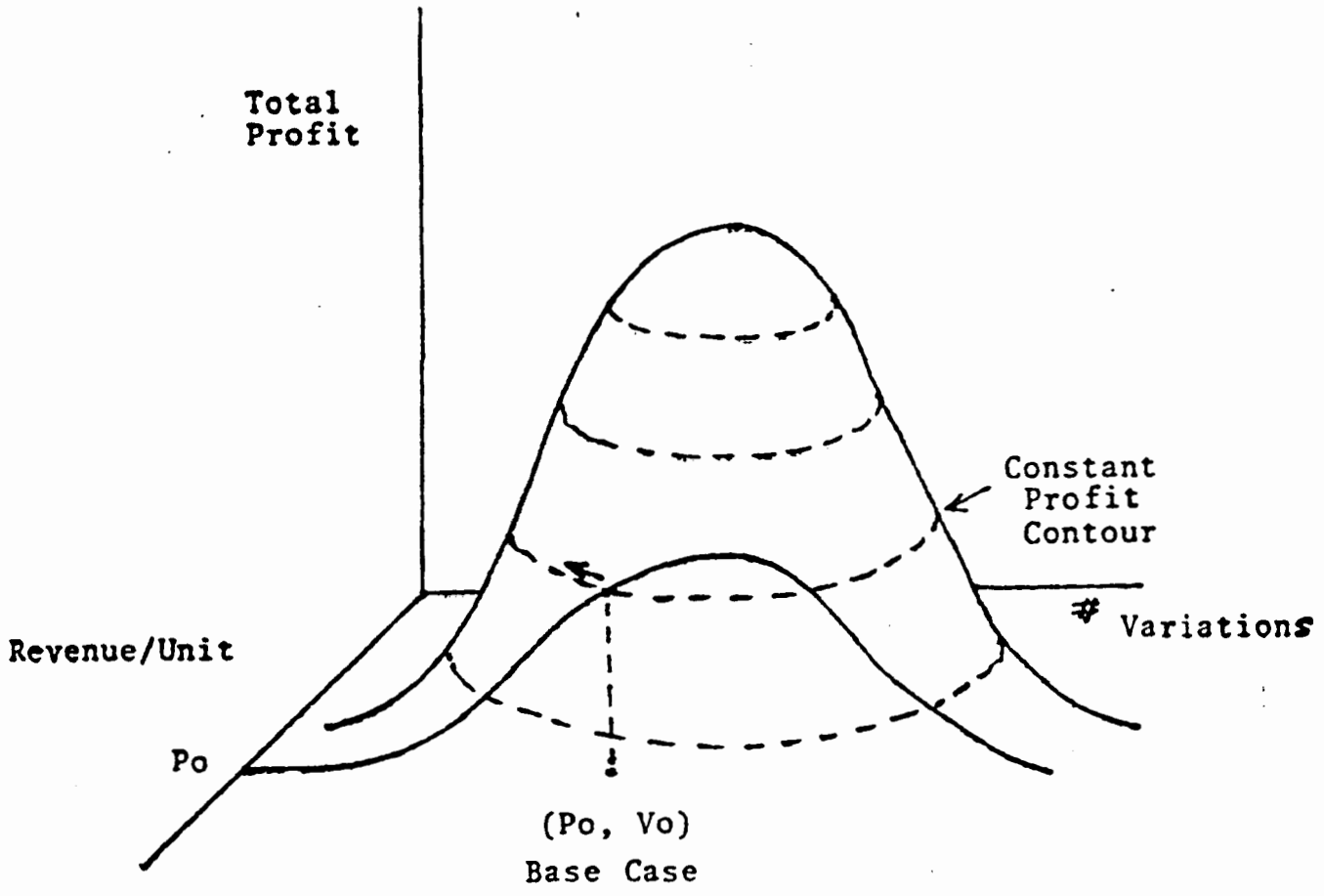
JUNE 26, 1980

7. Configuration Variations

- Status
  - SUPPORTED/NOT SUPPORTED PERIPHERALS LIST COMPLETED
  - PROBABLE 1ST YEAR PACKAGE SYSTEM LIST COMPLETED
  - CONCEPTUAL FRAMEWORK FOR ANALYSIS WORKED OUT
- Recommendations
  - GENERATE DATA FOR SENSITIVITY TESTING
  - DEVELOP METHODOLOGY FOR ACCOMMODATING CHANGE

June 26, 1980

7. CONFIGURATION VARIATION



JUNE 26, 1980

VENUS TASK FORCE

8. CUSTOMER SATISFACTION

SALES DEPT. SURVEYS

SOFTWARE SERVICES SURVEY

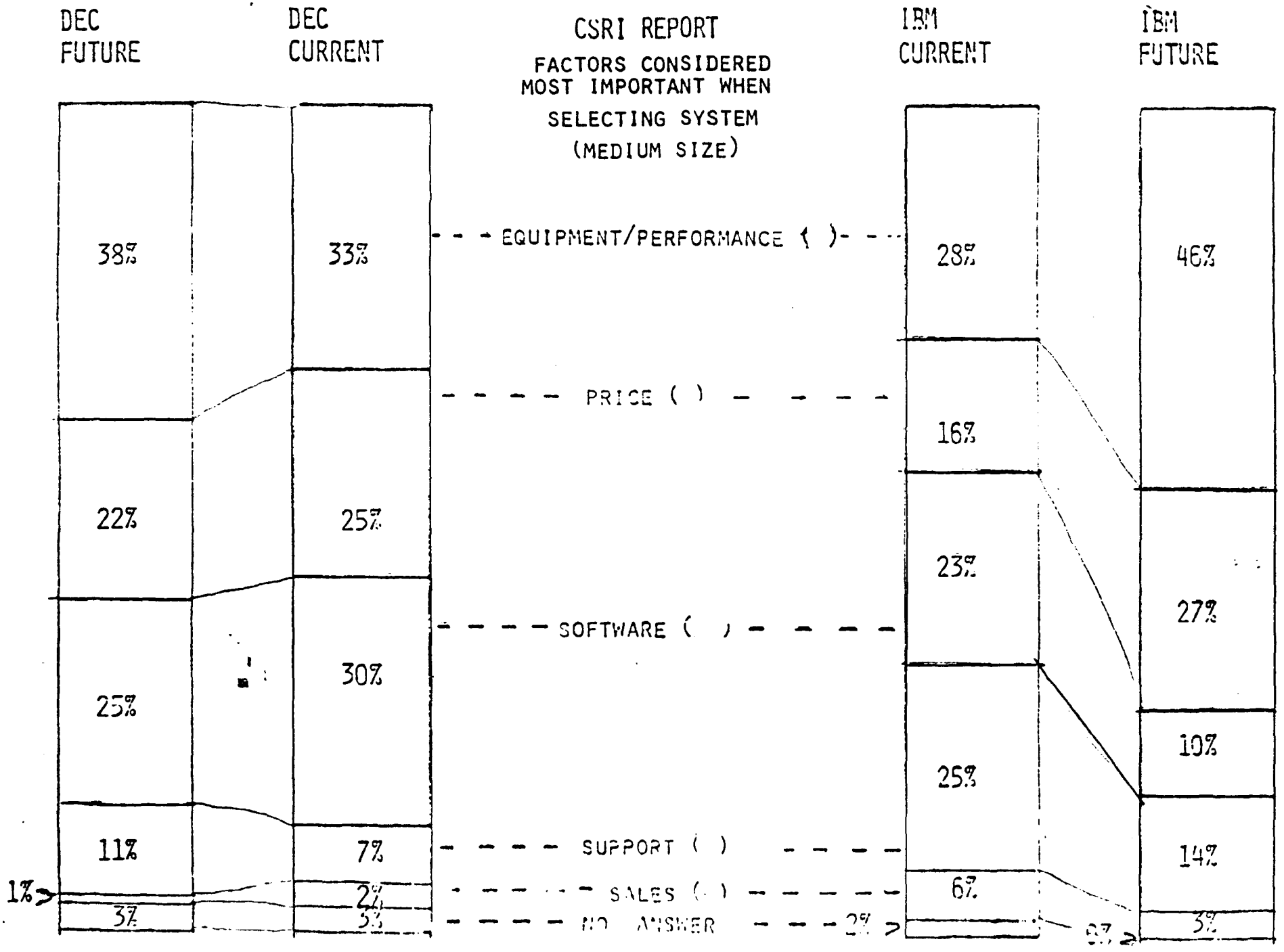
FIELD SERVICE SURVEYS

TRADE PRESS

CSRI

- 35 SYSTEMS FOR EACH VENDOR FOR EACH SIZE GROUPING
- 28 MINUTE TELEPHONE SURVEY

JUNE 26, 1980


DEC FUTURE
DEC CURRENT
CSRI REPORT
FACTORS CONSIDERED MOST IMPORTANT WHEN SELECTING SYSTEM (MEDIUM SIZE)
IBM CURRENT
IBM FUTURE
38%
33%
EQUIPMENT/PERFORMANCE ( )
28%
46%
PRICE ( )
16%
22%
25%
SOFTWARE ( )
23%
27%
25%
30%
10%
11%
7%
SUPPORT ( )
25%
14%
1%
2%
SALES ( )
6%
3%
3%
NO ANSWER
2%
3%

VENUS TASK FORCE

CUSTOMER SATISFACTION RESEARCH INSTITUTE

	MEDIUM				LARGE	
			DEC	IBM	DEC	IBM
	20's	VAX	(20's + VAX)	(370's)	(10's)	(303X)
THRUPUT	.53	.73	.64	.57	.68	.78
RELIABILITY/ UPTIME	.43	.74	.69	.73	.58	.81
PRICE	.8	.67	.80 S	.48	.70	.74
INTERACTIVE S/W	.75	.66	.69 S	.30	.89 S	.34
EASE OF PROG	.7	.74	.74 S	.28	.74 S	.47
COMP/CONV	.0	.14	-.13 W	.54	.17 W	.50
OPER SYSTEM	.73	.69	.70	.64	.81	.74
LANGUAGES	.40	.47	.44	.51	.54	.63
HDW.MAINTENANCE	.68	.47	.54	.67	.59	.89
SOFTWARE SUPP.			.26 W	.52	.25 W	.51
SALES	.23	.05	.16 W	.44	.14 W	.65
OVERALL			.54	.56	.58	.73

S: STRONG COMPETITIVE POSITION

W: WEAK COMPETITIVE POSITION

1.0	VERY SATISFIED
.5	SOMEWHAT SATISFIED
0	NEITHER SATISFIED NOR DISSATISFIED
-.5	SOMEWHAT DISSATISFIED
-1.0	VERY DISSATISFIED

JUNE 26, 1980

8. CUSTOMER SATISFACTION

- INDICATORS OF
  - STRENGTHS TO PROTECT
  - LEVERAGE AREAS
  - FOREIGN BASE'S STRENGTHS AND WEAKNESSES
- RECOMMENDATIONS
  - RETAIN STRENGTHS: THRUPUT; INTERACTIVE SOFTWARE; EASE OF PROGRAMMING
  - CLOSE IBM GAP: COMPATIBILITY; SOFTWARE SUPPORT; SALES

9. BUSINESS PLANNING

OBJECTIVES:

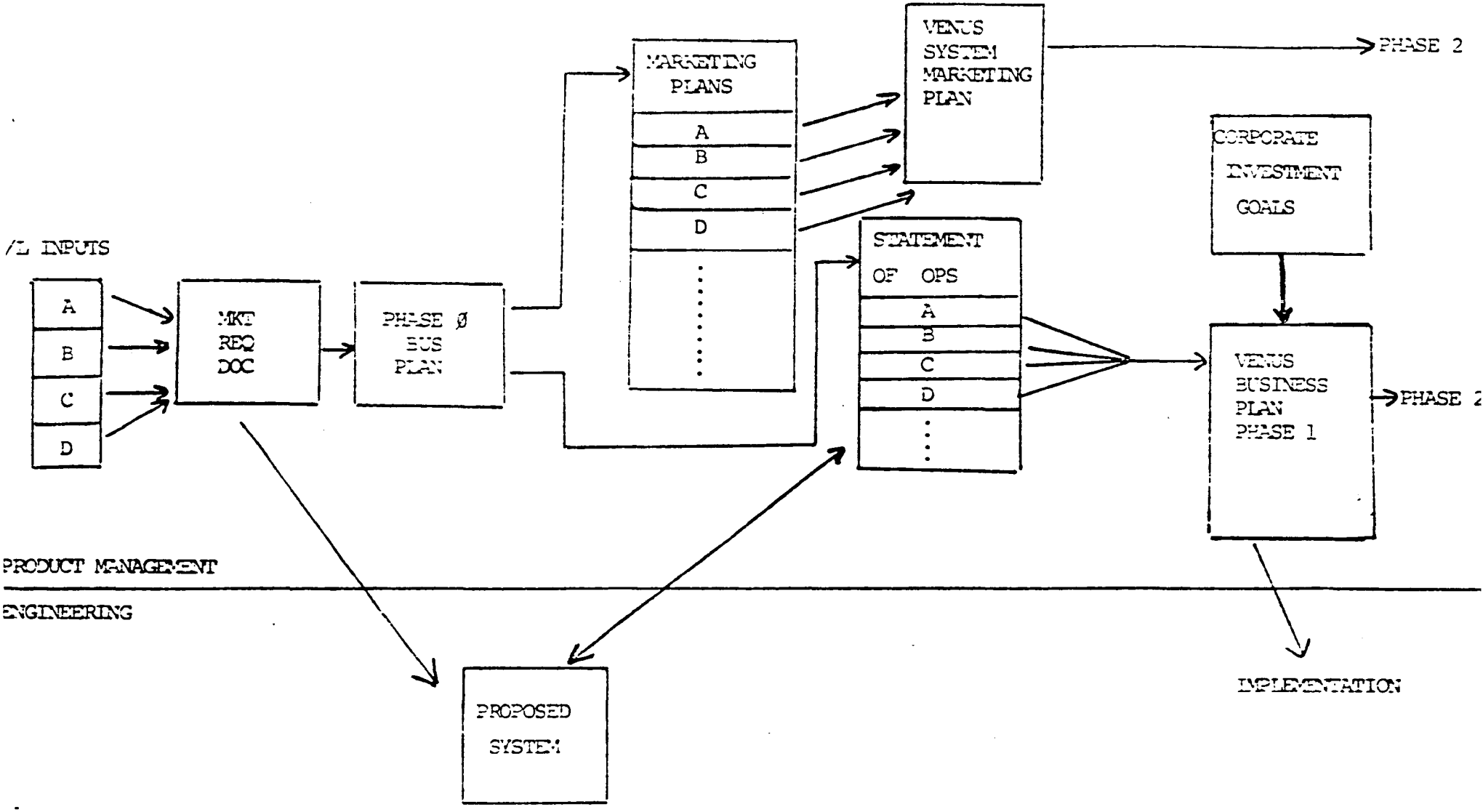
- JUSTIFY USE OF RESOURCES
- DECISION SUPPORT TOOL

VTF & BUSINESS PLAN:

- LIFE CYCLE COMPONENTS
- BETTER QUANTIFICATION
- BETTER TOOLS
- BROAD EXPERIENCE CONSOLIDATION

SYSTEMS MARKETING MANAGEMENT

PROCESS (PHASE Ø & 1)



9. BUSINESS PLANNING

SOME SENSITIVITIES RELATIVE TO PHASE 0 BASE CASE
Δ'S TO NPV OF OVERALL PROGRAM CASH

TIME TO MARKET	$9M/QUARTER
TRANSFER COST vs SCHEDULE	$5-6K/QUARTER

FA&T ALTERNATIVES (KERNALS)	(NPV)
100% COMPLEX	$22.5M FA&T COST
100% PACKAGE SYSTEMS	11.25M
100% DOCK MERGE	4.8M

WITH: DOCK MERGE	PACKAGE SYSTEM	COMPLEX	NPV OF COST
1%	50%	49%	$16.7M
10%	50%	40%	$15.1M (-9.5%)
40%	50%	10%	11.2 (-32%)

JUNE 26, 1980

VENUS TASK FORCE

9. Business Planning

- RECOMMENDATIONS

  PROVIDE PRODUCT MANAGERS WITH A DECISION SUPPORT TOOL WITH FOLLOWING CHARACTERISTICS

  - USES LCM
  - MAY BE CONCEPTUALLY DIFFERENT FROM P/L STATEMENT
  - ELIMINATE CUMBERSOME TRADITIONS
    - FA&T (% OF MLP)
    - I & W (% OF MLP)
    - COST-BASED PRICING
    - MARK-UP GOALS
  - ALLOWS EXPERIMENTATION WITH REVENUE STREAM/COST STRUCTURE ALTERATIONS

VENUS TASK FORCE

SUMMARY

1. Use of Business Planning Tool with LCM
   - Product manager job changes
   - Product development team job changes

2. Discount Rate
   - LCM usefulness dependent on appropriate discount rate

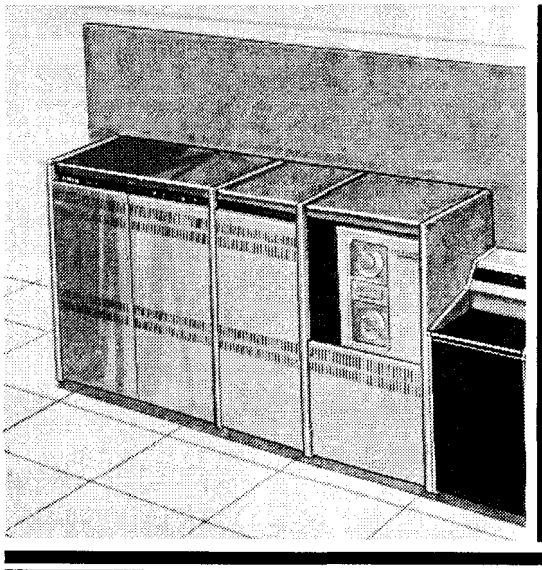
3. Maximizing life cycle profitability requires environment receptive to proposals altering revenue streams and cost structure
   - Increase engineering costs to reduce FA & T and service costs (testing).
   - Increase engineering costs to increase sales revenue and reduce service costs (sockets).
   - Increase engineering costs to increase transfer cost and reduce service costs (ramp).

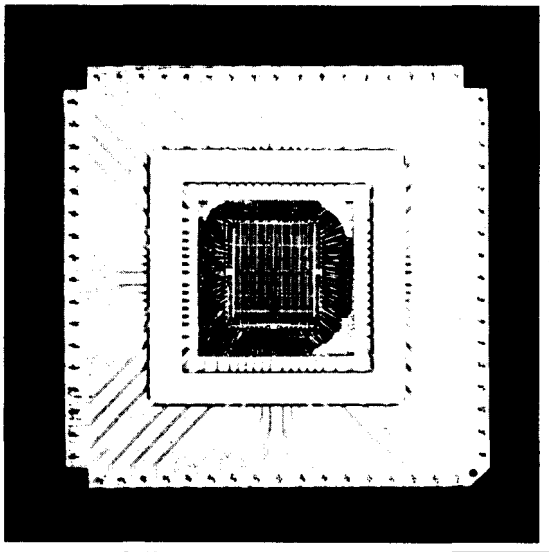
4. Customer Satisfaction Parameters
   - DEC customers like DEC strengths and IBM customers like IBM strengths.
   - To attract IBM customers, retain strengths and develop new strengths.

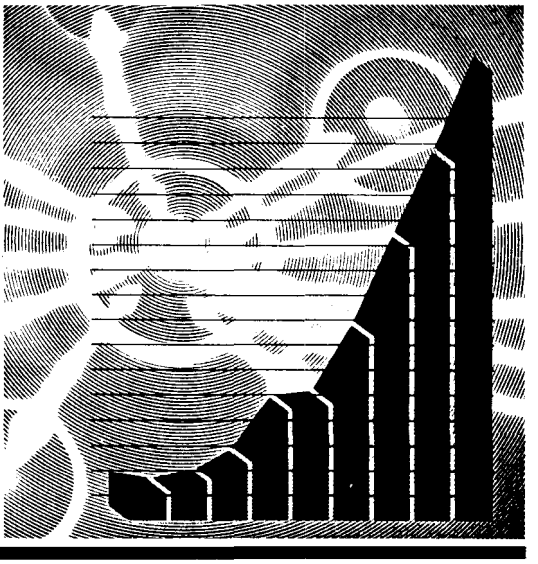
June 26, 1980

VAX-11/VENUS

VENUS TASK FORCE
FINAL REPORT
Vol. 2 of 2
Appendices
June 26, 1980







COMPANY CONFIDENTIAL

digital

VENUS TASK FORCE

FINAL REPORT

TABLE OF CONTENTS

VOLUME 2

Appendix 1a

Rolf McClellan
Management Science Group
20 JUN 80

LCM Definition and Requirements

Definition

The Venus Task Force has interpreted its charter to include the task of defining the requirements for an integrated financial tool (LCM) to aid in the process of optimally distributing product investment and cost across functions.

Figure LCM1 shows the positions of the product "optimization" process and LCM in a pyramid of product development interdependencies. Customer satisfaction, market share and ROI are dependent on the success of the optimization process. In turn, this process is reliant on the corporate and functional strategies, as well as the product planning and selection activities. LCM and other related discounted cash flow tools such as BURP and LCBM are particularly dependent on product-based financial strategy (for appropriate discount rates) and product-based accounting and cost estimation systems. COO is a cash flow model that analyzes a customer's lifetime costs of ownership.

Using LCM

Figure LCM2 shows a flow chart of how LCM would be used by product and program management.

(1) a list of product alternatives with cross functional impact would be generated by the development team and other interested parties.

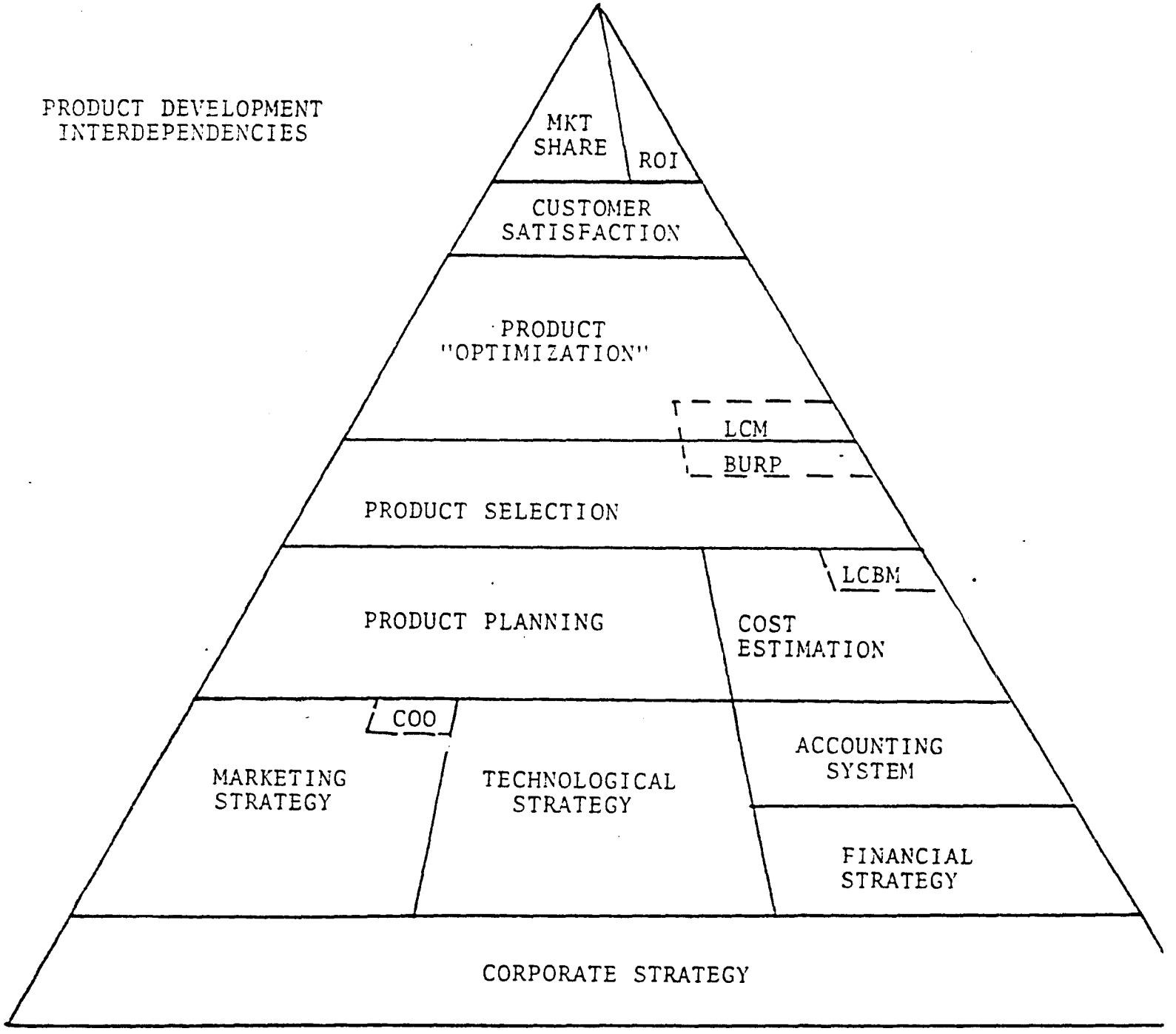
(2) product management would sort and prioritize the list. The top priority alternative would be selected.

(3) data specific to the alternative would be gathered from the functional groups. LCM would be run to evaluate the alternative relative to the base case. The model's output would include financial metrics (NPV, IRR, etc.), risk analysis, and sensitivity analysis.

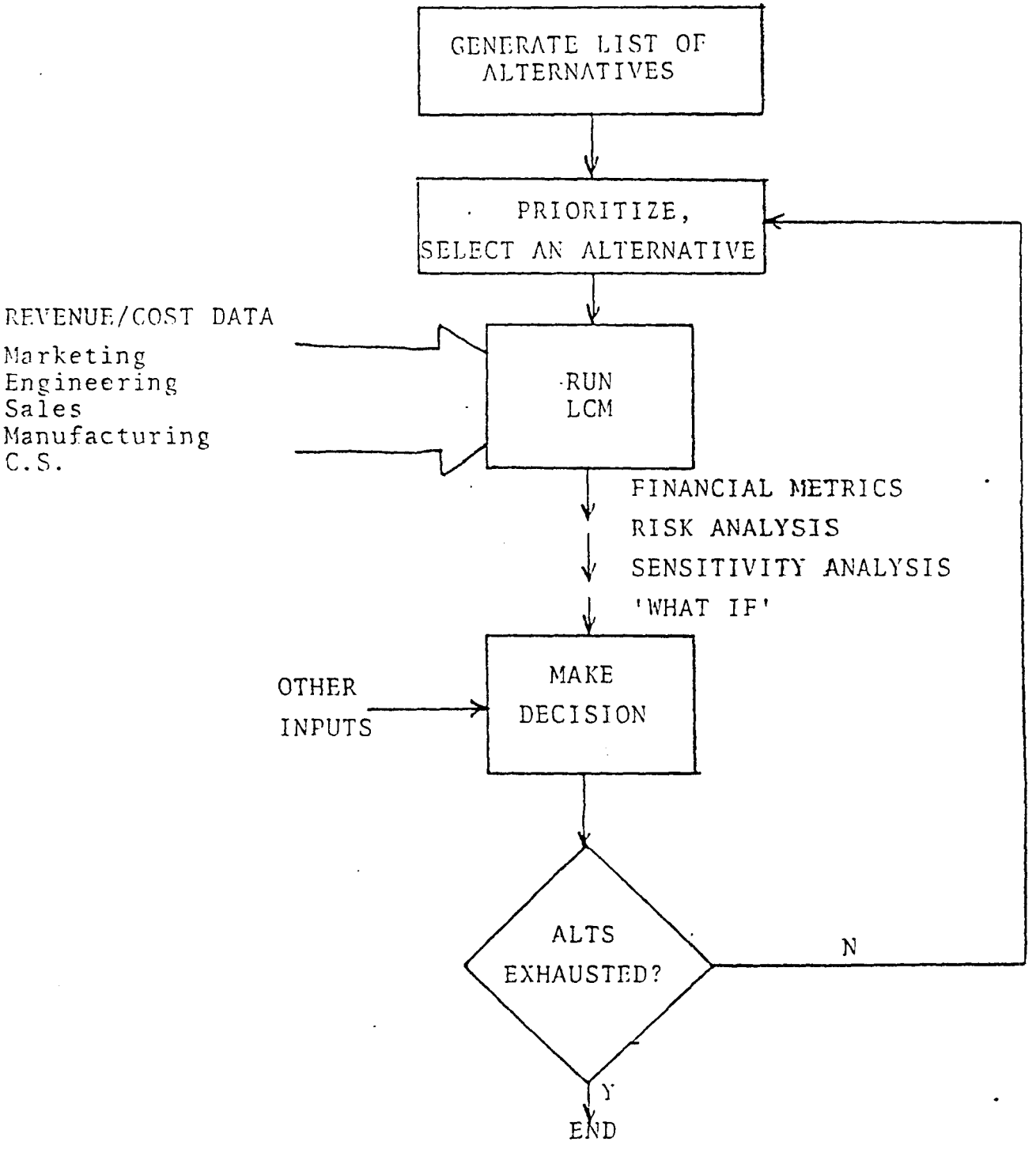
(4) LCM's outputs together with other non-financial strategic and technological considerations would support the decision to accept or reject the alternative.

(5) After the decision is reached, the base case data file would be modified and remaining alternatives re-prioritized as appropriate.

PRODUCT DEVELOPMENT
INTERDEPENDENCIES



USING LCM


GENERATE LIST OF ALTERNATIVES
PRIORITIZE, SELECT AN ALTERNATIVE
REVENUE/COST DATA
Marketing
Engineering
Sales
Manufacturing
C.S.
RUN LCM
FINANCIAL METRICS
RISK ANALYSIS
SENSITIVITY ANALYSIS
'WHAT IF'
OTHER INPUTS
MAKE DECISION
ALTS EXHAUSTED?
N
Y
END

Key Messages re Use of LCM

(1) for major projects, many alternatives should be proposed, considered, and analyzed.

(2) multiple alternatives make data generation, data transfer, and sensitivity analysis a major burden for financial analysts.

Requirements for LCM

In order to alleviate these burdens and to ensure consistent tradeoff analysis criteria, the following model features are required.

(1) Automatic transfer of data from current and additional functional planning models, such as Field Services' LCBM and a possible selling cost model. This would require that each such support model would include an output file compatible with LCM.

(2) Automated sensitivity analysis with detailed inputs and internal LCM logic to support:

(a) strategic tradeoffs - changes in strategic variables with offsetting effects on the projects net present value. For example, what if time to market slips by two quarters, what reduction in transfer cost would offset this?

To support this sort of tradeoff analysis, marketing would have to provide estimates of the elasticity of demand to strategic variables (TTM, MLP, BYC, etc.)

(b) prioritizing specific alternatives - changes in detailed cost parameters (such as direct materials and MTBF) that have offsetting effects on NPV.

(c) risk assessment - allowable margins of error in key estimates. Errors in excess of these margins would drive the NPV of the alternative to zero. For example, in the Venus socket study, the increased intermittent failure rate that would cancel the benefits (NPV) of the sockets was calculated.

Status of LCM

A manual version of LCM has been developed and used by both the Venus and 11/780 Task Forces. Recently an APL "breadboard" program has been written to test and demonstrate some of LCM's sensitivity features. It is planned to use this breadboard for future Venus tradeoff analysis.

Recommendations:

Because the Business Review Program (BURP) and LCM have a great deal in common, it is recommended that LCM's requirements, together with those of BURP, be automated in a new financial modeling tool. This new model would then have a 'business plan' mode appropriate to summary data and a 'tradeoff' mode with requirements for more detailed data. At least some of the sensitivity features described above would be available in both modes of operation.

lr

APPENDIX 1b

digital

INTEROFFICE MEMORANDUM

TO: Venus Task Force Working Group

DATE: 09 Jan 1980
FROM: Rolf McClellan
DEPT: Management Science Group
EXT: 223-9162
LOC/MAIL STOP: PK3-2/S53

SUBJECT: ADDITIONS AND CHANGES TO BURP REQUIRED FOR IMPLEMENTATION OF LCM

Attached is my preliminary list of LCM's requirements expressed as a list of proposed changes to BURP.

These preliminary requirements were reviewed by subsets of the Venus Working Group and the Product Management Support Team.

Some of the comments received from the latter group were:

(a) Generally the requirements would be nice to have, but for most products it would be difficult to gather marketing and functional cost estimates to the level of detail called for in the proposed version of LCM. (In other words, LCM seems to anticipate an improvement in forecasting and cost estimation and classification.)

(b) Most of LCM's requirements could be met by multiple runs of BURP supported by manual calculations.

(c) PMST would want to approve any enhancements to BURP, but would not be involved in the detailed specifications of changes.

(d) Some thought should be given to the issue of who should own and maintain BURP and any enhanced versions of it that evolve.

(e) Actual use of an LCM version of BURP was deemed to be approximately 2 years away due to present focus on current version of BURP.

(f) In addition to the attached requirements, enhancements to the "friendliness" of BURP are desired.

mb

Rolf McClellan
Management Science Group
January 9, 1980

LCM REQUIREMENTS

Purpose

The Life Cycle Model (LCM) is a financial tool that will enable product and program managers to evaluate various design, process, service and marketing alternatives based on after tax cash flows. Through consideration of all cash flows over a product's life time, the model's goal is to maximize a product's worth by achieving an optimal balance among functional costs and market variables.

By measuring the sensitivity of cash flow to the key parameters that determine profitability, LCM can help in assigning priorities to existing alternatives and to identify new ones worthy of consideration.

Relationship Between LCM and BURP

Because of the strong similarity between these two tools, it is anticipated that LCM would be implemented as an enhancement to the Business Review Program (BURP). The major areas of enhancement are:

(1) Inclusion of additional market parameters that impact sales volume (in addition to MLP, shipment delay)

(2) Inclusion of variables that drive Field Service costs (MTBF, MTTR, etc)

(3) Automatic computation of sensitivities of cash flow to key parameters. Determining changes in key variables that have offsetting effects on net present value (Gordon Bell's partial derivatives).

(4) Additional financial metrics calculated from the cash flows.

Relationship to Product Cost/Profit Models Used in the Functional Groups

The Life Cycle Model is intended as tool for summarizing and analyzing cost, volume, and revenue data projected by the different functional groups for a particular product. Therefore LCM would be supported by various product planning models that exist or may be developed. (e.g., Field service's LCBM and NEWREQ models) Here it would be important to ensure that the support models have output modes that are compatible with LCM's input requirements and that output data files could be transferred automatically or manually to LCM.

Outline of Proposed Additions/Changes to BURP

1. Shipment Volume

In addition to the base forecast, sensitivities of the forecast to changes in key factors are desirable inputs.

Factor	Δ Factor	Δ Volume
MLP	±20%	
Post Purchase Cost of Ownership (DEC BMC & nonDEC)	±20%	
Time of Product Availability (Time to Market)	±2Q	
Promotional Expenses	±10%	
Performance Metric	±20%	

For each factor the impact of the delta could be input as a shipment multiplier, or in the case of Product Availability as both a delay and a multiplier.

2. Cost Inputs

Cost inputs should be sorted by functional group and categorized by their nature (depreciable capital expense, non-depreciable capital expense, variable cost, and period cost).

Additionally, the input data should have a level of detail that exposes the primarily factors that determine variable cost (e.g. direct material and labor in volume manufacturing, and MTBF, MTTR, material cost of repair for field service).

Figures (1) thru (5) attached are manual worksheets that were developed to assist in the manual calculation of life cycle cost. These worksheets illustrate the partitioning of costs among functional groups and the classification of these costs as capital, variable, and period expenses. The highlighted lines would be inputs to LCM.

3. Ed Services, Customer Spares Costs and Revenues

For completeness, these should be included.

4. Working Capital (finished goods, W/P, A/R)

There should be the option to enter increments to working capital as non-depreciable capital expense by period. This would be a more flexible approach than the shifting done by BURP.

5. Financial Metrics

In addition to BURP's metrics, it would be desirable to calculate terminal values of NPV and IRR. (These entities assume reinvestment of positive cash flows at the opportunity cost of capital rather than the risk-adjusted rate or the IRR).

Furthermore, maximum exposure (cumulative negative cash flow) would be worth including.

The risk-adjusted discount rate and the opportunity cost of capital should be inputs to LCM.

6. Sensitivity Analysis

(a) Strategic Tradeoffs

This type of sensitivity analysis attempts to indicate how changes in various factors can affect product NPV through changes in market acceptance (shipment volume). The goal is to have a guide in adjusting product parameters to better serve the market and thereby to increase NPV. This guide would be used at an early phase of the development process prior to more detailed cost reduction analyses that are described in 7. below.

The procedure is to select a pair of variables from the following list:

(i) MLP
(ii) Transfer Cost } only one of these may be chosen in a given pair
(iii) Time to Market

(iv) Performance Metric
(v) BMC (Dec C.O.O.)
(vi) Cost of ownership (nonDEC)
(vii) Promotional Expense

For the factors chosen, the program would calculate (using the volume sensitivities of the shipment forecast) the effect on NPV of a 20% change (or 2Q in the case of time to market) in the first factor. Then the program would calculate the appropriate change in the second factor that would have offsetting NPV. (To do this the program may have to extrapolate beyond the ±20% range of the volume sensitivity estimation.) This procedure is equivalent to calculating a partial derivative with respect to these two factors with NPV and other factors held constant.

Example:

We'd like to have a rough estimate of how much of an increase we could tolerate in transfer cost to shorten the time to market. (MLP is assumed fixed.) A 2Q slip in TTM delays equipment NOR revenues and reduces volume and thus reduces product NPV by "$x". The program would determine (by trial and error and interpolation) the change in xfer cost that has the same $x impact on NPV, then:

$$\left.\frac{\partial \text{xfer cost}}{\partial \text{ TTM}}\right|_{\text{NPV = constant}} = \frac{\$\,\Delta \text{xfer cost calculated}}{2Q}$$

Similarly, we could calculate

$$\frac{\partial \text{Performance Metric}}{\partial \text{ xfer cost.}}$$

Then a third rough approximation would be possible.

$$\frac{\partial \text{Performance Metric}}{\partial \text{ TTM}} \approx \frac{\partial \text{ P.M.}}{\partial \text{ xfer}} \times \frac{\partial \text{xfer}}{\partial \text{TTM}}$$

(b) Risk Assessment

Assuming that the product has been defined in terms of market requirements, it would be desirable to know how much variation in key parameters is possible before NPV = 0 (or IRR = hurdle rate).

Factor	Δ Factor such that NPV of project = 0
MLP	
Volume	
Direct Material/Unit	
Direct Labor/Unit	
MTBF	
MTTR	
FA&T Expense	

Here it is assumed that the above factors are, in varying degrees, out of DEC's control.

7. Cost Minimization

Again with product definition as a given, it would be desirable to have a guide to aid in the prioritization of alternatives to be considered for reduction of cost. This could be accomplished if the program calculated how much each key factor would have to change to have a $\$10^6$ impact on NPV.

Factor	Δ Factor with NPV = $\$10^6$
Direct Material/Unit	
Direct Labor/Unit	
MTBF	
MTTR	
Mat'l Cost of Repair	
FA&T Expense	

Example: Suppose that a particular alternative offers the following tradeoff: By increasing Direct Material/ Unit by $500 for high reliability components, MTBF could be increased by 100 hours. If, in the above table, $500 in material/unit has a 10^6 impact on NPV, while 100 hours of MTBF has an NPV impact of only $300K (assuming linearity), the alternative would not appear worthy of detailed consideration.

8. Time

It would be desirable to add time zero (for discounting purposes) as an input to the program. This would allow the program to automatically account for sunk costs when analyses are performed in the course of the life cycle.

9. Outputs

In addition to the proforma cash flow statements of BURP, the following outputs would be desirable.

(a) Total Cash Flow

(b) Cash Flow by Functional Group

annual
discounted annual
cumulative.

mb
attachments

Figure 1

General + Revenue

G

(1) Shipments -

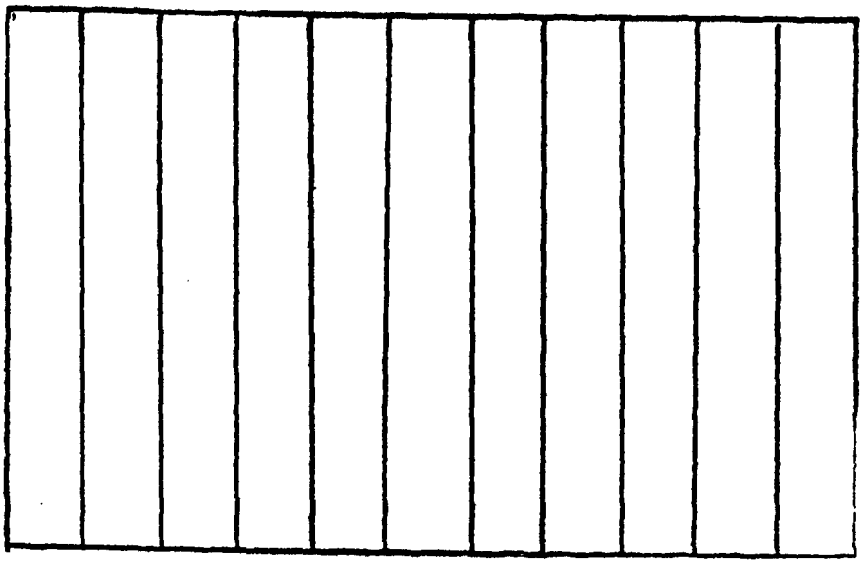
(2) MLP -

(3) Field Service BMC -

(4) SW Service NOR -

(5) Customer Training NOR

(6) Customer Spares NOR -



Shipment Volume Sensitivities

Factor	Δ Factor	Δ Volume
MLP	±20%	
Post Purchase C. of Ownership	±20%	
Time of Product Availability	±2Q	
Promotional Expenses	±10%	
Performance Metric	±20%	

E

Capital Expense

(1) Computer Equipment (5 year DDB)

(2) Purchased Equipment (8 year SYD)

(3) Building Cost (33 1/3 year SL)

(4) Building Improvement (20 year SL)

(6) Total Cap Expense

Period Expense

(7) Labor & Material

(8) NPSU

(9) Fixed Exp./Overhead

(10) Total per. exp. (7) + (8) + (9)

Total Eng'g Expense (6) + (10)

Marketing/Sales

Capital Expense

(11) Accounts Receivable
increment - decrement

(12) Other

(13) Total Cap. Exp.

Period Expense

(14) Advert

(15) Promotion

(16) Training

(17) Fixed Exp./Overhead

(18) Total per. exp.

Manufacturing--Volume

Capital Expense

(1) Computer Equipment (5 year DDB)
(2) Purchased Equipment (8 year SYD)
(3) Building Cost (33 1/3 year SL)
(4) Building Improvement (20 year SL)
(6) Addition to Avg Inventories
(7) Total Cap. Exp.

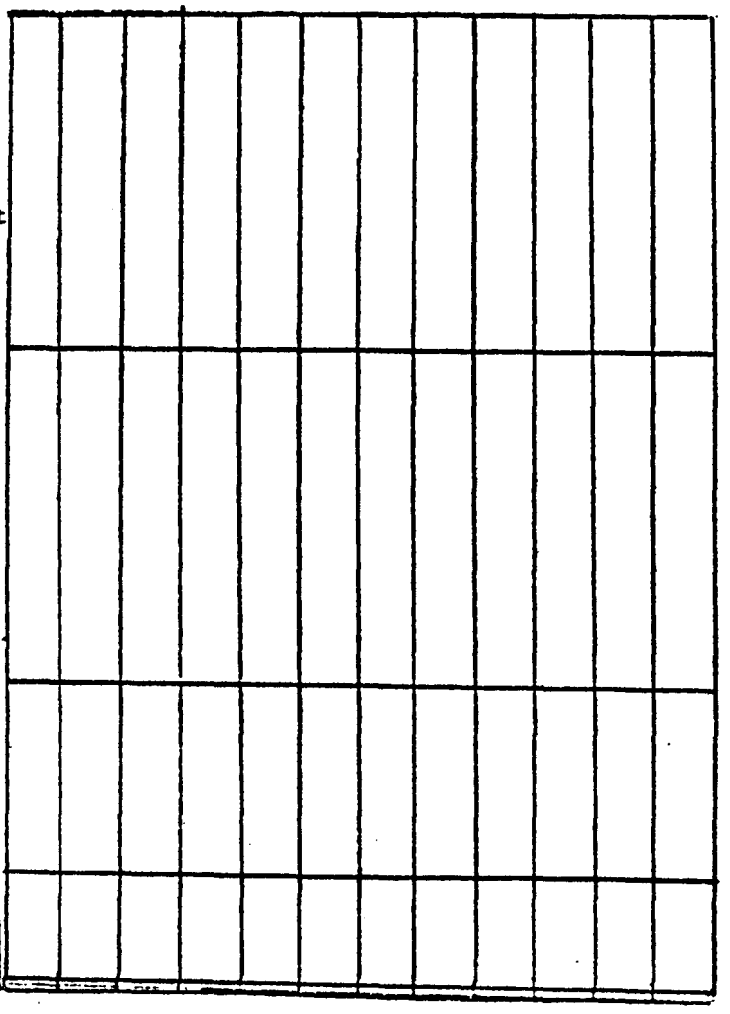
Variable Expense

(8) Labor Rate (Net Variable Overhead
(9) Direct Labor hrs/unit
(10) Direct Mat'l/unit
(11) Variable transfer cost/unit (8)(9) + (10)
(12) Total Var. Exp. (11) x Ships (61)

Period Expense

(13) NPSU
(14) Fixed Cost/Overhead
(15) Total Per. Exp. (13) + (14)

Total Vol. Mfg. Exp. (7) + (12) + (15)



Manufacturing--FA&T

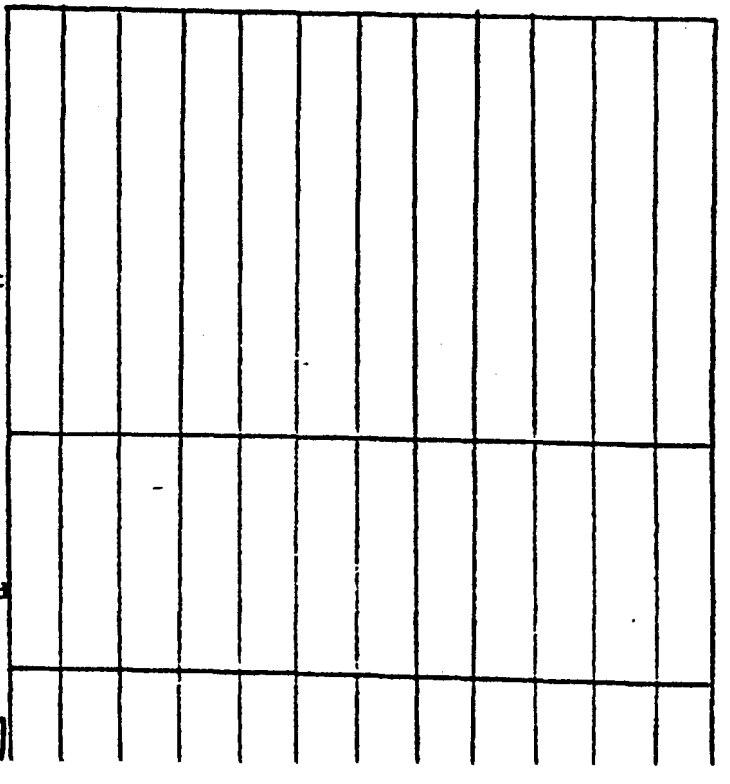
Capital Expense

(16) Computer Equipment (5 year DDB)
(17) Purchased Equipment (8 year SYD)
(19) Building Cost (33 1/3 year SL)
(20) Building Improvement (20 year SL)
(21) Addition to Avg. Inventories
(22) Total Capital Exp.

Period Expense

(23) NPSU
(24) Fixed Costs/Overhead
(25) Total Per. Exp. (23) + (24)

Total FA&T Mfg. Exp. (22) + (25)



Capital Expense

(1) Computer Equipment (5 year DDB)
(2) Purchased Equipment (8 year SYD)
(3) Building Cost (33 1/3 year SL)
(4) Building Improvement (20 year SL)
(6) Additions to Avg. Inventory (SR17 + C.D.]
(7) Total Cap. Expense

Variable Expenses

(8) Shipments
(9) Install Rate
(10) K penetration rate
(11) K renewal rate
(12) W. Period (years)
(13) Warranty Population unit years (8)×(9)×(12)
(14) K population unit years

$$= \left[\sum_{j=1}^{t=1} (8)_j \; (10)(11)^{t-1}\right] + \frac{(8)_1 \quad (10)}{2}$$

(15) MTBF (years)
(16) Duty Cycle
(17) Failures/unit-year

$$= \frac{(16)}{(15)}$$

(18) Labor Rate (Installation)
(19) Labor Rate (Repair)
(20) Installation hrs/unit
(21) MTTR (hrs)
(22) Mat'l Cost of Repair
(23) Variable Repair Cost per unit - year (17)x[(19)(21) +(22)]
(24) Installation Cost = (8)x(9)x(18)x(20)
(25) Warranty Cost = (13)x(23)
(26) Maintenance Cost = (14)x(23)
(27) Total "Variable" Cost (24)+(25)+(26)

Period Expense

(28) NPSU
Course Devel.
Product Support
Training
(29) Fixed Overhead
(30) Total Period Exp. (28) + (29)

Total Field Service Expense

S

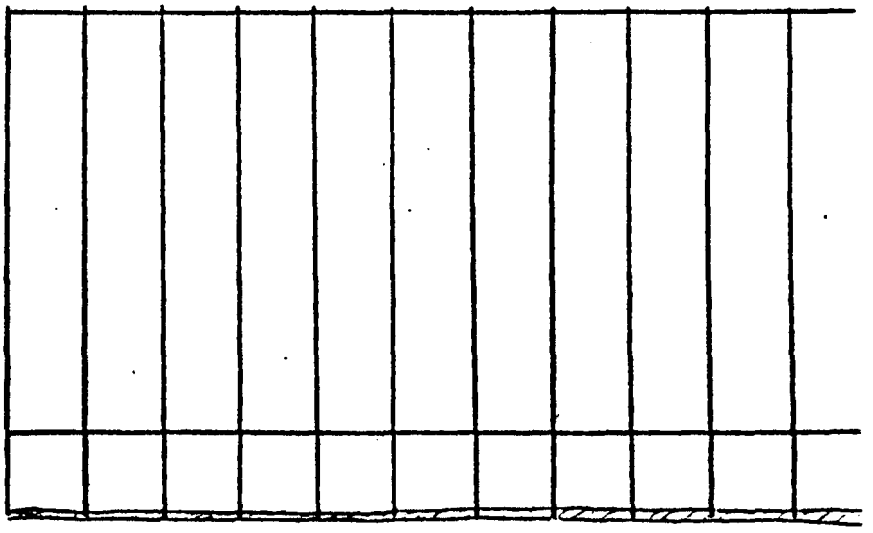
Variable Expense

(1) Installation Cost
(2) Warranty Cost
(3) Maintenance Cost
(4) Total (1)+(2)+(3)

Period Expense

(5) SW Development
(6) Other Fixed Expenses
(7) Total (5)+(6)

(8) Total Software Services
(4)+(7)



Customer Spares

Variable Expense

(9) Percentage of non-contract population purchasing spares

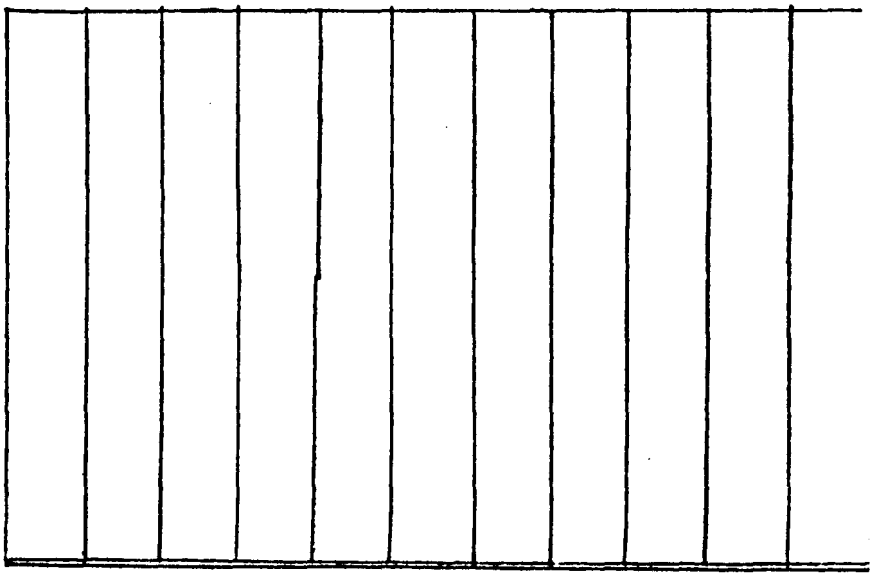
(10) Customer Spare Population in FYt

$$= \left\{ \sum_{j=1}^{t} (F8)_j - (F14)_t \right\} \times (9)$$

(11) Avg spare parts cost per unit-year

(F17) × (F22)

(12) Total customer spares cost
(10) x (11)



Customer Training

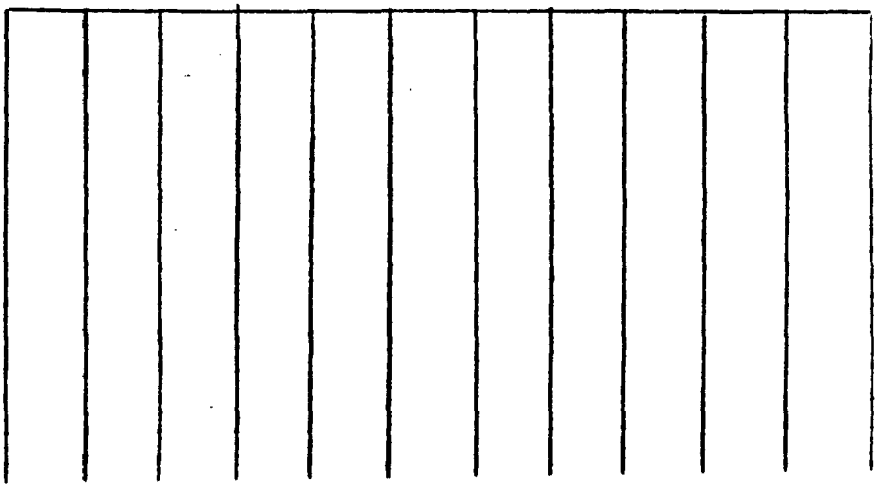
Variable Expenses

(13) # Trainees per year
(14) Variable cost/trainee
(15) Total variable expense
(13)×(14)

Period Expenses

(16) Course Development
(17) Training
(18) Fixed Expenses
(19) Total (16)+(17)+(18)

(20) Total Customer Training



APPLICATION OF SENSITIVITY ANALYSIS

Question: How many additional (unanticipated) intermittent failures of MCA sockets would cancel the benefits of incorporating these sockets in Venus. (Reference the study of 12/6/79)?

St = savings in year t (annual benefit)

$$\sum \frac{St}{(1+r)^t} = \text{Net present savings discounted at } r.$$

$$\text{Net present cost of additional failures} = \sum \frac{\left(\frac{\Delta \text{ failures}}{\text{unit-year}}\right) \times \left(\frac{\text{cost}}{\text{failure}}\right) \times \left(\text{contract population in year } t\right)}{(1+r)^t}$$

$$= \left(\frac{\Delta \text{failures}}{\text{unit-year}} \times \frac{\text{cost}}{\text{failure}}\right)\left(\sum \frac{\text{contract population in } t}{(1+r)^t}\right)$$

Cost and savings breakeven when:

$$\frac{\Delta \text{failures}}{\text{unit-year}} = \frac{\sum \frac{St}{(1+r)^t}}{\left(\frac{\text{cost}}{\text{failure}}\right) \times \sum \frac{\text{contract population in } t}{(1+r)^t}}$$

---

cost/failure = $356

Discount Rate	Net Present Savings	Breakeven Points: $\frac{\Delta \text{ failures}}{\text{unit-year}}$	Breakeven Points: $\Delta\lambda$ per sock
0	$4.54M	.238	.29
40% b.t.	$ .58M	.395	.48

digital

Appendix 1c

TO see "TO" DISTRIBUTION

cc: see "CC" DISTRIBUTION

DATE: SUN 2 DEC 1979 10:58 AM EST
FROM: GORDON BELL
DEPT: OOD
EXT: 223-2236
LOC/MAIL STOP: ML12-1 A51

SUBJECT: GETTING THE RIGHT DEVELOPMENT PRIORITIES SET FOR VENUS

Like the 2080, I am concerned that the development group from top to bottom have the same set of development goals. In talking with Jud, it is clear that this is not the case. In many respects I think the priorities should be about the same as the 2080, but possibly interchanging entry cost and performance.

To begin with, we need to write down a bunch of definitions as to just what the parameters are we have any control over. I assume these: t.development; perf., availability, t.repair, reliability, $.dev correlates with t.development; $.entry; $.ownership
After defining these, we need to define the interrelationships among them and get them crisp

Next, we have to get a standard set of assumptions about the market projection. Mine are roughly: 780 fcs may 78, 780 fvs sept 78, rate at about 100/month and 200K each until venus. I assume a linear ramp up in 6 mos from nov 82 till may 83, and then a 48 month run and then a 12 month gradual ramp down when venu replacement is announced in nov. 86...giving a life of rouglly that of 780. I also assume any delay in this will reduce 780 revenues and the venus plan will be shorter by the appropriate amount. This set of numbers, whatever (including the size distributions) should be used in the following analysis.

Now we enter the plan into BURP and begin to analyze some partial derivatives: d($.product cost)/d(t.development)---is my favorite. Namely using some really gross numbers of say $200K/machine, and 100 machines/month for 50 months, we get 20m (or 10m of cost/month) for sales (nor), and $1 b over machine's life. assume if the machine slips a month, we lose 20m and are spared 10m in cost. N Note the product contribution is the same, but the roi has gone way down becaau the 50m to bring to market is constant and probably increases as we slip, givin us a double whammy. In order to hold roi (by not slipping, I believe we can sp spend up to 10m for all the 5000 machines we will build for every month we migh slip...or a hit of 2K/machine. It isn't intuitively obvious how these all inte act, but the issue is the analysis of these critical partial derivatives to giv a constant roi.

The second partial of interest is d($.product cost)/d($.ownership). Namely how much can we add to product cost to reduce cost of ownership, while having const or increasing roi to the corp. Here, we have to get the service content model into burp (which is there if we use it). Issues like ecc on the cache, and various busses should pop out of the work.

Somehow I haven't seen any of the taskforce work on venus, and I want to. We have to get these critical issues resolved so our developers know which way to go. I would note that the straightforward 780 implementation at a relatively high product cost really paid off in time to market and reliability. I don't want another comet where we optimize the hell out fo cost and lose all.

"TO" DISTRIBUTION

APPENDIX 1D

LIFE CYCLE IMPACT OF DOCK MERGING 11/780 SYSTEMS

DEFINITION OF ALTERNATIVES

ALTERNATIVE	LEVEL OF FIELD INTEGRATION	% OF SYSTEM DOCK MERGEABLE*
A	CURRENT LEVEL	0
B	CPU OPTIONS	15%
C	SPC's CPU OPTIONS BULKHEAD CONNECTORS	50%
D	C PLUS EXPANSION BACKPLANES, AND BOXES, CABS	70%
E	D PLUS COMPLEX RECONFIGURATION	100%

*BASED ON AN ANALYSIS OF 50 11/780 SYSTEMS.

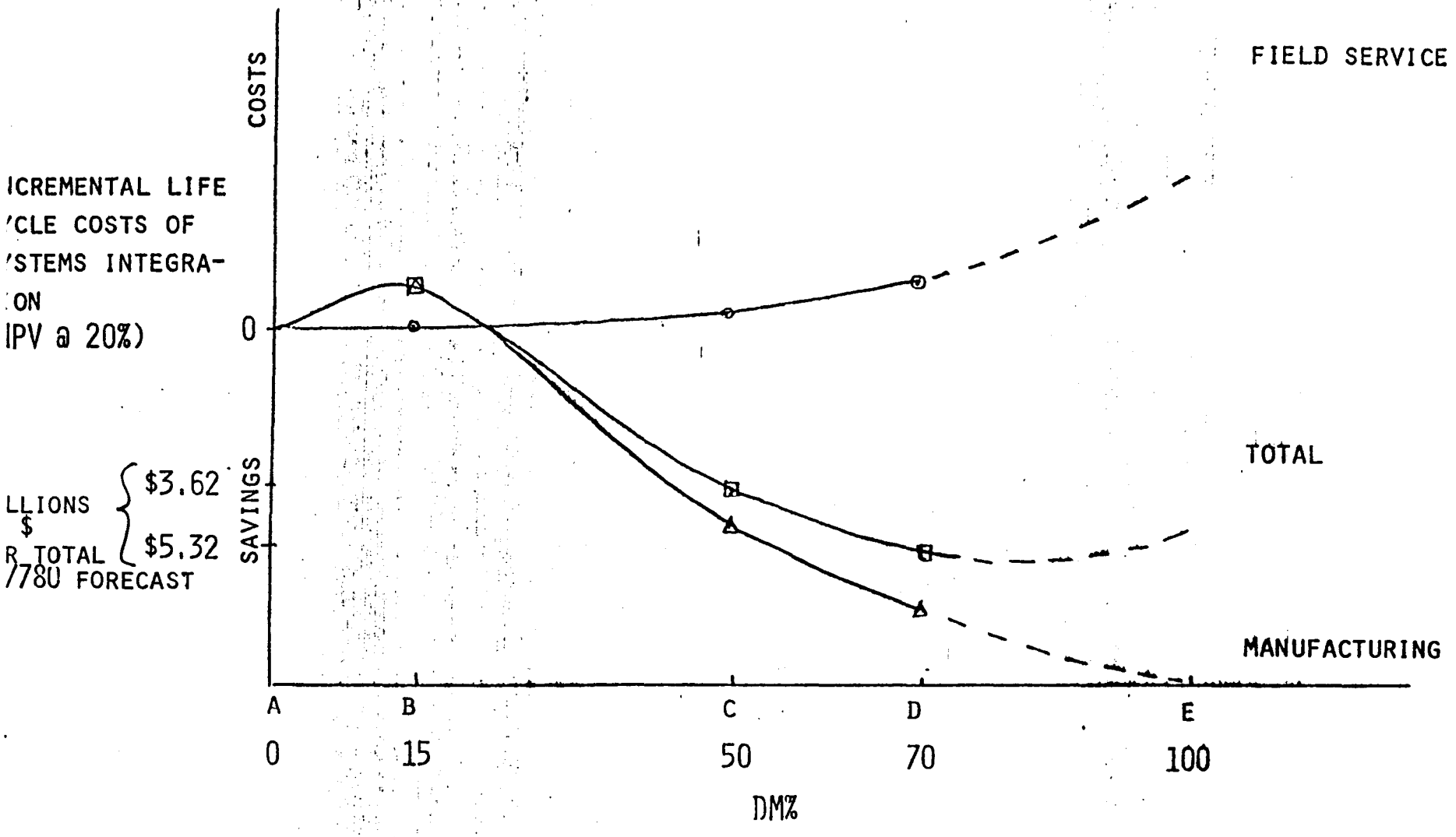
GENERAL ASSUMPTIONS

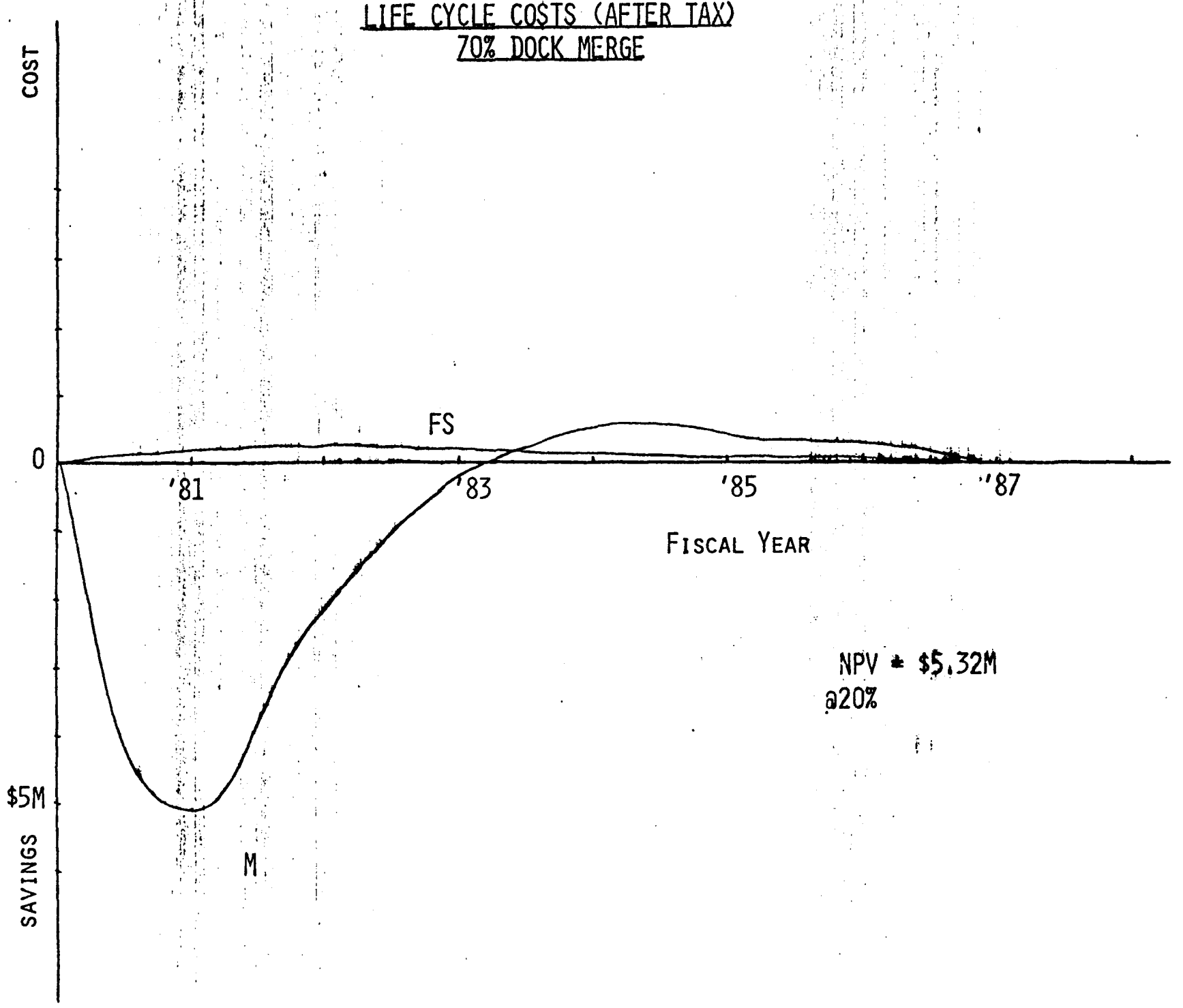
(1) DOCK MERGED SYSTEMS HAVE SAME QUALITY LEVEL AT INSTALLATION AS FA&T'D SYSTEMS.

(2) DOCK MERGED SYSTEMS ARE NOT SHORT SHIPPED MORE FREQUENTLY THAN FA&T'D SYSTEMS.

(3) VOLUME MANUF. INSTALLS KERNAL BACKPLANES.

SUMMARY OF RESULTS




LIFE CYCLE COSTS (AFTER TAX)
70% DOCK MERGE
COST
0
FS
'81
'83
'85
'87
FISCAL YEAR
NPV = $5.32M
@20%
$5M
SAVINGS
M

DETAILED ASSUMPTIONS

- INCREASED TESTING IN HVM RESULTS IN TRANSFER COST INCREASING BY $500/SYSTEM
- DOCK MERGING A SYSTEM RESULTS IN A SAVINGS IN FA&T OF $3600 (IN THE RANGE 50-100% DM) OR 1.8% OF MLP OF $200K
- WIP AVERAGES 2% ANNUAL NOR (FA&T SYSTEMS)
  WIP WOULD BE NIL FOR DOCK MERGED SYSTEMS.
- CARRYING COSTS AND "BRICKS AND MORTAR" IMPACT OF ΔWIP IS COVERED IN OVERHEADS INCLUDED IN $3600 (1.8%) FA&T SAVINGS
- FS INCREMENTAL EXPENSE REFLECTS ADD ON INSTALLATION CHARGES ONLY. 5 ADD ONS PER SYSTEM ARE EXPECTED AT 70% DOCK MERGE.
- FIRST IMPACT OF DOCK MERGE IS SEEN IN FY'81.

SENSITIVITY ANALYSIS

CHANGE IN KEY VARIABLES REQUIRED TO MAKE BASE CASE (FA&T) PREFERABLE TO DOCK MERGE ALTERNATIVES

50% ALTERNATIVE

Δ TRANSFER COST	$1645
Δ FAILURES/SYSTEM DURING I&W	4.1/SYSTEM
Δ FA&T SAVINGS	1.8% MLP

70% ALTERNATIVE

Δ TRANSFER COST	$2418
Δ FAILURES/SYSTEM DURING I&W	6.1/SYSTEM
Δ FA&T SAVINGS	1.9% MLP

OTHER CONSIDERATIONS

- BACKPLANES ARE EASILY DAMAGED IN SHIPMENT AND INSTALLATION.
- FIELD ASSEMBLY CALLED FOR IN ALT "D" MAY HAVE A NEGATIVE IMPACT ON CUSTOMER PERCEPTIONS OF DEC.

RECOMMENDATION

- ALL THINGS CONSIDERED, ALTERNATIVE "C" (50% OF SYSTEMS DOCK MERGED) IS PREFERRED.

Appendix 1e

digital

INTEROFFICE MEMORANDUM

TO: Distribution
CC: John Beisheim

DATE: 17 October 1979
FROM: Rob Hilbrink
DEPT: F.S. Strategy Analysis
EXT: 223-2767
LOC/MAIL STOP: PK3-2/F29

SUBJECT: VENUS LCC, BASE CASE (10/13/79 Data File

MANUFACTURING INPUT SHEET

Capital expense for equipment and facilities is self-explanatory. Investment in working capital is included in this category even though it is not depreciated; it is considered to be a cash flow at the time it occurs. For Manufacturing, working capital is assumed to consist of inventories only. Manufacturing inventory in the Venus base case is calculated at a standard 15 weeks of transfer value. The investment/(disinvestment) equals the annual increase/(decrease) in average annual inventory level.

Variable expenses can be input on a summary level or on a detail level depending on the design phase and data availability. In the past, it has been observed that variable manufacturing costs per unit are subject to learning curves and material cost trends. These trends can be formalized and incorporated in the LCC model.

In the presented Venus base case, the data file contains the unit cost for the 800th unit manufactured, in 1982 dollars. The average annual variable manufacturing costs for the Venus CPU have been calculated using a 92% learning curve and an 8% average annual rate of inflation.

New product start-up costs and overhead are input as period costs. In the current base case, overhead is allocated as a percent of labor costs.

FIELD SERVICE INPUT SHEET

Equipment capital expense consists of the incremental investments in field test equipment, product support equipment, and product engineering equipment.

Increase/(decrease) inventories contain the Field Service annual investments/(disinvestments) in working capital and is computed as the increase/(decrease) in average annual inventory levels of stockroom 17, pipeline and branches.

Variable Field Service expenses are calculated via a simplified annualized Business Plan submodel, which has been designed to allow LCC sensitivity analyses on such cost driving parameters as volume, MTBF, MTTR, and material cost/repair.

Computation of the total Field Service expenses using both the annualized LCC model and the quarterly LCBM model shows a variance of less than 10% between the outcome of either model, which is deemed acceptable.

In the LCC submodel both warranty and contract population are expressed in unit-years and computed using these algorithms:

Warranty pop. = (Install. Rate) x (Ship Volume) x (Warr. Period)

(1983 Venus: W. Pop. = 100% x 800 x .25 = 200)

Contract Pop. = (Contr. Pop. Prior Yr.) x Renewal Rate + ½ (Ships) x (Penetration Rate)

(1983 Venus: K. Pop. = 0 x .85 + ½ x 800 x .85 = 340)

MTBF is expressed in years, and the number of failures per unit-year is computed as:

Failures/Unit-Year = 1 ÷ MTBF

(Venus: Fail./Unit-Year = 1 - .34 = 2.94)

Labor rate installation is the standard labor rate excluding overhead (Maynard portion of CPH) and prorated for travel time, which is assumed to be 20% of installation time.

(Venus: Labor Rate Install. = 37.25 x 1.2 = 44.70)

Labor rate repair is the standard labor rate excluding overhead (Maynard portion of CPH) and prorated for travel time, support labor and preventive maintenance.

(Venus: Labor Rate Repair = (1 + .18 + .54 + .15) 37.25 = 69.66)
spt trv p.m

The variable total Field Service costs are calculated via the variable repair cost per unit-year.

(Var. Rep. Cost/Unit-Yr.) = (Fail./Unit-Yr.) (Labor Rate x MTTR + Material Cost/Repair)

(Venus: (Var. Rep. Cost/Unit-Yr.) = 2.94 x (69.66 x 3.3 + 277.72) = $1,492)

Total Field Service cost is then:

Install. Cost = (Install. Rate) x (Ships) x (Install. Hrs./Unit) x (Labor Rate)

('83 Venus: Install. Cost = 1 x 800 x 10 x 44.70 = $357.6)

Warranty Cost = (Warranty Pop.) x (Var. Rep. Cost/Unit-Yr.)

('83 Venus: Warranty Cost = 200 x 1,492 = $298.4)

Maintenance Cost = (Contract Pop.) x (Var. Rep. Cost/Unit-Yr.)

('83 Venus: Maintenance Cost = 340 x 1,492 = $507.3)

Course Development and Training expenses are input as period costs.

Product Support and Overhead are allocated as a function of direct labor hours as is currently the practice in the Field Service Business Model.

LCC MODEL OUTPUT

A graphic display of the model output is shown in attachments 1-3.

In addition to these graphs, a sensitivity analysis can be performed on the cost driving parameters identified in the model.

The following gives an example of a Manufacturing - Field Service cost trade-off using the LCC model on the Venus base case :

- Manufacturing LCC is $278,712 (variable portion).
- Manufacturing NPLCC @ 40% is $63,101 (base = FY'81).
- Field Service LCC is $86,831 (variable portion).
- Field Service NPLCC @ 40% is $9,847 (base = FY'81).

Conclusions:

A. A reduction of 5% in the unit manufacturing cost is acceptable only if MTBF does not decrease to below 2.77 mos. (ceteris paribus)

calc. $4.08 - [(63,101 \times .05) \div 9.847] \times 4.08 = 2.77$

B. An increase of 10% in the unit manufacturing cost is acceptable only if MTBF increases to at least 6.69 mos. (ceteris paribus)

calc. $4.08 + [(63,101 \times .1) \div 9,847] \times 4.08 = 6.69$ mos.

The same can be shown in matrix format:

Manufacturing NPLCC	$63,101
Field Service NPLCC	9,847
Total NPLCC	$72,948

	MTBF				
Variable Mfg. Costs	-40% 2.45	-20% 3.26	base case 4.08	+20% 4.90	40% 5.71
+10%	+10,249	+8,279	+6,310	+4,341	+2,371
+5%	+7,094	+5,124	+3,155	+1,186	-784
base case	+3,939	+1,969	72,948	-1,969	-3,939
-5%	+784	-1,186	-3,155	-5,124	-7,094
-10%	-2,371	-4,341	-6,310	-8,279	-10,249

Conclusions:

A. An increase in unit manufacturing cost of 5% can be offset by increasing MTBF with a percentage between 20 and 40%.

B. An increase of 10% in unit manufacturing cost cannot be offset by an MTBF increase of less than 40%.

/jfm
Encs.

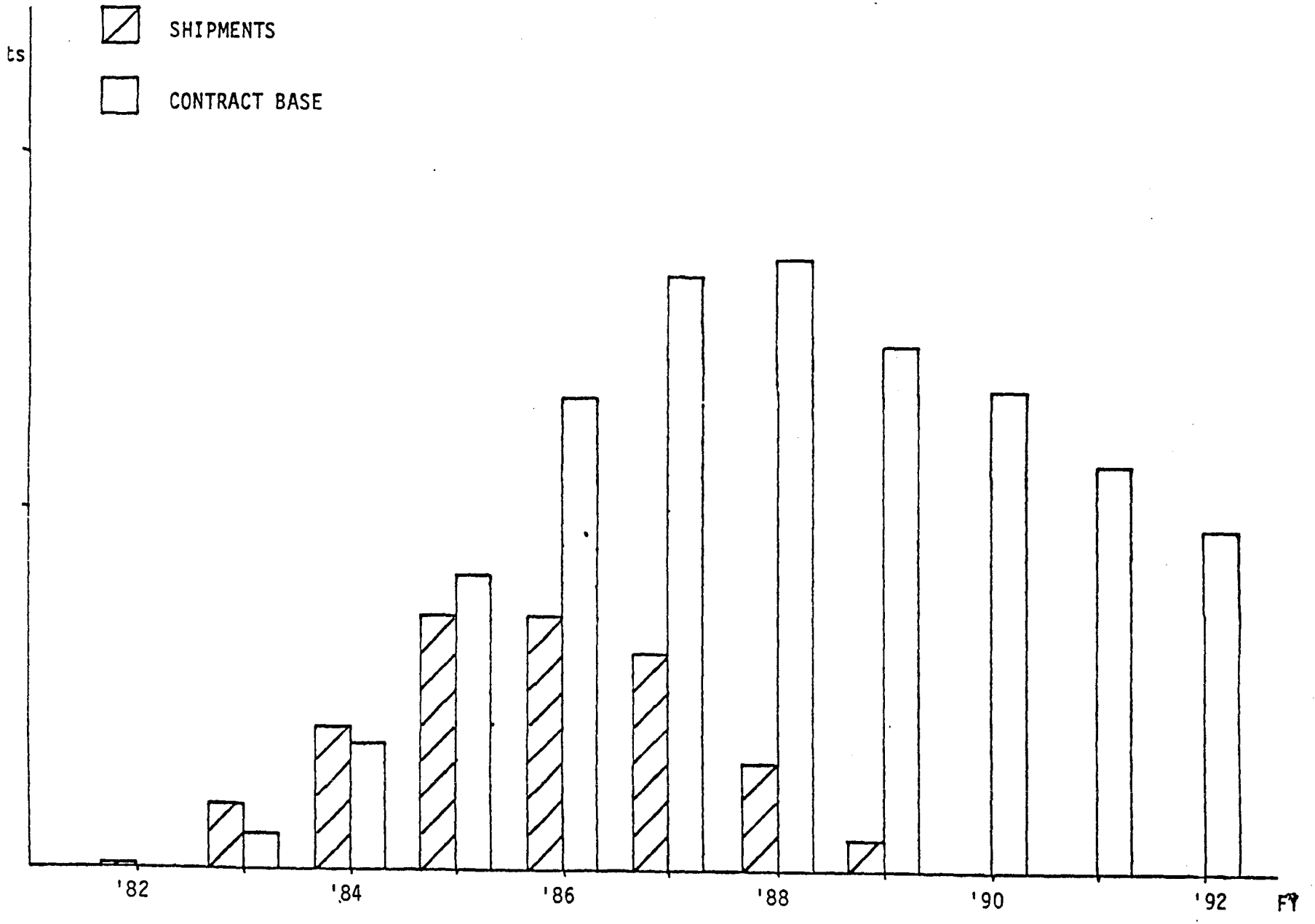
VENUS BASE CASE SENSITIVITY ANALYSIS

Unit Manufacturing Cost Variance	LCC Variance	NPLCC Variance
+10%	+27,871	+6,310
+ 5%	+13,936	+3,155
base case	364,166	72,948
- 5%	-13,936	-3,155
-10%	-27,871	-6,310
MTBF Variance		
+10%	-8,545	-985
+ 5%	-4,273	-492
base case	364,166	72,948
- 5%	+4,273	+492
-10%	+8,545	+985
MTTR Variance		
+10%	+3,845	+443
+ 5%	+1,880	+217
base case	364,166	72,948
- 5%	-1,880	-217
-10%	-3,845	-443
Material Cost Per Repair Variance		
+10%	+4,700	+542
+ 5%	+2,307	+266
base case	364,166	72,948
- 5%	-2,307	-266
-10%	-4,700	-542

R.H.
10/17/79

LCC MODEL - VENUS BASE CASE

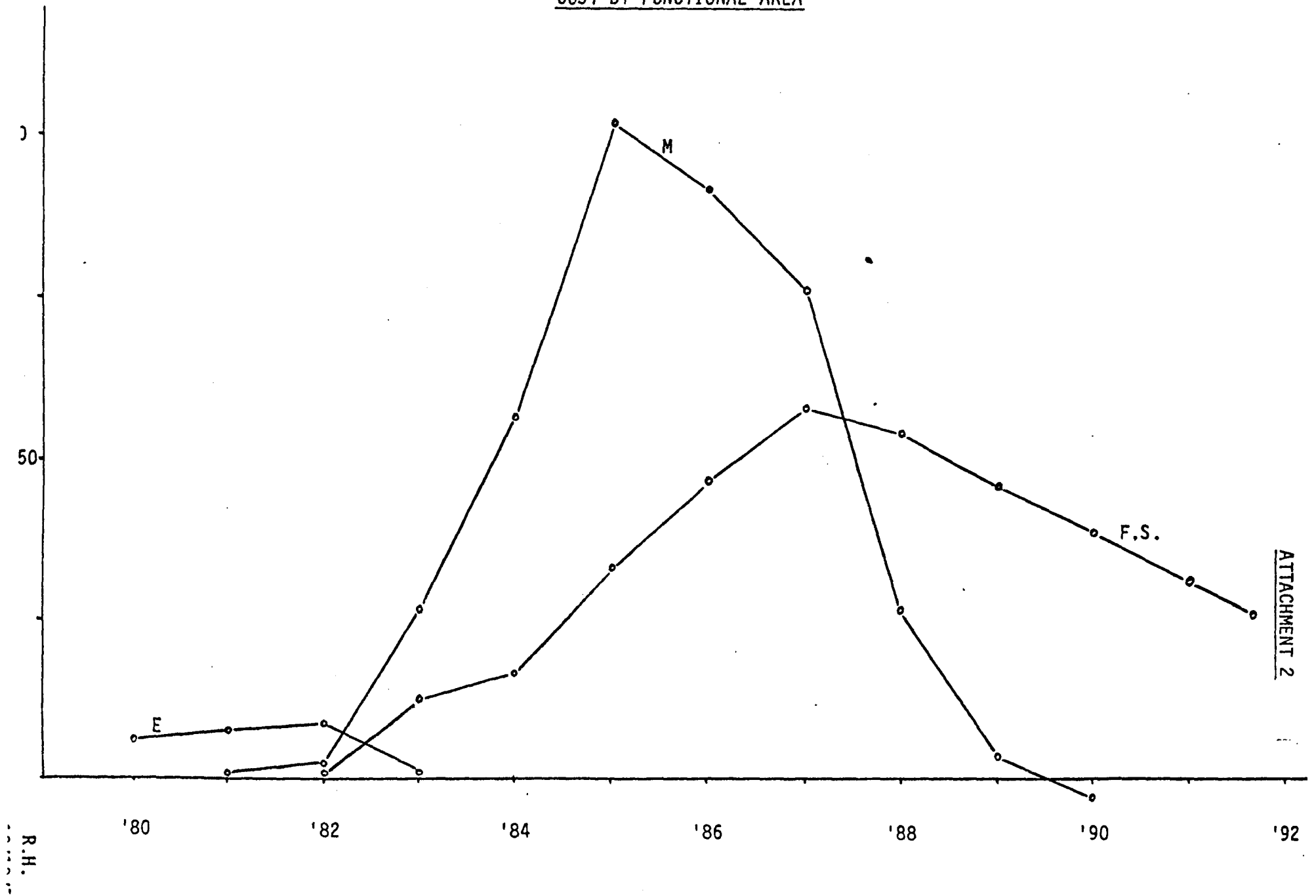
POPULATION



ATTACHMENT 1

LCC MODEL - VENUS BASE CASE

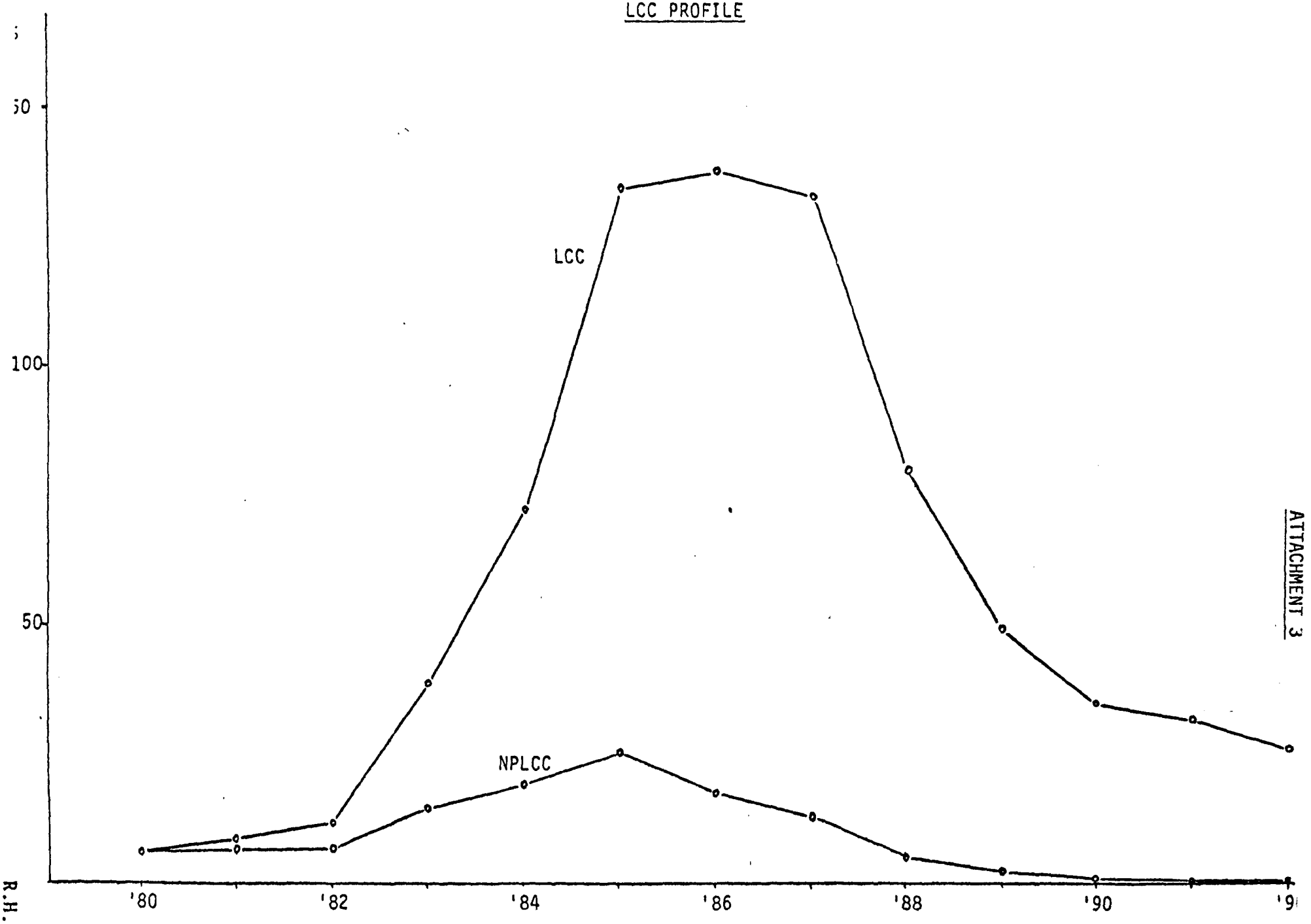
COST BY FUNCTIONAL AREA



ATTACHMENT 2

LCC MODEL - VENUS BASE CASE

LCC PROFILE



ATTACHMENT 3

Life Cycle Cost Data Input

Product: VENUS

FCS: Q4 82

FY	80	81	82	83	84	85	86	87	88	89	90	91	92
GENERAL													
Ship Volume			30	800	1900	3500	3500	3000	1400	400			
ENGINEERING													
Capital Expense													
Equipment	250	250											
Facilities													
Total Cap. Exp.	250	250											
Period Expense													
Labor & Mat'l													
NPSU													
Fixed Exp/Over-head													
Total Period Expense	6,100	7,500	8,300	300									
Total Engineering Expense	6,350	7,750	8,300	300									
MARKETING/SALES													
Capital Expense													
Accounts Recv													
Other													
Total Cap. Exp.													
Period Expense													
Advertising													
Promotion													
Training													
Fixed Expense /Overhead													
Total Period Expense													
Total Marketing/ Sales Expense													

FY	80	81	82	83	84	85	86	87	88	89	90	91	92
MANUFACTURING VOLUME													
CAPITAL EXPENSE													
Equipment			1,482	1,186	593	296	296						
Facilities													
Incr/(Decr) Inventories			256	5,282	5,862	8,893	303	(2,336)	(9,860)	(5,827)			
Total Cap. Exp.			1,738	6,468	6,455	9,189	599	(2,336)	(9,860)	(5,827)			
VARIABLE EXPENSE													
Labor Rate													
Direct Labor Hrs/unit													
Direct Mat'l/ unit													
Var. cost/unit*			26.4	23.8	19.2	18.4	18.4	19.0	20.0	21.5			
Total Var. Expense			792	19,040	36,480	64,400	64,400	57,000	28,000	8600			
PERIOD EXPENSE													
NPSU	344	1,081	827										
Fixed cost/ overhead			65	1721	4087	7,529	7,529	6453	3,011	860			
Total Period Expense	344	1,081	892	1,721	4,087	7,529	7,529	6,453	3,011	860			
TOTAL VOL. MFG. EXPENSE	344	1,081	3,422	27,229	47,022	81,118	72,528	61,117	21,151	3,633			
MANUFACTURING F.A. & T.													
CAPITAL EXPENSE													
Equipment			120										
Facilities													
Incr/(Decr) Inventories													
Total Capital Expense			120										
PERIOD EXPENSE													
NPSU	54	178	309										
Fixed Cost/ Overhead**		--	120	2,592	5,335	9,497	9,639	8,545	3,931	1,204			
Total Period Expense	54	178	429	2,592	5,335	9,497	9,639	8,545	3,931	1,204			
TOTAL MFG. F.A. & T. EXPENSE	54	178	549	2,592	5,335	9,497	9,639	8,545	3,931	1,204			

* Var. cost/unit = Direct Labor Hrs/unit × Labor Rate + Direct Mat'l/unit

** FWAT Cost = 13.5% of mkg exp

FY	80	81	82	83	84	85	86	87	88	89	90	91	92
FIELD SERVICE													
CAPITAL EXPENSE													
Equipment			165	270									
Incr/(Decr) Inventories			90	799	1,368	2,261	1,941	1,641	(249)	(676)	(1000)	(926)	(788)
Total Cap. Exp.			255	1,069	1,368	2,261	1,941	1,641	(249)	(676)	(1,000)	(926)	(788)
VARIABLE EXPENSES													
Installation Rate			100	→	→	→	→	→	→	→	→	→	→
Contr. Penetration rate			85	→	→	→	→	→	→	→			
Contr. Renewal Rate			85	→	→	→	→	→	→	→	→	→	→
Warranty Population (unit yrs)*			7	200	475	875	875	750	350	100			
Warr. period(yrs)			.25	→	→	→	→	→	→	→			
Contr. Population*			340	1437	3516	5963	7832	8527	8013	6981	5934	5044	
MTBF (years)			.34	→	→	→	→	→	→	→	→	→	→
Duty Cycle													
Failures/unit-year*			2.94	→	→	→	→	→	→	→	→	→	→
Labor Rate(Installation)*			44.70	→	→	→	→	→	→	→	→	→	→
Labor Rate(Repair)*			69.66	→	→	→	→	→	→	→	→	→	→
Install. hrs/unit			10	→	→	→	→	→	→	→	→	→	→
MTTR (hrs)			3.3	→	→	→	→	→	→	→	→	→	→
Mat'l Cost/repair			277.72	→	→	→	→	→	→	→	→	→	→
Installation Cost*			13	358	849	1.565	1,565	1,341	626	179			
Warranty Cost*			11	298	707	1,301	1,301	1,185	521	148			
Maintenance Cost*				507	2,144	5,246	8,897	11,685	12,722	11,955	10,416	8,854	7,52
Total Variable Exp.			24	1,163	3,700	8,112	11,763	14,141	13,869	12,282	10,416	8.854	7,52
PERIOD EXPENSES													
Course Develop.		20	200	50									
Product Support			4	134	330	625	676	637	398	230	146	124	105
Training			561	2667	980	863	885	853	836	693	632	506	407
Fixed Overhead			7	208	543	1,069	1,264	1257	990	723	551	468	398
Total Period Ex		20	772	3059	1,853	2,557	2,825	2,747	2,224	1,646	1,329	1.098	910
Total Field Service Expense		20	1,051	5,291	6,921	12,930	16,529	18,529	15,844	13,252	10,745	9,026	7,64

* For algorithms see pg. 5

	80	81	82	83	84	85	86	87	88	89	90	91	92
Variable Expense													
Installation Cost													
Warranty Cost													
Maintenance Cost													
Total													
Period Expense													
SW Development													
Other Fixed Expenses													
Total													
Total Software Services													

Customer Spares

Variable Expense													
Percentage of non-contract population purchasing spar													
Customer Spare Population in FYt (unit yrs.) (algorithm on pg. 5)													
Avg spare parts cost per unit-year													
Total customer spares cost													

Customer Training

Variable Expenses													
# Trainees per year (unit wks)			60	1600	3000	5500	6500	7000	3000	1200			
Variable cost/~~trainee~~ unit wk			350	400	420	450	480	530	570	600			
Total variable expense (K $)			21	640	1,260	2,475	3,120	3,710	1,710	720			
Period Expenses													
Course Development			100		25		25		25				
Training													
Fixed Expenses			3	80	180	330	455	490	240	96			
Total Period Exp.			103	80	205	330	480	490	265	96			

Life Cycle Model Algorithm

MANUFACTURING - VOLUME

Variable cost/unit = (labor rate including variable overhead) x (direct labor hours/unit) + (direct material/unit)

(Total variable expense) = (ship volume) x (variable cost/unit)

FIELD SERVICE

Warranty population unit - years = (ship volume) X (installation rate) x (warranty period years)

Contract population = (contract population previous year) x (renewal rate) + ½ (shipments) x (penetration rate)

Failures per unit year = ~~[(MTBF hrs) x (duty cycle)]~~ $\frac{\text{Duty Cycle}}{MTBF}$

Variable repair cost per unit year = (failures per unit year) x [(labor rate)(MTTR) + (material cost/repair)]

Installation cost = (Ship Volume) x (Installation Rate) x (labor rate installation) x (Installation hours/unit)

Warranty cost = (warranty population) x (variable repair cost per unit year)

Maintenance cost = (contract population) x (variable repair cost per unit year)

Labor Rate Installation equals labor cost per hour prorated for travel time.

Labor Rate equals labor cost per hour prorated for travel time; preventive maintenance and nuisance calls.

$Cust.\ Spares\ Pop_{(t)} = \{Cum.Ships_{(t-1)} + \frac{1}{2} Ships_{(t)} - Contr.pop.\} x$ (% non contr. pop. buying spare

digital

INTEROFFICE MEMORANDUM

TO: Distribution

CC: Bob Cirrone

DATE: 15-NOV 79
FROM: Rob Hilbrink
DEPT: Field Service Finance
EXT: 2767
LOC/MAIL STOP: PK302/F29

SUBJECT: SAMPLE COST TRADE-OFF CALCULATIONS

Vic Ku requested to see how the LCC model can be used to make cost trade-offs between Engineering, Manufacturing and Field Service, specifically what are the minimum required cost savings in Manufacturing or Field Service for each additional dollar invested in Engineering development. Table 1 shows the total LCC for the Venus base case (MCA socketing), and sections A and B give some sample computations.

In section A is assumed an incremental Engineering cost of $100K in 1980, which at the minimum, should result in a Manufacturing cost saving of $38 per CPU-kernal, or in a Field Service cost saving of $16 per service year.

In section B is assumed an incremental Engineering cost of also $100K, but spread evenly over the 4 years in which Engineering costs occur (i.e. $25K per year, 1980-1983). This should, at the minimum, result in a Manufacturing cost saving of $25 per CPU-kernal or in a Field Service cost saving of $10 per service year.

Section C gives the mathematical equations used to arrive at these results, where the second equation implicitly implies that the cost savings are distributed proportionately over the years in which base case costs occur.

TABLE 1

VENUS CPU (WITH MCA SOCKETS) LIFE CYCLE COSTS

(K$)

	ENGINEERING		MANUFACTURING		FIELD SERVICE	
FY	LCC	NPLCC	LCC	NPLCC	LCC	NPLCC
80	6,350	6,350	344	344	--	--
81	7,750	5,534	1,081	772	20	14
82	8,300	4,233	3,422	1,745	1,040	530
83	300	109	27,229	9,911	4,959	1,805
84			47,022	12,226	6,313	1,641
85			81,118	15,088	11,302	2,195
86			72,528	9,646	15,545	2,067
87			61,117	5,806	17,469	1,660
88			21,151	1,438	15,525	1,056
89			3,633	174	12,784	614
90					10,457	366
91					8,953	224
92					7,572	142
TOTAL	22,700	16,226	318,645	57,150	112,439	12,314

A. Engineering costs increase with \$100K in 1980.

$$\Delta \text{LCC ENG} = \$100K \Longrightarrow \Delta \text{NPLCC ENG} = \$100K$$

This increase can be offset by:

- Manufacturing:

$$\Delta \text{NPLCC MFG} = \$(100)K \Longrightarrow \Delta \text{LCC MFG} = \$\ (558)K$$

or \$(38) per device

- Field Service:

$$\Delta \text{NPLCC F.S.} = \$(100)K \Longrightarrow \Delta \text{LCC F.S.} = \$(913)K$$

or \$(16) per device per year

B. Engineering costs increase with \$25K per year (1980-1983).

$$\Delta \text{LCC ENG} = \$100K \Longrightarrow \Delta \text{NPLCC ENG} = \$65K$$

This increase can be offset by:

- Manufacturing:

$$\Delta \text{NPLCC MFG} = \$(65)K \Longrightarrow \Delta \text{LCC MFG} = \$(362)K$$

or \$(25) per device

- Field Service:

$$\Delta \text{NPLCC F.S.} = \$(65)K \Longrightarrow \Delta \text{LCC F.S.} = \$(594)K$$

or \$(10) per device per year

C. In General:

$$\Delta \text{NPLCC ENG} = -\Delta \text{NPLCC MFG.}$$

$$\frac{\Delta \text{NPLCC MFG}}{\text{NPLCC MFG.}} = \frac{\Delta \text{LCC MFG.}}{\text{LCC MFG.}}$$

$$\frac{\Delta \text{LCC MFG.}}{\text{TOTAL SHIPS}} = \Delta \text{MFG Cost/Ship}$$

R. Hilbrink
11/15/79

! d i g i t a l !

INTEROFFICE MEMORANDUM

TO:- VENUS TASK FORCE
CC:- MIKE ROBEY
ART O'DONNELL

DATE: 26 JUNE 1980
FROM: REG. BURGESS.
DEPT: LSG CSSE
EXT: 231-4484
LOC/MAIL STOP: MR1-1/S35

SUBJECT:- Report on Venus Task Force Working Group Activities.

The following text is a supplement to the attached slides and should be read with them.

It is essentially an attempt to document the words I normally say during the presentation of these slides.

I will adopt a format which I hope aids the reader by reproducing the lines from the slides in upper case and placing the text between them in lower case.

Something will naturally be lost due to the fact that presentations are a form of two way communication whereas this text is one way. I will try to compensate for this somewhat by writing a little more than I usually say and including the explanations which normally form the answers to the usual questions which arise.

Interested parties are welcome to contact me for a 'live' presentation where question and answer sessions can take place, should they feel that to be desirable.

Reg. Burgess.
26 June 1980.

LIFE CYCLE MODEL

TRADITIONAL APPROACH

TRANSFER COST GOALS DOMINATE OTHER COSTS

DIFFICULT TO SELL COST-JUSTIFIED INCREASES IN TRANSFER COST.

Historically transfer cost goals have dominated other costs in the development of a new product. From the Customer Services viewpoint it has been extremely difficult to convince development groups that transfer cost should increase in the interest of Warranty and Contract cost savings. The development engineer being measured more on transfer cost than on total cost to the customer or to digital throughout the entire life cycle of the product.

WITHIN VENUS PROGRAM

A PRIME GOAL IS TO MINIMIZE LIFE CYCLE COST

USING QUICK APPROXIMATIONS TO IDENTIFY RAMP ITEMS WHICH ARE EITHER CLEAR WINNERS OR CLEAR LOSERS.

On a daily basis design decisions are made which affect the RAMP profile of the product. Often times a particular feature may appear to cost very little but offer the opportunity for substantial savings in manufacturing test and repair or service costs. Typically features such as these are included. Features which appear to cost a very great deal and are perceived to return very little in life cycle cost savings have not been included, unless they also provide a potential customer benefit such as increased assurance of data integrity.

Most of the RAMP features of VENUS have been included as a result of such very quick approximations, this has typically been fairly easy since it has been necessary to consider only one or two variables when making these trade-offs.

Reg. Burgess.
26 June 1980.

LIFE CYCLE MODEL

WITHIN VENUS PROGRAM

EASIER TO SELL COST TRADE-OFFS.

As a result of the goal to minimize life cycle costs it has become easier within the venus program to sell cost trade-offs.

NEEDED

QUANTITATIVE METHOD FOR EVALUATING INTER-DEPARTMENTAL COST TRADE-OFFS.

For somewhat more complex items which involve changes in many parameters across several functional areas it is clearly necessary to develop methods for evaluating interdepartmental cost trade-offs and making the recommendation which will lower the total corporate cost, even though some costs within some functional areas may increase.

WITHIN VTFWG

UNIFORM METHODOLOGY DEVELOPED FOR QUANTIFYING COSTS AND SAVINGS TO EACH DEPARTMENT, TEST CASE NEEDED.

The VTFWG has provided the forum for us to understand the characteristics of costs incured in each other's areas. We have developed the methodology whereby we can assemble the total corporate costs over time and recommend a decision to the VENUS program. The model we are using is a standard discounted cash flow analysis. Clearly a test case was needed, preferably a fairly complex one, but not too complex for us to check the results against our intuition.

SOCKETS CHOSEN AS TEST CASE.

This seemed to be about as complex an issue as we wanted to handle for a test case. It is also controversial in that (some) additional failures will occur as a result of the sockets themselves and it was thought by some that the cost of the additional failures would more than outweigh any material cost saved by replacing components in the field.

Reg. Burgess.
26 June 1980.

SOCKET STUDY FOR MCAs AND RAMs ON VENUS

INITIAL STUDY NOV '79

REPAIR MATERIAL vs **PRODUCT COST** and **REPAIR LABOUR**

The savings are clearly to be found in the amount of material required to support the product. The failure of a $2Ø RAM will cost the same $2Ø in consumed material to replace, but if the replacement is done at the customer's site there will be less total material required to: a) support repair pipelines; b) reside in inventories at the Branch level; c) be held as safety stock at higher levels of the logistics hierarchy.

Against these savings there will be additional costs incurred to manufacture the boards with sockets and to repair the additional failures which the sockets themselves contribute. These additional costs have been included in this study, worst case numbers have been used throughout.

The savings are expresses as a delta study against a set of cost curves for a module replacement approach which is assumed to be 1ØØ% successful and achieves fault isolation to a single board every time, i.e. no bad boards in the kits and no good boards going back to repair centres because the fault could not be isolated to a single board on site.

In modelling the cost impacts of the sockets a very high cost socket was used, along with a very pessimistic failure rate. This ensures a worst case study and produces minimum savings, since the errors due to worst caseing will be compounded in some of the calculations it is likely that the actual savings could be several times greater than this study shows.

The following product volume and time frame were used as input assumptions, the distribution of failures within the CPU cluster were taken from the Phase Ø profile of the product.

OVER 1Ø YEAR LIFE CYCLE (5 PRODUCTION PLUS FURTHER 5 SUPPORT)

PRODUCT VOLUME OF 12500 SYSTEMS.

The following results were obtained:-

POTENTIAL $4.5M SAVINGS BY USING MCA SOCKETS.

POTENTIAL $1Ø.8M SAVINGS BY USING RAM SOCKETS.

Reg. Burgess.
26 June 198Ø.

SOCKET STUDY FOR MCAs AND RAMs ON VENUS

No work has been done to accurately understand the benefit of the additional revenue deriving from the additional systems which could be sold as a result of the reduction in spares requirement. However a 15% - 20% increase in first year shipments would be my personal estimate of the number of additional processors that could be shipped if the spares demand were reduced in this way. The implicit assumption being that processor modules are the resource restricting volume the most.

Other work within the VTFWG addresses discount rate, at the time this study was done we agreed to use 40% and to discount all the way back to FY80, even though the spending to achieve this alternative does not start until FY82. So not only were worst case technical parameters used but we also discounted at the highest rate we could find, producing a net present value which grossly understates the potential value of socketing these parts.

WHEN DISCOUNTED @40%, $580K SAVINGS BY USING MCA SOCKETS.

WHEN DISCOUNTED @40%, $1,526K SAVINGS BY USING RAM SOCKETS.

SENSITIVITY STUDY FEB '80

DETERMINE THE ADDITIONAL INTERMITTENT FAILURE RATES OF SOCKETS TO REDUCE THEIR NPV TO ZERO.

This was a request to determine risk, the results show the point at which the sockets still pay for themselves, but only just.

FOR MCA SOCKETS, ORIGINAL FAILURE RATE ESTIMATE TIMES 135%

FOR RAM SOCKETS, ORIGINAL FAILURE RATE ESTIMATE TIMES 2213%

CONCLUSIONS

POTENTIAL RETURN WORTH A MANAGEABLE AMOUNT OF RISK.

VERY LOW RISK OF A ZERO (OR NEGATIVE) RETURN.

Indeed there is a very high probability of a very much higher return.

Reg. Burgess.
26 June 1980.

digital

INTEROFFICE MEMORANDUM

TO: VTF

DATE: April 9, 1980
FROM: B. Campelia, R. McClellan,
DEPT: D. Wetherbee
EXT: 231-6887
LOC/MAIL STOP: MR1-2/E78

SUBJECT: REQUIREMENTS FOR SUCCESSFUL APPLICATION OF DISCOUNTED CASH FLOW TECHNIQUES IN PRODUCT INVESTMENT ANALYSIS

One of the primary purposes for which the VTF was founded was to develop a model that would encompass all the financial parameters that effect the Life Cycle return to DEC and guide us towards an optimum solution. This model has been developed and is in use today in conjunction with other models such as BURP and LCBM. All of these models employ one or more discounted cash flow techniques to measure the financial performance of the planned investment or, in the case of Venus, to measure the impact of specific changes in the plan.

The purpose of this paper is to demonstrate the need for such analysis, the problems inherent in its use, and a recommendation that the entire area of Product Investment Analysis requires corporate level definition, administration, and leadership. We believe that this type of analysis is critical to the continued success of DEC and as such, its implementation and management must come from a corporate level of authority.

Why Use Discounted Cash Flow (DCF) Analysis?

Decisions regarding investments in new plant and equipment or product development are typically based on financial analysis together with a number of extra-monetary considerations such as Technological trends or corporate and marketing strategy.

Analysis of the financial implications of an investment typically would include estimation of pay back period, maximum cumulative negative cash flow (exposure), as well as calculation of the DCF metrics, net present value (NPV) and internal rate of return (IRR). The DCF metrics are of

prime importance because they take into account the time value of money and are uniquely suited to determine whether an investment has sufficient return to contribute positively toward corporate financial goals and which of several investment alternatives offers the best return.

What are the Prerequisites for DCF Analysis?

(1) Financial strategy specifying a target return on capital

A major aspect of financial strategy is centered on providing the funds required to support long term growth of the corporation. This growth strategy is implemented by combining the growth rates of sales and earnings along with dividend policy, target debt to equity proportions, and target asset turnovers to specify a required return on invested capital. This required return may be expressed as return on stockholder's equity (ROE) or return on total capital (ROC). (capital = permanent interest-bearing debt plus stockholder's equity).

The target ROC, then, is the return that the corporation must realize on its average investment in order to finance growth in a fashion consistent with financial policies and investor expectations. In other words, the ROC would be the appropriate discount rate with which to compute the NPV of the average investment.

(2) Choice of the appropriate discount rate

While the target ROC is the proper discount rate for an investment of average risk, it may not always be appropriate for a particular investment decision.

For example, the opportunity cost of the capital being requested may differ significantly from the ROC. In this case, the NPV of the project should be computed using the opportunity cost of capital rather than target ROC.

Another situation where the applied discount rate is inappropriate exists when the corporation adopts a policy of adjusting the ROC in proportion to the riskiness or uncertainty of the project. Here the risk adjustment should not be arbitrary, nor universally applied. Rather, risk adjustments should be based on and applied to classifications of projects with simular amounts of risk. Each class of investment should be assigned a discount rate, determined from an analysis of historical and forecasted data. The weighted average of class-specific risk adjusted rates should equal target ROC.

(3) Compatible accounting system

The requirements placed on the accounting system by DCF analysis are two fold. The system must be capable of collecting and reporting all product-specific investments, costs, and revenues, as well as allocating common selling and G & A costs.

With this capability, DEC can develop a data base of historical costs and returns for each type of investment. This historical data could serve as the basis for the estimation of future costs, improving forecasting techniques, assigning risk adjustments to the discount rates, and improving the tools used in DCF analysis.

(4) Consistent application and assumptions

In order that the results of DCF analysis of multiple projects be directly comparable, it is necessary that cost estimates, volume forecasts, time horizons, and discount rates all be determined in a highly consistent manner.

This suggests that DCF analysis have a well-defined role in the business planning process. This role should be supported with fully-documented, standardized procedures and computerized analytical tools. The administration and maintenance of these procedures,

tools, and product profitability data is a significant responsibility that might best reside in a group chartered and dedicated to review investment analysis under the auspices of corporate finance and/or corporate product management. In addition this group would be responsible for the consistent application, interpretation, and presentation of these analyses to the appropriate committee within DEC.

DCF Analysis for Business Planning and Investment Decisions at DEC

Recently Central Engineering has developed a DCF tool, the Business Review Program (BURP), to aid in the business planning and investment decision processes.

While BURP holds significant promise for improvement of both of these processes, it has emerged prior to the prerequisites (2) - (4) outlined above being met.

Since BURP's intent is primarily to measure relative acceptability of a product business plan, each of the prerequisites outlined above will have to be satisfied before BURP will be fully effective.

In the interim, application of BURP will necessitate continued compromises such as (a) adjusting the discount rate to account for incomplete reporting of fixed asset and (b) making an arbitrary blanket risk adjustment for all new product investments.

DCF Analysis for Tradeoff Decisions at DEC

Part of the charter of the Venus and 11/780 Task Forces has been to develop a process for the analysis of tradeoffs that will lead to maximization of the life cycle profitability of product investments.

This responsibility has lead to the development (in concept) of a Life Cycle Model that employs DCF techniques to determine the financial implications of a design/process tradeoff. This

Life Cycle Model (LCM) is similar in many respects to BURP, but is optimized for analysis of incremental cash flows between the base case product plan and alternate plans that may be proposed.

A major feature of LCM would allow automated sensitivity analysis which would aid in prioritizing multiple alternatives, in terms of effects on a given financial metric, as well as assessing the risks inherent in both the base plan and the alternatives in question.

Being used primarily to analyze incremental cash flows within a program, LCM is somewhat less dependent than BURP on fulfillment of the prerequisites for DCF, however, the analysis LCM performs, implies the use of a discount rate that is not risk adjusted. Based on the following considerations, it is our opinion that the target ROC is the appropriate discount rate to be used for LCM.

Because the alternatives under consideration would generally not require investments beyond those already budgeted for the project, opportunity costs of capital need not be considered in selecting a discount rate.

Since a major application of sensitivity analysis is to assess the risk related to possible errors in estimates of shipment volume, cost, and price, risk adjustment of the discount rate would be redundant in LCM.

Furthermore, the sensitivity analysis approach to risk is preferred since it is based on detailed, product-specific information, while the use of a product-nonspecific risk adjustment assumes average risk which may or may not be appropriate to the particular product in question.

digital

INTEROFFICE MEMORANDUM

TO: Venus Task Force

DATE: 20 JUNE 80
FROM: John Grose
DEPT: New Products
EXT: 231-5265
LOC/MAIL STOP: MR1-3/P74

SUBJECT: MANUFACTURING LEARNING CURVE

When we were analyzing the Manufacturing Learning Curve (LC) as it effects the life cycle model (LCM), we discovered there really wasn't just one LC, but a combination of sub modules which could impact the sensitivity of the overall LCM.

We broke down those areas which a learning curve could be applied to. (Attachment #1) When analyzing the submodels, it was found that throughout the company there was not a wide variety of variations between plants; the same fixed cost would have to be spent. The trade offs are being made, socket vs. non-socket, and as we know more about the system, the availability of Engineering is expected to be an asset with the product being in Marlboro.

We analyzed the projected cost of the System after the 850th system. This cost was based on quoted prices for materials, during the time frame and quantities which Manufacturing would need, and an 85% assembly and test time learning curve for Marlboro. All options outside Marlboro were treated as material alone, because those products would be mature products and their LC and STD would already be established. The break down of the system (attachment 2) shows the bulk of the system transfer cost being made up of 45% for the CPU kernal and 41% for Mass Storage.

Over and above this system transfer cost of $60,766, there would be an additional Manufacturing System charge.

Complex System	(4 weeks)	$14,000
Common System	(2 weeks)	$ 9,000
Dock Mergeable System	(1 week)	$ 3,000

We tried to address which part of the system transfer cost that we could effect the most. It was the CPU Kernal.

In analyzing the CPU Kernal cost of $31,148, we found the breakdown as follows: (attachment #3)

1 MB Memory	9%
Mod. & B/P	61%
Power	11%
Packaging	16%
Unit Assembly & Test	3%

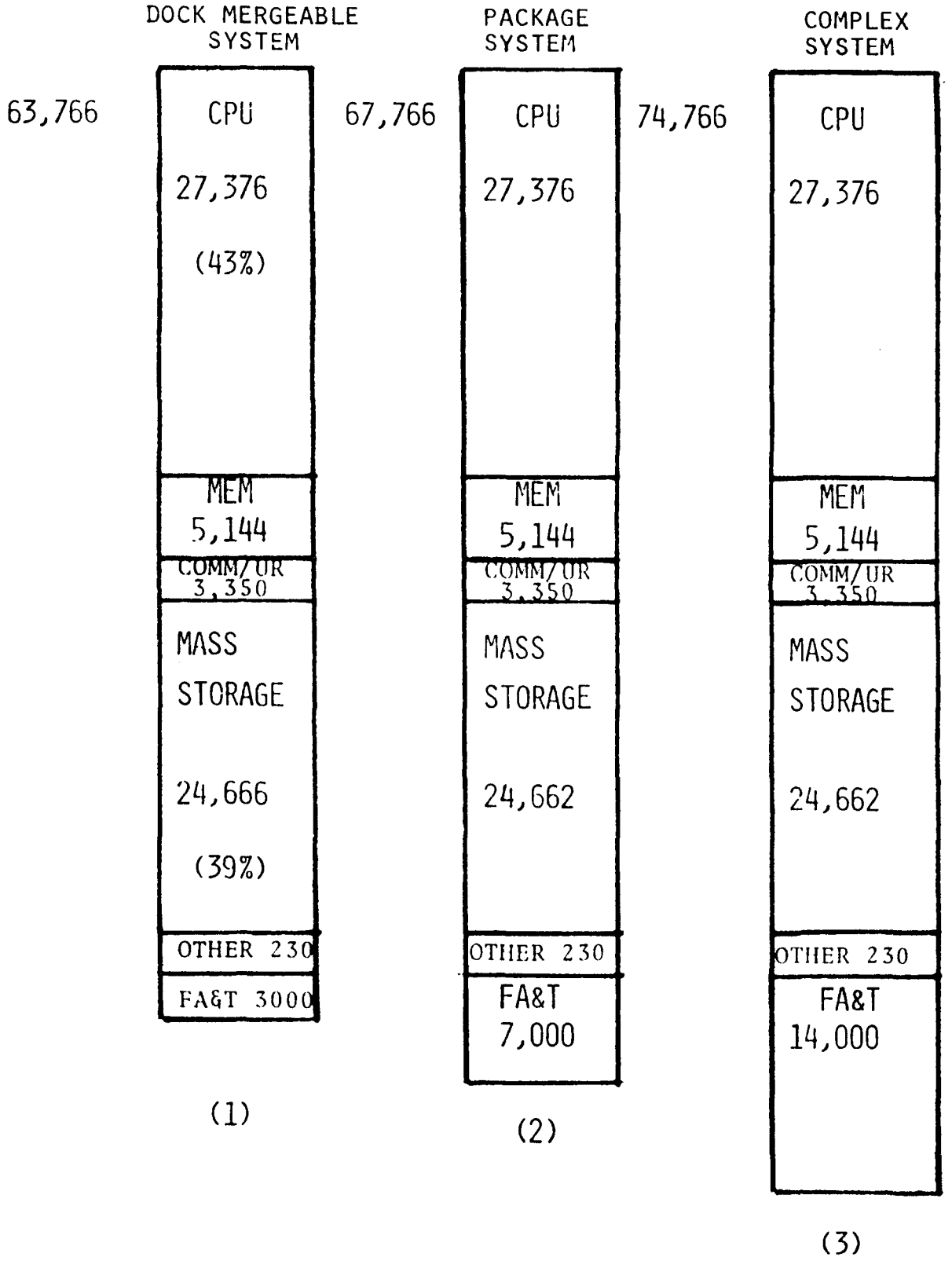
When we addressed the material vs. the Marlboro value added cost, the comparison was 72% to 28%. (Steady State production).

This ratio leads us to come up with the following recommendations:

1 - Purchasing to obtain the lowest possible cost of material (both in-house and outside DEC). (Machine 92% material)

2 - Have a Dock Mergeable System. Potential savings (over life of the product) between Package System and Dock Merge -- $40 million; between Complex System and Package System -- $70 million.

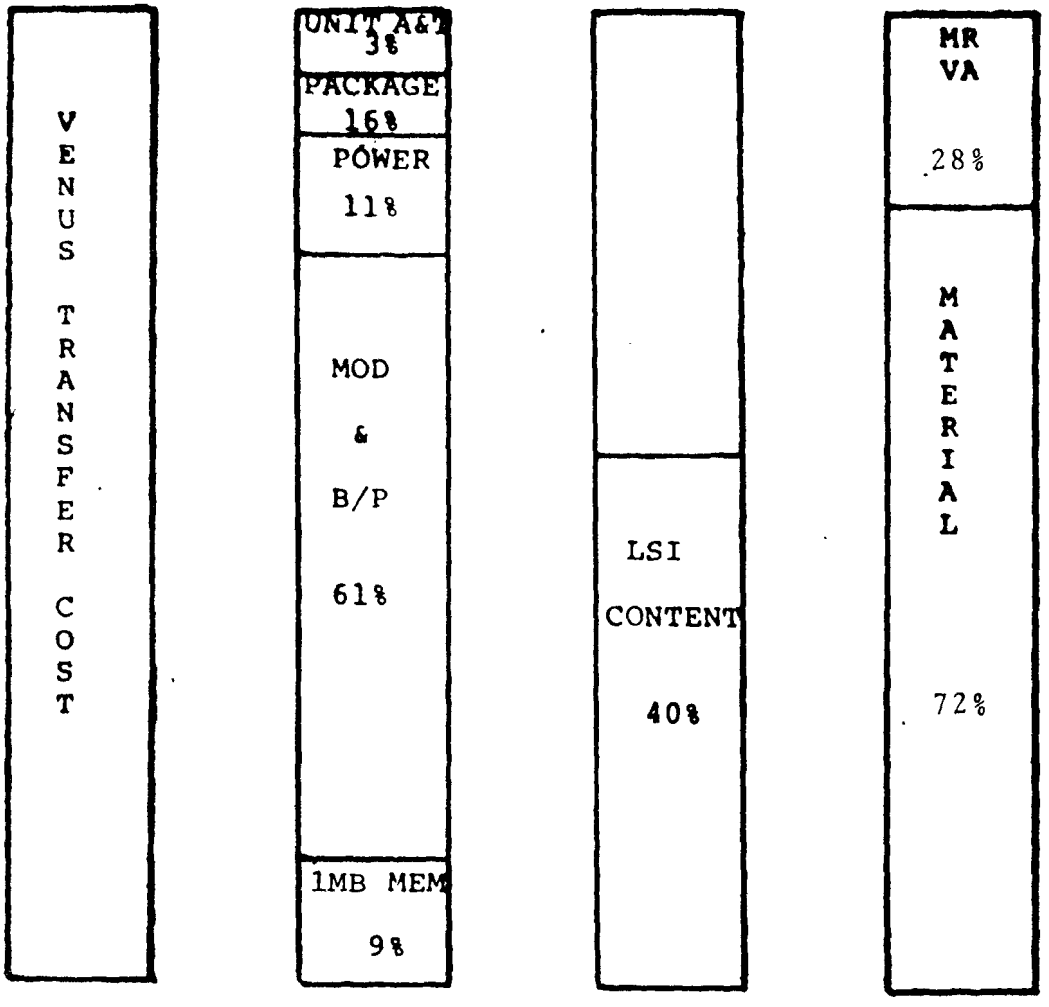
BASIC SYSTEM COST
STEADY STATE



Attachment #3

STEADY STATE
VENUS CONCEPTUAL COST ESTIMATES
COST STRATIFICATION

31148*



*INCLUDES CPU, P.S. CAB, MEM (1MB), CONSOLE, AND SBIA

6-20-80
John Grose

! d i g i t a l !

INTEROFFICE MEMORANDUM

TO:- VENUS TASK FORCE
CC:- MIKE ROBEY
ART O'DONNELL

DATE: 26 JUNE 1980
FROM: REG. BURGESS.
DEPT: LSG CSSE
EXT: 231-4484
LOC/MAIL STOP: MR1-1/S35

SUBJECT:- Report on Venus Task Force Working Group Activities.

The following text is a supplement to the attached slides and should be read with them.

It is essentially an attempt to document the words I normally say during the presentation of these slides.

I will adopt a format which I hope aids the reader by reproducing the lines from the slides in upper case and placing the text between them in lower case.

Something will naturally be lost due to the fact that presentations are a form of two way communication whereas this text is one way. I will try to compensate for this somewhat by writing a little more than I usually say and including the explanations which normally form the answers to the usual questions which arise.

Interested parties are welcome to contact me for a 'live' presentation where question and answer sessions can take place, should they feel that to be desirable.

Reg. Burgess.
26 June 1980.

The following presentation is a separate subject and addresses the Learning Curve phenomenon within Customer Services. Two subjects were chosen to test our ability to perform this analysis, one was Repair Times on Contract Systems, the other was Installation Times.

CUSTOMER SERVICES
LEARNING CURVE

STRATEGY TO GET DOWN LEARNING CURVE FAST IS TO FRONT LOAD THE PROJECT WITH EXPERTISE. OVER TIME, AS TOOLS MATURE, PHASE OVER FROM EXPERTISE INTENSIVE TO TOOLS INTENSIVE MAINTENANCE.

The tools specifically referred to here are:-

SPEAR
TECHNICAL DOCUMENTATION
JOB AIDS
P.M. PROCEDURES
TROUBLE SHOOTING FLOWS
DIAGNOSTICS
TRAINING COURSE CONTENT

We acknowledge that the supporting tools will mature over time with a good feedback plan. Since these tools track the implementation, which often times is unstable even for sometime after the product starts to ship, it is to be expected that they will not be in their fully mature state until they have been put to use for a while (on the stable product) in the field environment. We plan to do the best job we can to accelerate the maturity of these tools, part of our strategy to achieve this will be to establish a strong support base staffed by engineers capable of participating in the process of evaluating the supporting tools and providing the required feedback. This support base also enables us to provide the same kind of feedback on the product itself, we acknowledge that we are also in the design support business and will play at least a part in correcting design shortcomings.

Reg. Burgess.
26 June 1980.

CUSTOMER SERVICES
LEARNING CURVE

USUALLY INSTALLATION HOURS SHOW APPROXIMATELY A 35% - 50% REDUCTION AFTER FIRST YEAR.

This is an area where installation specialists could help to make a larger savings earlier. We plan to explore this area for faster cost reduction over time.

OVERALL - MTTR IMPROVES.

On the average this is true for the total range of products which we service. The principle reasons follow:-

CONTINUEING IMPROVEMENT IN FIELD EFFICIENCY.

We are continuously seeking ways to improve our efficiency.

CHANGING NATURE OF BUSINESS AT LOW END.

This is represented by the terminals business. Essentially an 'option swap' program is implemented which minimizes the need for high levels of expertize and therefore obviates the need for a lot of training cost while reducing the on site time.

PHASE-OUT OF OLDER PRODUCTS.

As older products with poorer RAMP profiles drop out of our contracts files their effects on our averages are lessened.

Reg. Burgess.
26 June 1980.

CUSTOMER SERVICES
LEARNING CURVE

11/780 CHANGES IN 6 MONTHS

NEW ARCHITECTURE, NOT A REPLACEMENT PRODUCT.

The point must be made here that VENUS, in some service respects at least, is a replacement product for the 11/780. Having already experienced what might be thought of as the 'architectural learning curve' it is unlikely that such great changes will be seen on VENUS.

	Q1 FY 80	Q3 FY80	CHANGE
11/780 SYSTEM			
	(3431 CALLS)	(6618 CALLS)	
MTTR	3.52	2.86	-19.0%
MLH	3.75	3.06	-18.4%
Support	6.5%	7.4%	13.8%

This appears to represent very well our strategy of phasing over from higher levels of expertise with little support to lower levels of expertise with more support.

	Q1 FY 80	Q3 FY80	CHANGE
11/780 KA780			
	(1103 CALLS)	(1958 CALLS)	
MTTR	3.74	3.3	-11.8%
MLH	3.86	3.50	-9.3%
Support	3.2%	6.1%	90.6%

For the processor repairs the above effects are even more marked, however it is noteworthy that although the increase in support is far greater the decrease in MTTR and MLH is far less. Presumably there is a steeper learning curve on some of the peripherals which accounts for this.

Reg. Burgess.
26 June 1980.

CUSTOMER SERVICES
LEARNING CURVE

RPØ6 on 11/78Øs

	Q1 FY 8Ø	Q3 FY8Ø	CHANGE
	(142 CALLS)	(234 CALLS)	
MTTR	5.13	3.8	-25.9%
MLH	5.91	4.19	-29.1%
Support	15.2%	1Ø.2%	-32.9%

As expected the RPØ6 learning curve is very much steeper than the 11/78Ø system or the CPU. Interestingly enough we observe a reduction in support ratio, which seems odd at first. When looking at all RPØ6 data in the same time frame there was very little change in repair times, and the averages were in any case significantly lower. More work remains to be done on this subject but it is my personal opinion that we see here the effect of introducing a glitch in the learning curve of peripherals when we attach them to a system which is unfamiliar to the device specialists who are used to diagnosing them via a different host system.

Reg. Burgess.
26 June 198Ø.

CUSTOMER SERVICES
LEARNING CURVE

INSTALLATION HOURS, 11/780 SYSTEMS

MONTH	QTY	ASSIGNED	UN ASSIGNED	TOTAL	Support
AUG	75	31.1	10.3	41.4	33.1%
SEPT	76	36.9	12.5	49.4	33.9%
OCT	79	31.0	12.0	43.1	38.7%
NOV	82	34.2	13.8	48.0	40.4%
DEC	95	32.7	12.0	44.7	36.7%
JAN	?	?	?	?	?
FEB	121	31.4	10.5	42.0	33.4%
MAR	144	28.3	13.2	41.5	46.6%
APR	154	22.5	11.7	34.2	52.0%

Unfortunately January is missing from this set of data, rather than hide the fact I will highlight it, somehow that month's data was not on the system when this analysis was run.

This appears to show similar trends to the repair time learning curve, a decreasing total with increasing support. The next table smoothes some of the peaks, reduces the amount of data and appears to show the trend more clearly.

3 MONTH MOVING WINDOW

3 MONTHS	QTY	ASSIGNED	UNASSIGNED	TOTAL	Support
AUG-OCT	230	32.98	11.61	44.59	35.2%
SEP-NOV	237	34.00	12.78	46.78	37.6%
OCT-DEC	256	32.66	12.58	45.24	38.5%
NOV-FEB	298	32.58	11.89	44.47	36.5%
DEC-MAR	360	30.50	11.98	42.48	39.3%
FEB-APR	419	27.06	11.87	38.93	43.9%

Again the January data is not present, making the 3-month window look a little strange.

Reg. Burgess.
26 June 1980.

Appendix 6

|d|i|g|i|t|a|l|

Interoffice Memorandum

TO: VENUS TASK FORCE
VENUS TASK FORCE WRK. GRP.

DATE: June 24, 1980
FROM: Carl Gibson
DEPT: LSG VAX Product Mngt.
EXT: 231-6779
LOC/MAIL STOP: MR1-2/E78

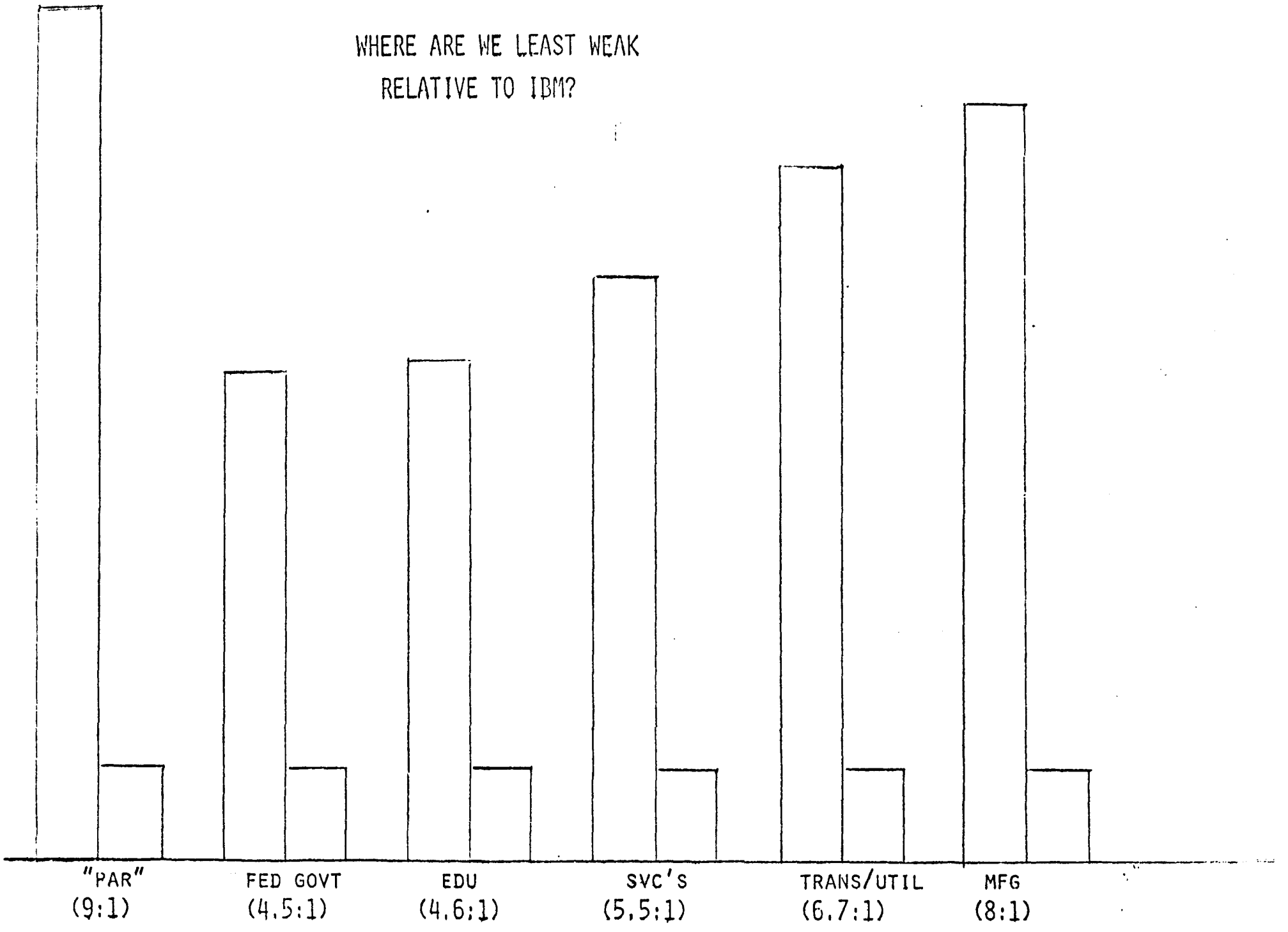
SUBJ: MARKET TARGETTING FOR VENUS

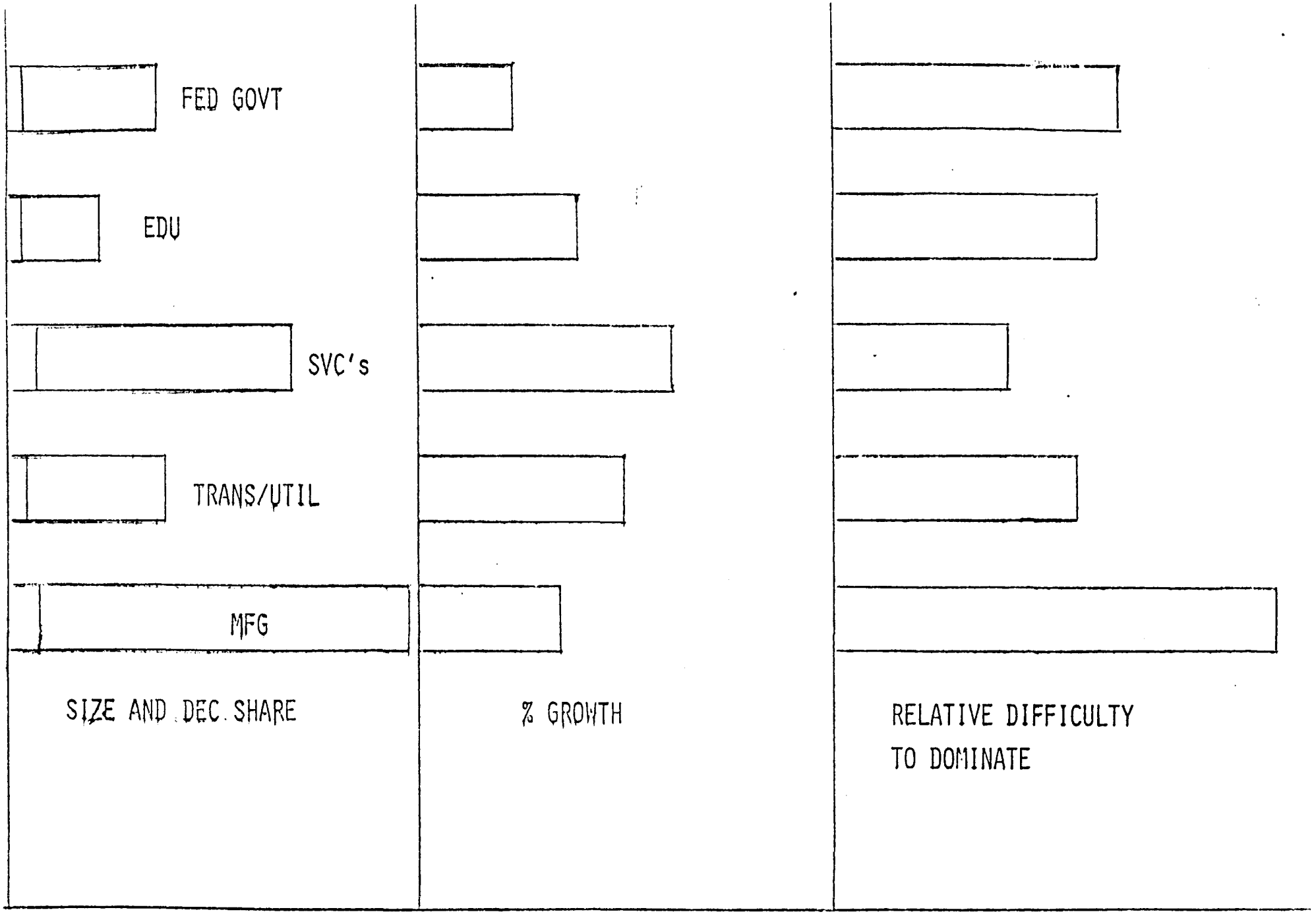
Taking a cue from Boston Consulting Group (BCG), we have examined the likelihood that selective marketing could be relied upon to yield ROI benefits in the VENUS program. BCG's observation that ROI correlates positively with market share stems from the fact that dominant vendors not only enjoy economies of scale in their internal operations, but because of market presence, often take the leadership in establishing pricing and business practices. Another major factor contributing to this phenomenon is a strong posture when acquiring resources; Capital, People, and Material.

In order for any business to exercise real influence in resource markets, it has to dominate the industry as a whole (ie; the Computer Industry), as opposed to an arbitrary subset (ie; Education, or Timesharing). However, deep penetrations of a segmented market does allow the vendor to take leadership in pricing and policy. Therefore, we looked into the major segments of the economy where computers are most commonly found in end use, with an eye to comparing our penetration (measured in $) to IBM's. At par, IBM is 9x our size, but this varies dramatically by market segment. A crude measure of the likelihood that we could displace IBM is to consider how long it would take to equal IBM's penetration if we captured all new business in that segment each year. Even this optimistic approach illustrates that VENUS will have come and gone before we will dominate any market to the level which could allow us to exercise significant policy influence. (See charts)

Yet, targetting markets does have relevance to the VENUS program. Near term profitability can be influenced by targetting markets which DEC services via the most efficient channels of distribution. The scarcest resource for the first 12-24 months of shipments is volume production capacity. Achievable short term profit varies directly with proportionate shipments through various Product Lines. TIG, TOEM and ESG appear to presently be the most efficient users of this resource. As we draw closer to FRS, some shift in the ranking may occur.

In summary, market targetting for VENUS does appear to present opportunities for enhanced program ROI. This potential seems to be due more to channel efficiency than to high market share, at least at the end use level.


WHERE ARE WE LEAST WEAK
RELATIVE TO IBM?
"PAR"
(9:1)
FED GOVT
(4.5:1)
EDU
(4.6:1)
SVC'S
(5,5:1)
TRANS/UTIL
(6.7:1)
MFG
(8:1)



DOES THE MARKET PERMIT US
TO GET THERE ?

CHOOSING CHANNELS

FOR

PROFIT

(IN THE SHORT TERM)

IF A RESOURCE IS SCARCE,

APPLY IT TO THE CHANNELS

THAT ARE MOST EFFICIENT

PROFIT GENERATORS

1ST YEAR VOLUME CAPACITY IS SCARCEST RESOURCE

- WHICH CHANNEL USES IT MOST EFFICIENTLY?

PRODUCT GROUP	PBT / TC *
TIG	.80
TOEM	.71
ESG	.68
MSG	.63
LDP	.61
CSI	.59
COEM	.42
GSG	.42
ECS	.34
MDC	.23
G/A	.05

* PER Q1,Q2 FY80

CONCLUSIONS

MOST LIKELY TO BE ABLE TO BEAT IBM IN

1)	FED GOVT	9.5% OF DEC
2)	EDUCATION	5.8% OF DEC
		15.3%

AND, MAYBE

3)	TRANS/UTIL	10.5% OF DEC
		25.8%

BUT,

- o FED GOVT REQUIRES 355% GROWTH IN A MARKET GROWING 6% / YR.
- o EDUCATION REQUIRES 702% GROWTH IN A MARKET GROWING AT 10% / YR.
- o TRANS/UTIL REQUIRES 740% GROWTH IN A MARKET GROWING AT 13% / YR.

Appendix 7a

|d|i|g|i|t|a|l|

Interoffice Memorandum

TO: VENUS TASK FORCE
VENUS TASK FORCE WRK. GRP.

DATE: June 24, 1980
FROM: Carl Gibson
DEPT: LSG VAX Product Mngt.
EXT: 231-6779
LOC/MAIL STOP: MR1-2/E78

SUBJ: VENUS CONFIGURATION VARIATIONS

Accepting the general principle that a wide variety of system configurations precipitate costs in many areas of the company, we have been seeking ways to:

A) Limit the planned variety of VENUS configurations

B) Develop a methodology and metrics to determine near optimum variety.

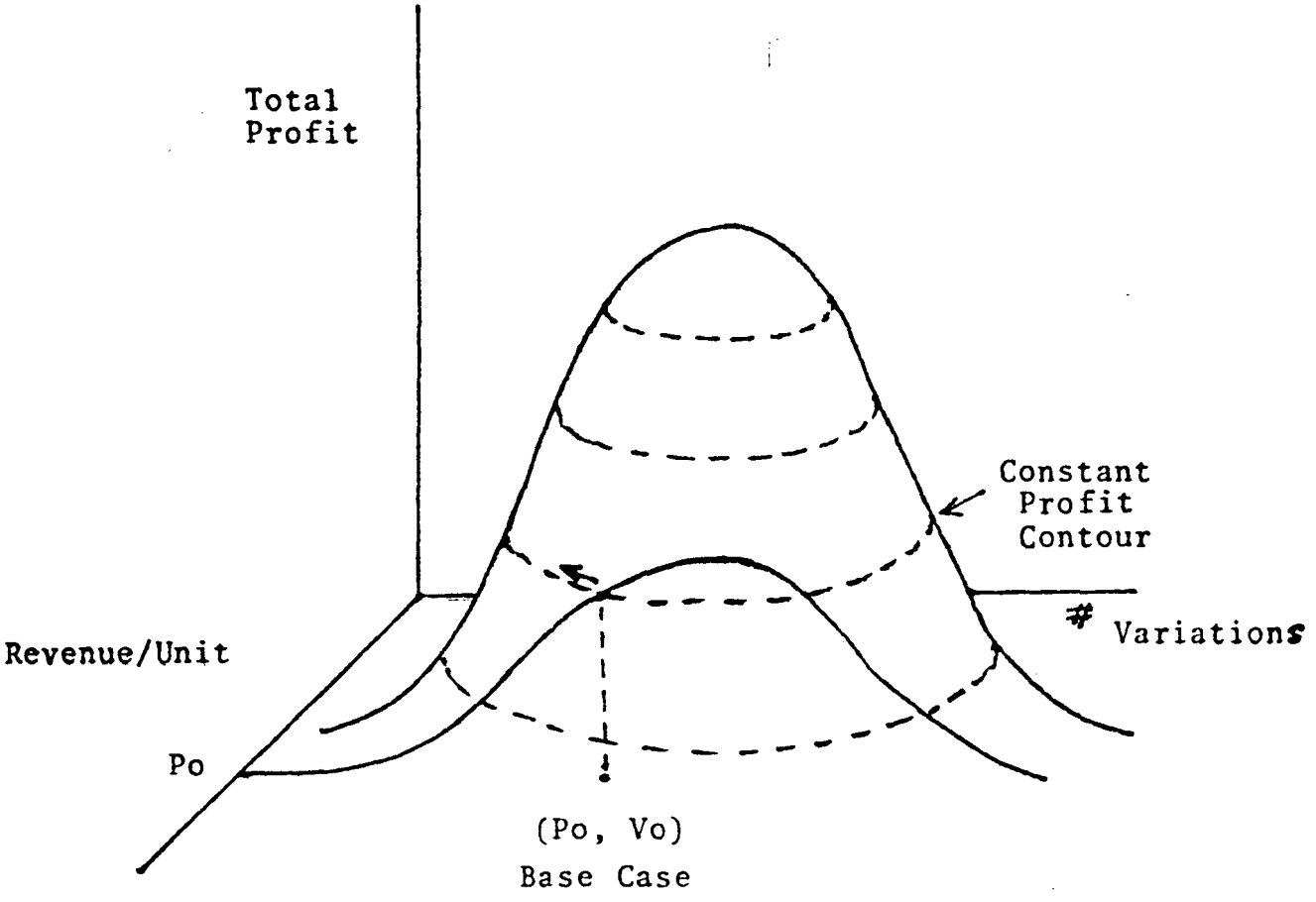
The methodology that has been generated is documented separately by Rolf McClellan (memo of March 21, 1980). We are still in the process of generating realistic (if not empirical) data with which to apply the model.

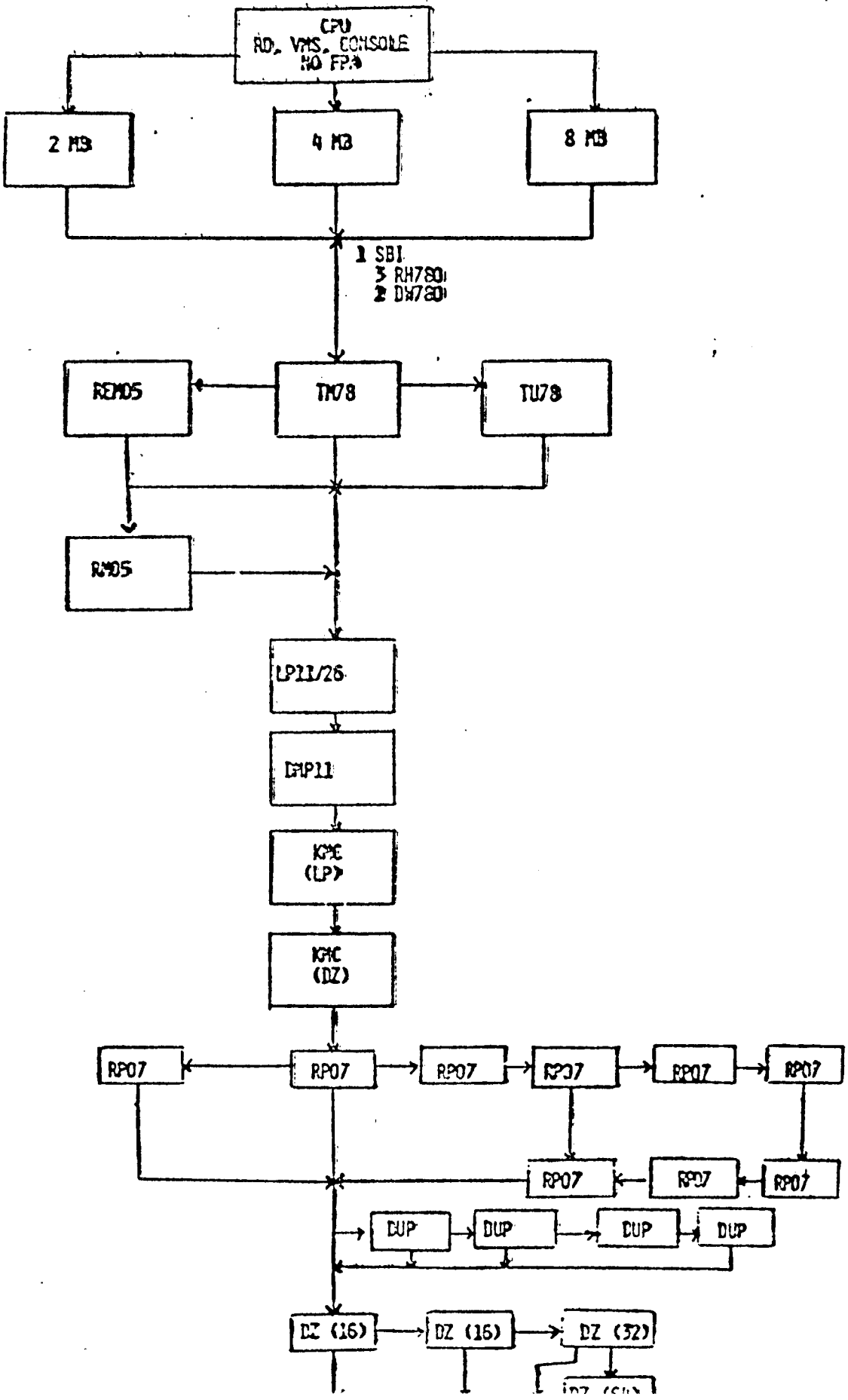
In parallel, the Product Lines and I have taken the step of establishing a baseline set of peripherals and packaged systems for VENUS systems. We have generated:

1) List of supported peripherals
2) List of not supported peripherals
3) Priority of above
4) Set of 4 packaged systems for first year's shipments

These will guide the planning for System Qualification, Dock Merge Planning, and early forecasting. Additionally, this represents the base case against which the impact of change, expansion, or contraction can be assessed.

Future activities are centered around attempting to quantify the tangible benefits that customers associated with flexibility and growth potential. In the high end in particular, generality has value to the purchaser. Pricing that value should be considered as an alternative to cost reduction through specialization.


Total
Profit
Constant
Profit
Contour
Variations
Revenue/Unit
Po
(Po, Vo)
Base Case


CPU
RD, VMS, CONSOLE
NO FP
2 MB
4 MB
8 MB
1 SBI
3 RH780
2 DW780
REM05
TM78
TU78
RM05
LP11/26
DMP11
KMC
(LP)
KMC
(DZ)
RP07
RP07
RP07
RP07
RP07
RP07
RP07
RP07
RP07
DUP
DUP
DUP
DUP
DZ (16)
DZ (16)
DZ (32)

digital

INTEROFFICE MEMORANDUM

TO: VENUS TASK FORCE
WORKING GROUP

CC: Ed Vail

DATE: 21 MAR 1980
FROM: ROLF MCCLELLAN RM
DEPT: Management Science Group
EXT: 223-9162
LOC/MAIL STOP: PK3-2/S53

SUBJECT: AN APPROACH TO ANALYZING THE EFFECT OF RESTRICTING THE NUMBER OF ALLOWABLE VENUS SYSTEM CONFIGURATIONS

Background

A possible partitioning of Venus systems would be as follows:

(1) packaged systems - standard configurations offered at discount prices. Limited add-ons allowed without affecting the discount on the package portion of the system. Probably dock merged. Service contracts possibly included in the package definition.

(2) "à la carte" systems - customer - defined configurations with "sum of the parts" prices. Customer must adhere to a set of configuration rules. Possibly 50% of these systems would be dock merged.

(3) "special" systems - customer - defined systems whose configurations are restricted only by feasibility. In small quantities, the prices of such systems would be likely to exceed the "sum of parts" price. These "Special" systems might be FA&T'd by CSS. Service contract fees would also be likely to exceed "sum of parts" pricing, since diagnostics may not support all options.

The questions we would like to ask are:

"Would life cycle profitability of the Venus program be increased if certain marginal (but feasible) system configurations were disallowed?" (i.e. do total costs of such configurations exceed the corresponding revenue?)

"Assuming that every configuration makes a positive contribution to profit, what price changes would be necessary to offset profits lost through limiting the number of allowable configurations?"

"Assuming that all feasible systems should continue to be made available at an appropriate price, what partitioning of packaged, à la carte, and special systems would maximize the profitability of Venus?"

An analysis of the configurations of existing 11/780 systems used in conjunction with the following process should yield at least partial answers to these complex questions.

A Process for Analyzing Configuration Variation

(1) Definition of Variation

A key first step in this process is to define what constitutes a distinct variation in system configuration. In making this definition, we should focus on those system characteristics that have an impact on the costs of integrating, testing, and maintaining systems. In other words, if manufacturing, field service, and software services are not affected by the difference between two configurations, then these systems belong to a single "variation".

Bearing this in mind, a possible approach to this problem would be to characterize a variation in terms of CPU kernel type, bus type, number and type of interconnect options, and number and type of peripheral options.

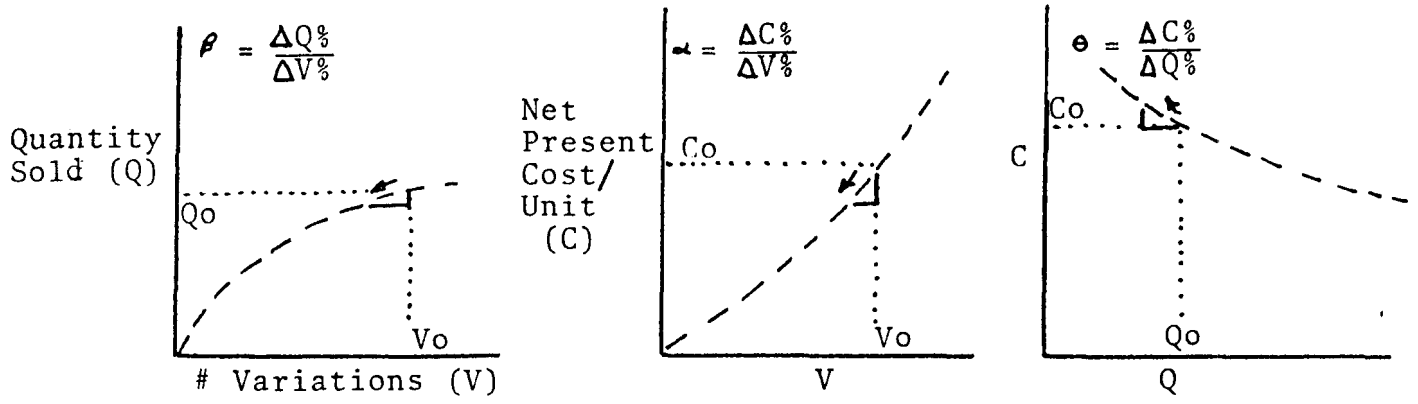
(2) List of Possible Variations

Once the rules of defining a variation have been set, a list of likely variations can be compiled and partitioned among packaged systems, à la carte systems, and special systems. This initial partitioning would become the base case in analyzing the life cycle impact of alternative levels of variation.

(3) Estimation of Elasticities

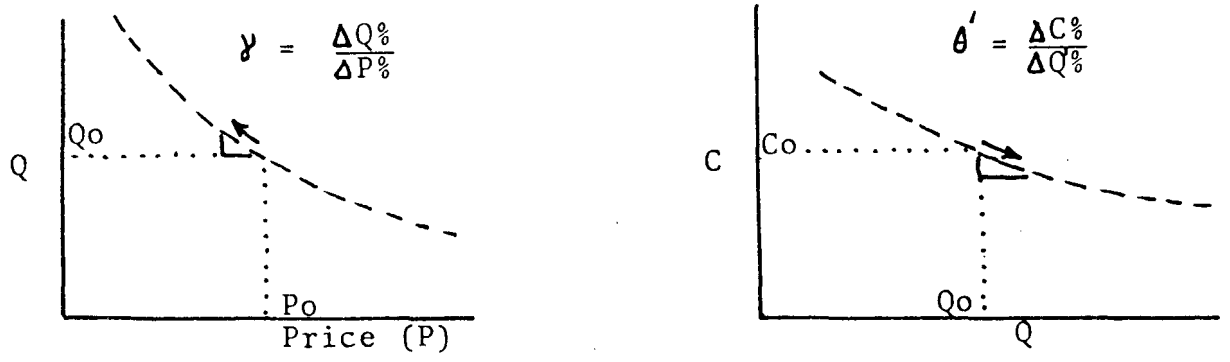
In addition to analysis of the base case, the functional groups should estimate the approximate effects on shipment volume, manufacturing, and service cost of the following changes in the base plan.

Case I (eliminating a specified list of marginal variations)

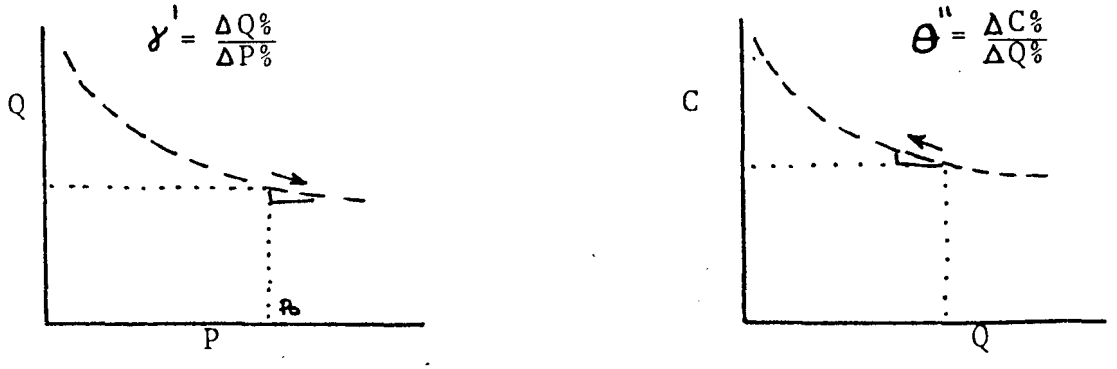


(Qo, Vo, Co, Po defines the base case)
(P is the net present revenue per unit)

Case II (reducing price by a specified relaxation of restrictions on packaged system add ons)



Case III (increasing price by shifting a specified list of variations from à la carte to special category)

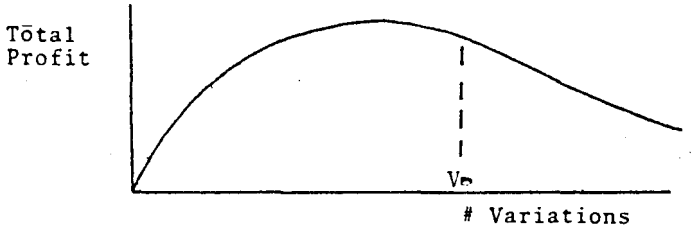


(α, β, θ, etc. can be interpreted as approximate elasticities)

(4) Analytical Results Using Elasticities

Having defined three possible variation scenarios (Cases I, II, III), we would like to use the estimated elasticities to help us decide which scenario offers the best chance for improving Venus profitability and to decide what specific alternatives deserve detailed analysis during the life cycle model.

Case I. Under this scenario, variations should be eliminated if the number of variations in the base case (Vo) lies on the downward sloping portion of the following profit curve.

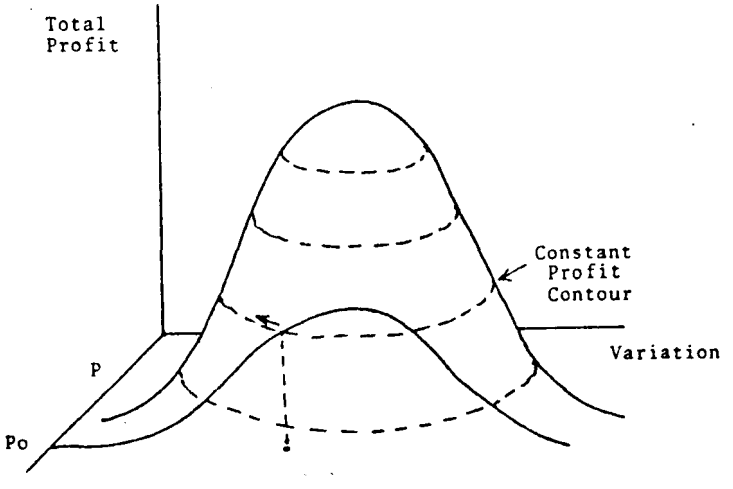


According to an analysis (which assumes constant elasticities), variations should be eliminated if

$$\frac{Po}{Co} < \frac{\alpha'}{\beta} \qquad \alpha' = \alpha + \beta(\theta + 1)$$

where Po = base case net present revenue/unit
Co = base case net present cost/unit

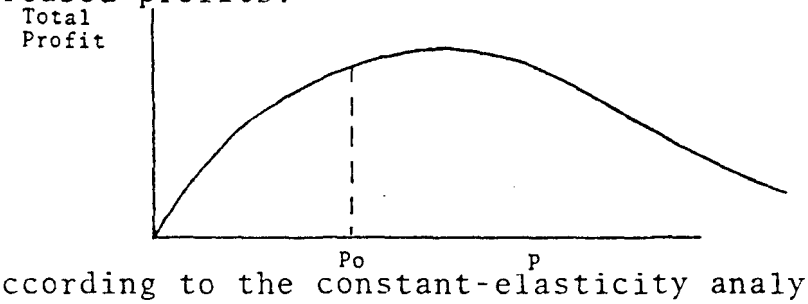
Case II. Under this scenario, eliminating variations reduces total profit. We'd like to know how rapidly price would have to be reduced to offset the profit lost through eliminating variations. In terms of the following diagram, we want to identify price/variation tradeoffs that lie on the constant profit contour.



If we again assume constant elasticities, the analysis shows that the rate of substitution of price (net present revenue/unit) for variations is:

$$\frac{Vo\left[\frac{Co}{Po}\left(\frac{\alpha}{\beta}\theta' + 1\right) - \frac{(\gamma + 1)}{\beta}\right]}{Po - Co\left(\frac{\alpha}{\beta} + \theta + 1\right)}$$

Case III. In this scenario, the demand for marginal à la carte variations is price inelastic. Thus increasing the price charged for such systems has little effect on quantity sold and results in increased profits.



According to the constant-elasticity analysis, the price of such marginal systems should be increased if

$$\frac{Po}{Co} > \frac{\gamma'(\theta'' + 1)}{\gamma' + 1}, \text{ where } 0 > \gamma' > -1$$

(5) Proper Intrepretation of Analytical Results - Narrowing the Field of Possible Alternatives

The analytical results referred to above are based on a number of simplifying assumptions and are therefore, intended to be used simply as guidelines in selecting alternatives to be analyzed in detail with the Life Cycle Model. For example, if the elasticities estimated for Case I indicate clearly that all variations are profitable, we can save ourselves the trouble of detailed analysis of any Case I alternatives.

(6) Analysis of a Limited Number of Alternatives with LCM.

digital

Interoffice Memorandum

TO: VENUS TASK FORCE
VENUS TASK FORCE WRK. GRP.

cc: Ed Grysiewicz

DATE: June 24, 1980
FROM: Carl Gibson
DEPT: LSG VAX Product Mngt.
EXT: 231-6779
LOC/MAIL STOP: MR1-2/E78

SUBJ: CUSTOMER SATISFACTION

A number of tools exist to help us to measure customer satisfaction and arrive at strategies which may influence it.

A) Sales, Software Services and Field Services all conduct regular customer surveys to help manage their operations. These have the advantage of being regular, repeatable, and enjoy a high reponse rate. Their disadvantage are that they address only DEC customers and, being designed as management tools, tend to focus exclusively on a specific area of the customer's relationship with us.

B) Datapro and the trade press also regularly survey customers to assess satisfaction levels. These studies cover a very broad base, summarize large numbers of responses and draw a lot of public attention. They are, however, rarely repeatable, and tend to treat issues at a superficial level. They tend to focus on what Datapro, rather than the customer, thinks is important.

C) Customer Satisfaction Research Institute is a firm which specializes in field research among data processing and telecommunications users. They perform in-depth inquiries with a fairly small, randomly selected, sample of users. The advantages that they offer are; coverage of DEC and competitors, repeatability, relatively detailed breakdown of customer selection criteria and a history of repeat business with our competitors. Disadvantages are small sample sizes used, some disappointment in the literary quality of their reports, and some evidence of tabulating errors. Customer Satisfaction Research Institute has given us some insight into the relative significance that a customer attaches to the various facets of his relationship with the vendor. Furthermore, these reports describe the customer's attitudes concerning selection criteria for future systems.

Our work has, on the whole shown that we attract customers whose interests coincide with our strengths, and IBM attracts customers whose preferences allign with their strengths.

The customer's responses indicate that both firms meet their customer's important needs relatively well. (Our customers appear to be a little too happy with price). Less than complete satisfaction (with a particular facet of a vendor) always results in an increased emphasis of that factor in his next purchase decision. Our present base is reported to be heavily influenced by equipment and software (product components of our relationship) whereas IBM's base takes a lesser interest in products but weighs sales and support service components) up to 6 times as heavily. Both bases claim more product interest in the future. The DEC base has a slightly higher interest in support in the future, but less concern with sales.

The conclusion here is that we should be careful to protect our strengths (product components) due to our customer's heavy emphasis, while making incremental investments in the service components. The trend even in the service areas appears to be closing the gap. Because Customer Satisfaction Research Institute's absolute scores consider the customer's weight of each factor, IBM need make smaller adjustments in their service policies to earn equivalent score increases to that which we might achieve with hurculean efforts directed at a relatively less interested base. One product area which seems to need attention is that of compatibility. This is the only product attribute with which our customers were less than satisfied.

As a note of caution - particular care must be taken with interpreting the Customer Satisfaction Research Institute 1980 Medium SYSTEMS Report. At DEC's request, Customer Satisfaction Research Institute dramatically altered the demography of the survey from 1979 to 1980. Whereas previously the survey had dealt exclusively with 2040's and 2060's. we asked them to include 50% 11/780's in the sample. This had an unanticipated impact upon the mix of applications and customer industries that were surveyed. Any results which compare our 1979-->1980 scores have been biased so unpredictably as to make conclusions about our performance over this period nearly impossible to draw. Making this change has, however, greatly enhanced the utility of this service in the future.

In conclusion, we are still a long way from science in this area, but can say;

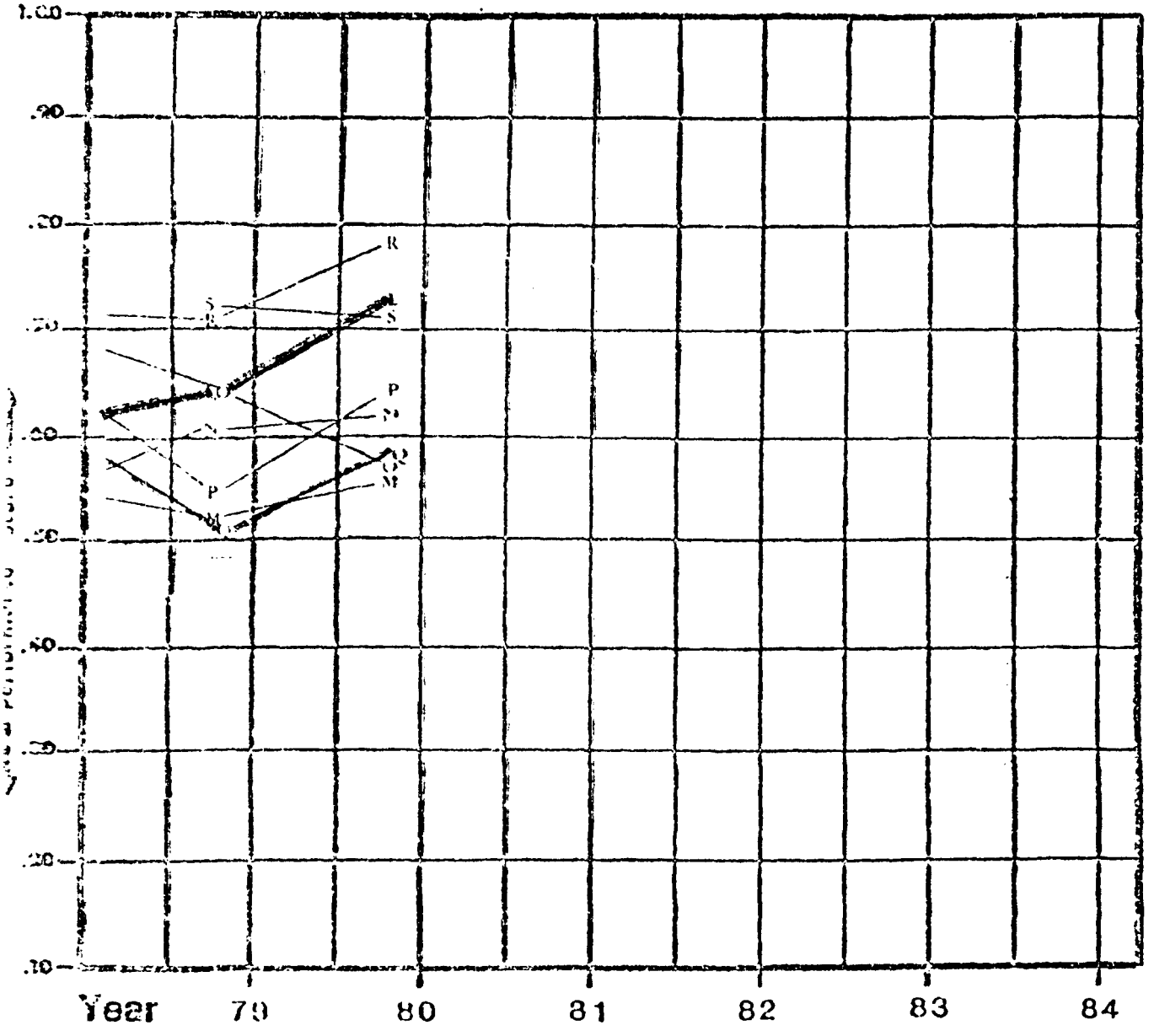
a) If we market to DEC's customers we should strive to retain our product strengths, ask higher prices, and somewhat expand service proficiency in selected areas.

b) If we go after IBM's base, we should place higher emphasis upon service and sales components of the customer relationship.

I believe that the primary benefit to the VENUS program from this inquiry has been as much heightened corporate awareness of customer satisfaction and a better understanding of the customer's preferences beyond features and functions.

OVERALL SCORES ON CUSTOMER SATISFACTION

Chart 1



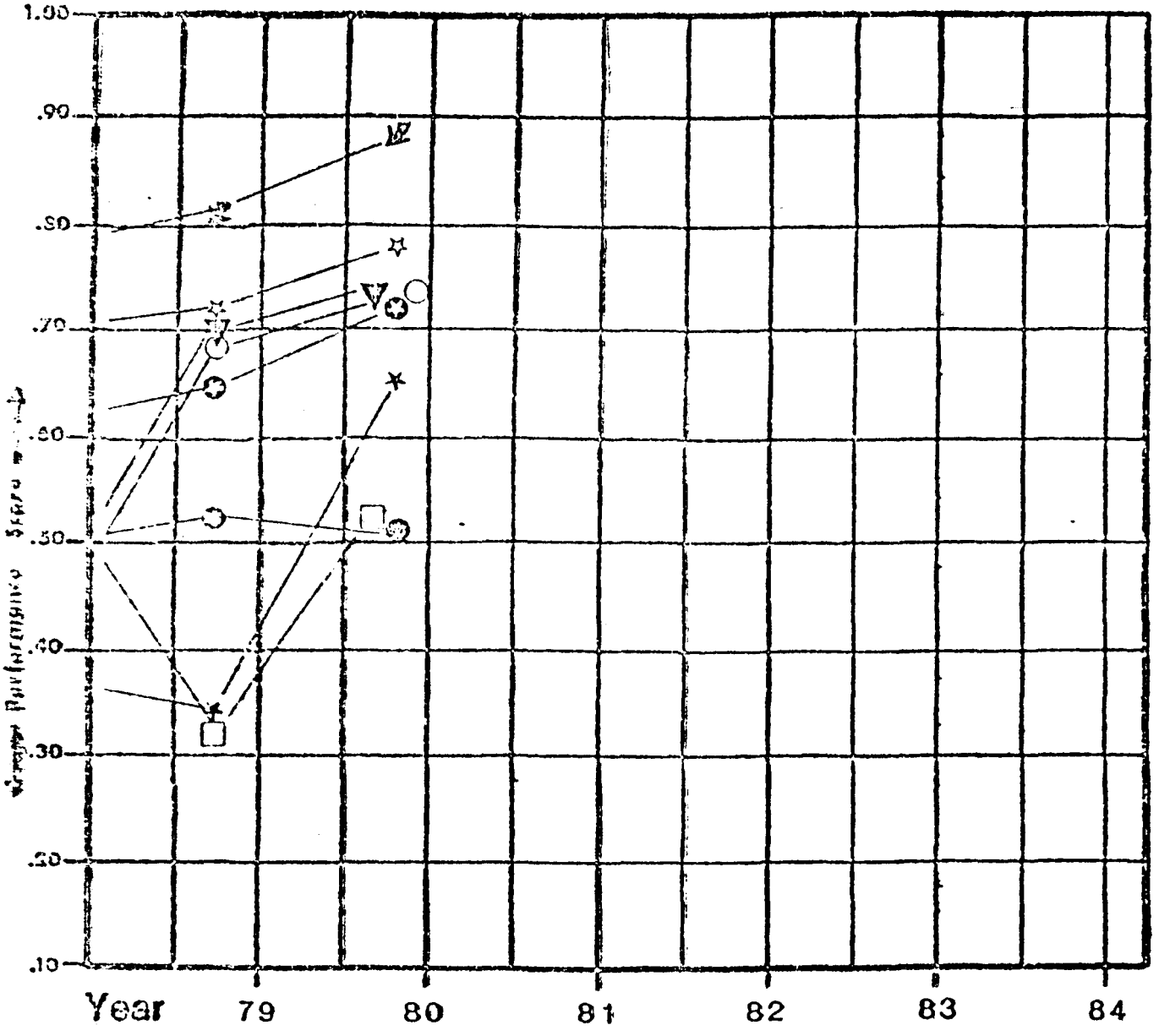
L = IBM

Q = DEC

SUMMARY OF ELEMENTS

OF SATISFACTION WITH "L" — IBM

Chart 9



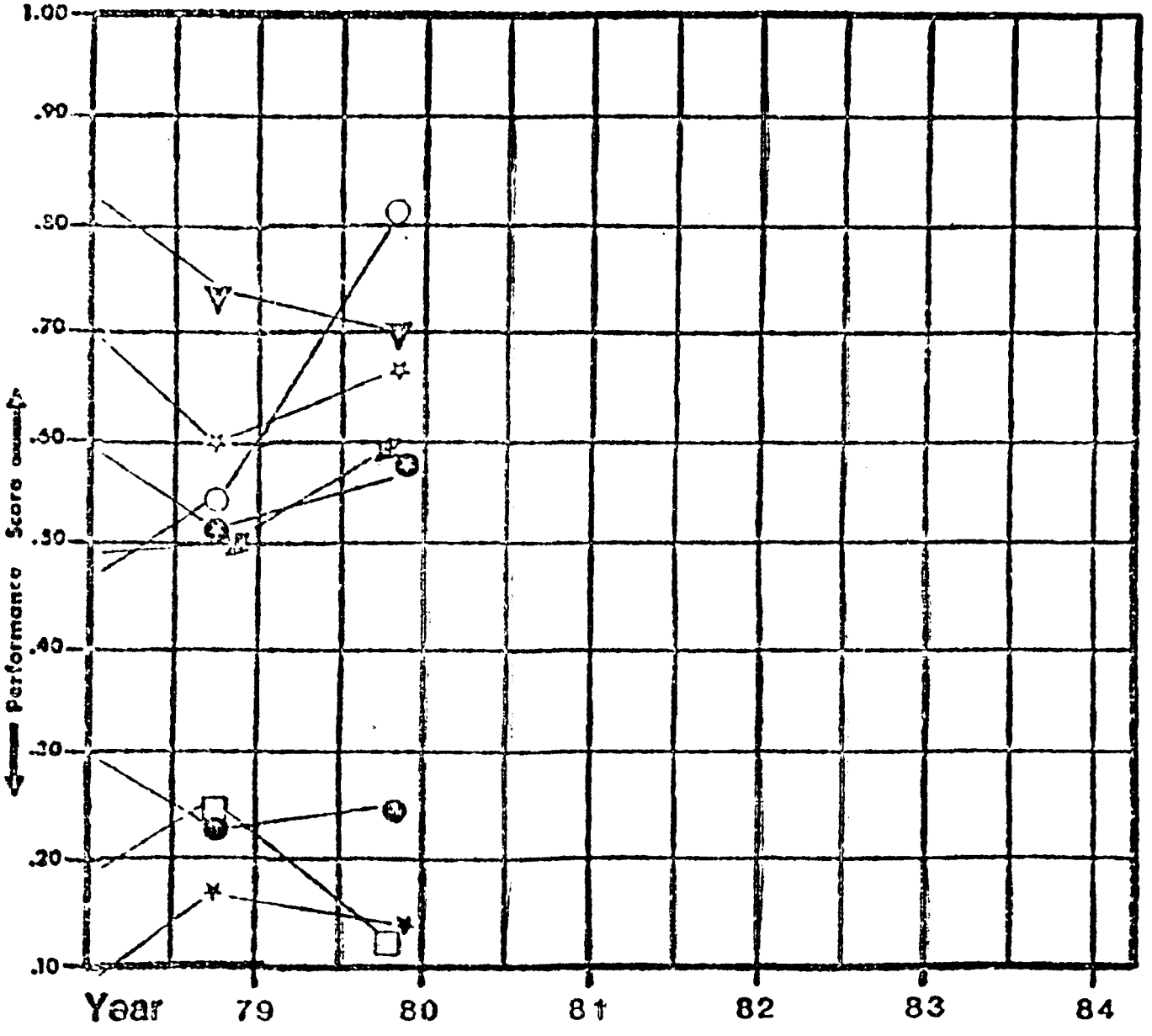
KEY

- = Overall Score
- V = Price
- = Hardware
- = Systems Engineering
- = Software
- = Maintenance
- = Sales Performance
- □ = Conversion

SUMMARY OF ELEMENTS OF SATISFACTION WITH "C"

DEC

Chart 14



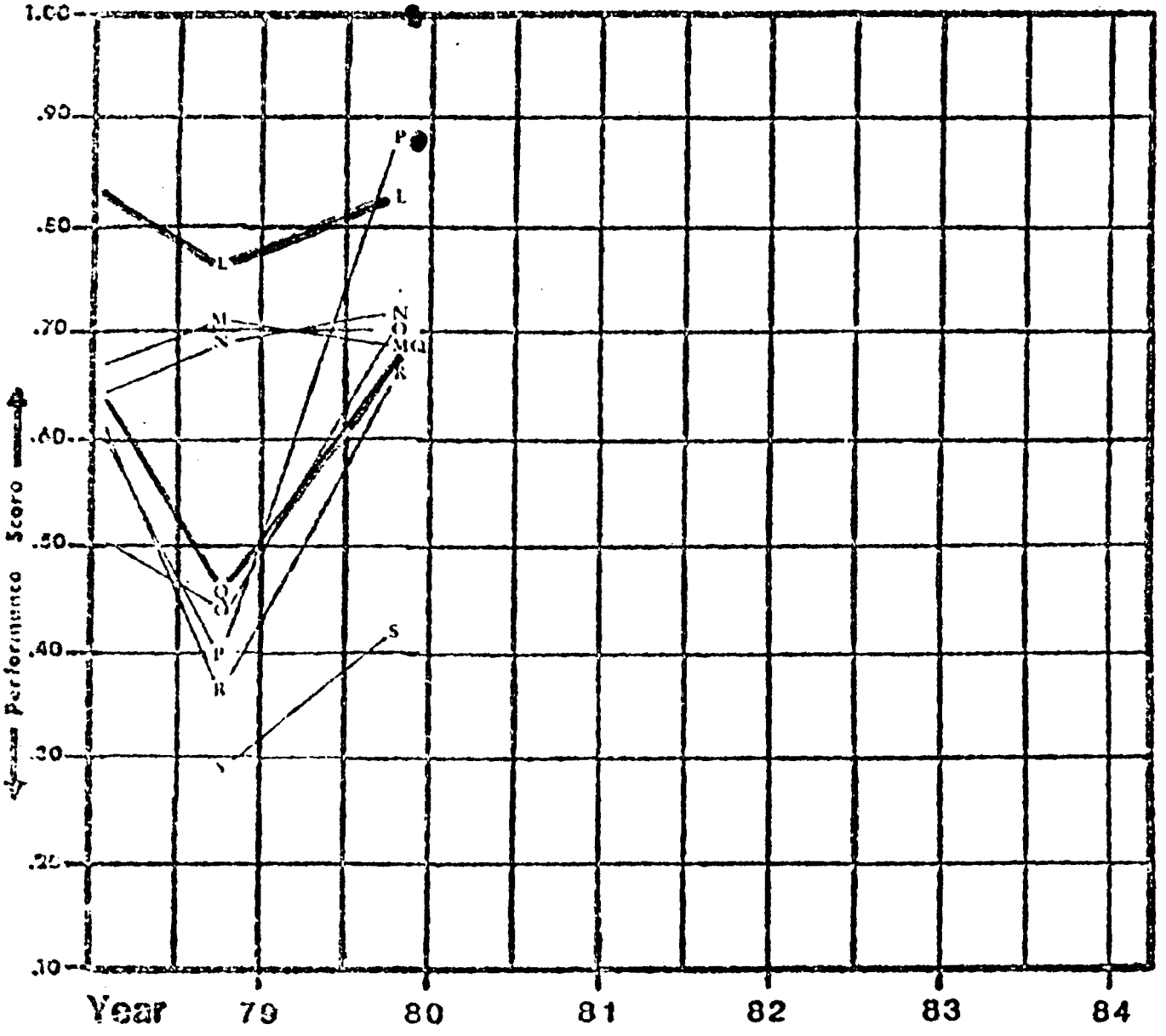
KEY

- = Overall Score
- = Price
- = Hardware
- = Software
- = Maintenance
- = Sales Performance

PREDICTED LOYALTY:

Chart 17

Percentage Of Users Who Would Recommend Their Present Supplier When Ordering A Replacement System



L = IBM

Q = DEC

Appendix 9

|d|i|g|i|t|a|l|

Interoffice Memorandum

TO: VENUS TASK FORCE
VENUS TASK FORCE WRK. GRP.

cc; Per Hjerppe

DATE: June 24, 1980
FROM: Carl Gibson
DEPT: LSG VAX Product Mngt.
EXT: 231-6779
LOC/MAIL STOP: MR1-2/E78

SUBJ: VENUS TASK FORCE AND BUSINESS PLANNING

Basically, business planning serves two purposes:

A) Provide justification for application of resources

B) Serve as a decision-support tool

Historically we have done well at the former, but largely ignored the product business plan as a vehicle to evaluate decisions which impact the business performance of the product.

The VENUS Task Force has provided a forum and a spirit of teamwork which have greatly facilitated the development of a business plan which is a useful business decision support tool.

The VENUS business plan is now sensitive to a much wider variety of life cycle factors than it otherwise might have been. The broad experience base provided by the task force members have enabled us to quantify many aspects of the systems business to a far greater degree than before. Better analytic tools are beginning to emerge thereby making quantative analysis of alternatives a practical reality.

The VENUS Task Force has been working with the Phase Ø Business Plan as a data base for evaluating alternatives during Phase 1. As we reach the end of Phase 1, the updated business plan will be supported by Product Line marketing plans and current estimates by all functional groups.

Particularly noteworthy has been the role that the VENUS Task Force has played in sensitivity analysis. The VENUS Task Force provided a forum where we could model the behavior of the business and test/correct our assumptions about the more subtle impacts of various changes. The slides show some examples of results in this area.

Use of the System Business Plan as a decison support tool has caused us to depart from some business planning traditions which proved cumbersome. In particular, we found that traditional notions of expressing costs as percents of MLP or percents of NOR tended to mask real business behavior. Instead we found that thinking in terms of a transfer cost based model has made product/process tradeoffs more meaningful. Under this concept, each expense line (FA&T, HWW, etc) is examined to determine what value it adds to the manufactured good.

This approach tends to quickly indicate bottom line effect of tradeoffs. However, some life cycle profit opportunities tend to shift revenue/expense patterns in ways that would be inconsistent with our organizational structure and philosophy. Truly maximized life cycle profits may not be feasible given other corporate goals.

COMPANY CONFIDENTIAL

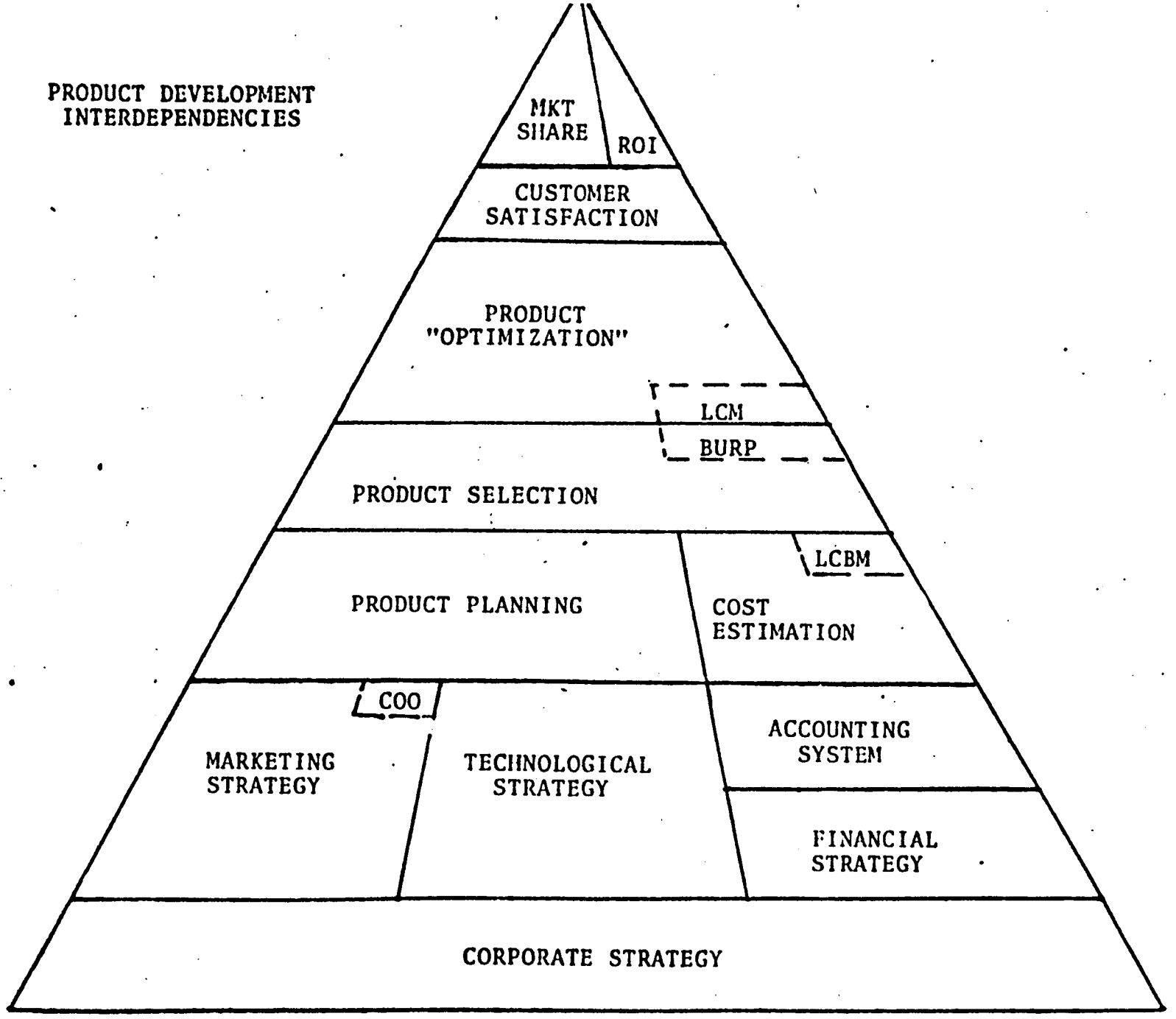
PH/2/17

6/20/80

LIFE CYCLE MODEL

BUSINESS PLANNING PERSPECTIVE

PRODUCT DEVELOPMENT
INTERDEPENDENCIES



LIFE CYCLE MODEL

- VOLUME IS GROWING 30% PER YEAR

- INVESTMENTS IN MANUFACTURING AND CUSTOMER SERVICES BIGGER THAN SYSTEM DEVELOPMENT INVESTMENT

VENUS 32

• DEVELOPMENT	$26M
• CUSTOMER SERVCES	$90 - 100M
• MANUFACTURING - START-UP	$13M
- TOTAL PROGRAM	$750M

LIFE CYCLE MODEL

- MANUFACTURING INVESTMENT OVER 8 TO 10 YEARS

- CUSTOMER SERVICES INVESTMENT OVER 10 TO 12 YEARS

LIFE CYCLE MODEL

- WE HAVE TO START TO ADDRESS TOTAL PROGRAM COST OVER LIFE OF PRODUCT / SYSTEM

- NEED TOOLS / METHODS TO MAKE RIGHT TRADE - OFF DECISIONS TO MINIMIZE LIFE CYCLE COST TO DEC

LIFE CYCLE MODEL

- WE HAVE TO START TO ADDRESS TOTAL PROGRAM COST OVER LIFE OF PRODUCT / SYSTEM

- NEED TOOLS / METHODS TO MAKE RIGHT TRADE - OFF DECISIONS TO MINIMIZE LIFE CYCLE COST TO DEC

LIFE CYCLE MODEL

- BURP FIRST STEP IN RIGHT DIRECTION

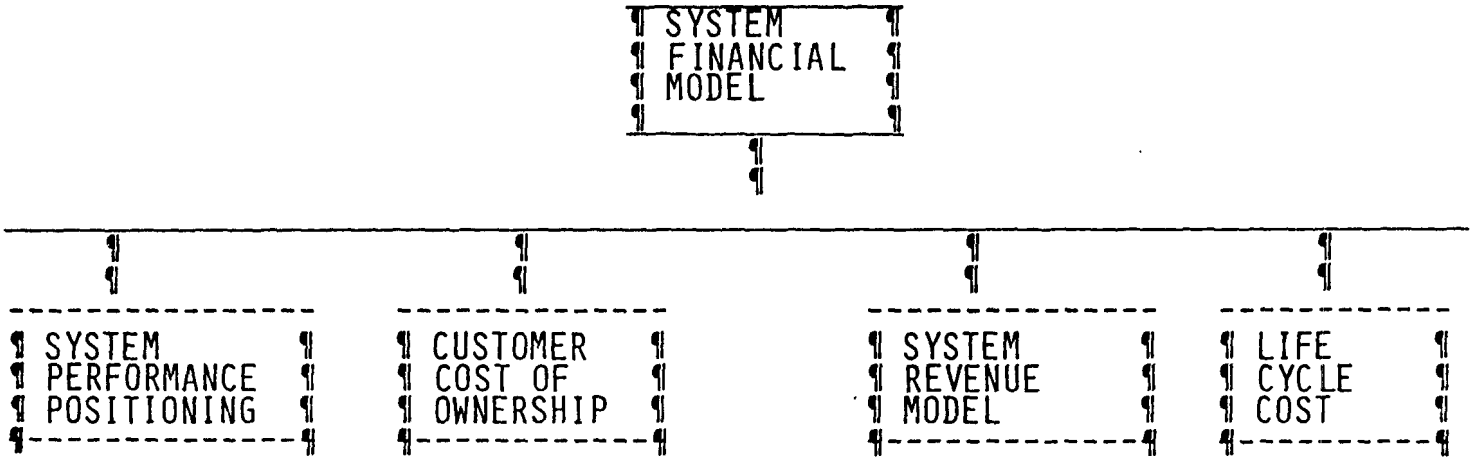
- HAVE TO ADD VENUS LCM CAPABILITY TO BURP
  TO BE ABLE TO DO LIFE CYCLE SENSITIVITY ANALYSIS

COMPANY CONFIDENTIAL

6/20/80

LIFE CYCLE MODEL

- EVENTUALLY HAVE A MODELING TOOL AS FOLLOWS:



- INTERNAL
- COMPETITION

ONE TIME COSTS - 30%
- PURCHASE
- INSTALLATION
- TRAINING
- CONVERSION

RECURRING COSTS - 70%
- OPERATORS
- H/W MAINT.
- S/W MAINT.
- PROGRAM DEVEL
- FACILITIES & OPERATING

COST REVENUE DUE TO DOWN TIME

REVENUE OVER LIFE
- SYSTEM
- ADD-ONS
- S/W
- SERVICES

TOTAL PROGRAM COST
- H/W DEVEL
- S/W DEVEL
- MFG.
- CUSTOMER SERVICES
- MARKETING
- SALES

LIFE CYCLE MODEL

- NEED CLOSER RELATIONSHIP WITH PRODUCT LINE LRP TO UNDERSTAND VOLUME IMPACTS AS WELL AS IMPACTS OF TRADE - OFF DECISIONS BETWEEN FUNCTIONAL GROUPS (ENGINEERING, MANUFACTURING, CUSTOMER SERVICES)

- NEED UNDERSTANDING OF IMPACT OF PRICING IF CAPACITY IS CONSTRAINED

  - PRICING ON MARK - UP

    VS.

  - PRICING FOR MAXIMUM PROFIT

LIFE CYCLE MODEL

BUSINESS PLANNING

WORK DONE ON VENUS LIFE CYCLE MODEL

WILL BE INCLUDED IN NEW BUSINESS PLANS

AND

BURP ENHANCED WHERE NECESSARY

LIFE CYCLE MODEL

FUTURE

- MODEL VERY USEFUL TOOL
- HAVE TO UNDERSTAND IMPLICATIONS ON VARIOUS ORGANIZATION'S GOALS AND MEASUREMENT CRITERIA WHEN IMPLEMENTING LIFE CYCLE MODEL RESULTS
- TEST PROGRAM MIGHT BE NEEDED TO UNDERSTAND EFFECTS

74JA373-0216

digital INTEROFFICE MEMORANDUM

0402

TO: Dragon PSG Members

CC: Gordon Bell
Dick Clayton
Bruce Delagi
Bill Demmer

DATE: October 24, 1974

FROM: Jega Arulpragasam

DEPT: 11 Engineering

EXT: 5545 LOC: ML5/E54

SUBJ: 11/VAX Proposal.

Attached is a proposal for a Linear Extension to 11 Virtual Addressing.

This was drafted by a group consisting of Ron Brender, Mike Garry, Craig Mudge, Dave Nelson, Bill Strecker and myself. It meets the objective of being the best proposal that could be generated by November 1. While it will doubtless be changed (improvements only are permitted!) if ever implemented it brings out all the issues for making a meaningful comparison with Craig's segmented proposal. It also meets, of course, some basic criteria of implementability and performance.

Refinement of performance evaluation is still proceeding and will be reported on at or before the next PSG session.

Jega Arulpragasam

/br

Jega Arulpragasam
October 23, 1974

11/VAX (Linear) Proposal

0403

Summary

1. We define a 28-bit Virtual Address Space.
2. A mapping mechanism is provided for which allows smaller page sizes (1K bytes) but is still compatible with the existing KT. Double mapping is needed to accomplish this.
3. Register extensions are defined to enable extended addressing (and for no other purpose).
4. New instructions are defined for operating on the Register extensions. The set is limited, but both complete and efficient for manipulating addressing entities, including subscripts.
   None of the existing instructions affects the Register extensions, nor are their effects redefined in anyway.
5. Means are provided for distinguishing between 16 and 28 bit entities in memory for use as either "pointers" or "index constants", so that existing programs may run unmodified.
6. Means are also provided for distinguishing between 16 and 28 bit address stacking and unstacking on status switching whether synchronous (Subroutine Calls and Traps) or asynchronous (Interrupts).

The Virtual Address Space.

A 28-bit Virtual Address is formed, in general, in one of three ways:

1. By using the contents of a register R_n concatenated with its extension R_nX. or
2. By picking up a 28-bit entity from Memory. or
3. By picking up a 16-bit entity in Memory and concatenating it with the extension of that particular Register that was used to reference the 16-bit pointer itself.

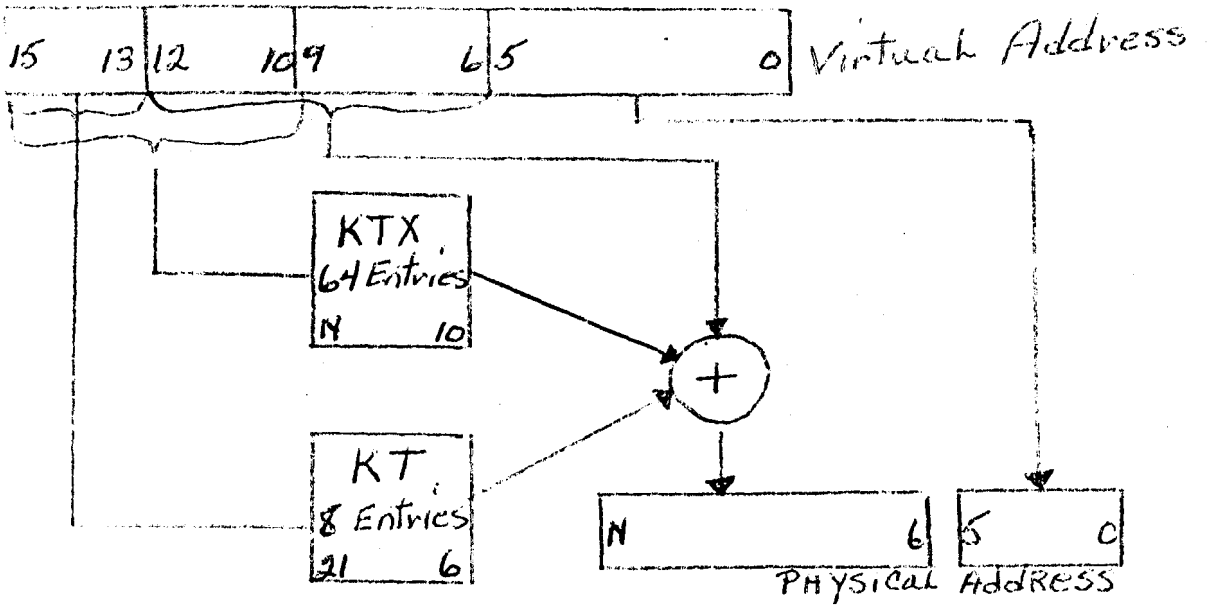
The externally generated addresses are assumed to be extended by leading zeros.

THE MAPPING MECHANISM.

The Virtual Address is conceived of as having three elements. They are a Chapter,a Page and a Displacement. The page, and displacement within the page, together constitute the low 16 bits of the Virtual Address. The high order 12 bits give the Chapter Number.
We define the page size to be optionally compatible with the present KT (i.e. 4K words) or alternatively 512 words.

Linearity of the 4K page with respect to the 1/2K page is effected by double mapping. In order to save serial double accesses to KT tables, and to reduce the context switch time by keeping the number of table entries (per Chapter) small, we define the following traslation mechanism.
The high order 12 bits, or Chapter Number, defines the page tables to be used. For each Chapter Number there are two sets of page tables which are used together. One set has eight entries and corresponds to the present KT tables exactly. The other set has 64 entries, one of which is selected by the high order 6 bits of the low order 16 bits of the Virtual Address. The Address Translation is effected by a 3-way Adder.

The bit alignments are indicated in the diagram.



If we call the 8-entry table the KT table and the 64-entry table the KTX table, it will be noted that provided the entries in the KTX tables were all zeros, the whole mechanism is exactly equivalent to todays KT's in its action.
There is a certain degree of inelegance in the contents of the KTX entries being modified by their own addresses. The KTX can effectively provide the base addresses of 512 word pages in such a manner as to preserve linearity, either this way or with additional hardware to remove this inelegance from the ken of the systems programmer.

Register Extensions and New Instructions.

Reserving the Register Extensions for Addressing entities only has two significant benefits. First, it allows an efficient but small set of instructions to be defined that is complete for Address and subscript manipulation. Secondly, it allows the kind of clean implementation that Craig's scheme permitted, (outlined in the Appendix to his Summary 9/13/74).

The new instrucions are Load Address (LDA), Store Address (STA), Add Address (ADA), Multiply Address (MPA), Subtract Address (SBA) and Compare Address (CPA). They are of the format OP R_1 S_2 S_2. With the obvious exception of the Store Instruction, these Long instructions always have a Register as their destination. Furthermore, the modes permitted for the "Source" are restricted to four: Direct, Deferred, Autoincrement and Indexed. The Index is always assumed to be 28-bits in these cases.
The Multiply is treated as an unsigned 28 x 16 bit operation, optimized for 12 leading zeros in the Multiplier, and yielding a 28-bit result in R_nX/R_n (with overflow indication).

Today in forming the address of A(I,J) we have

```
MOV
MUL
ADD
ASL
ASL
ADD
```

This will be replaced by

```
LDA
MPA
ADA
MPA   (by Data Type length, less than 4 bits of
       Multiplier)
ADA
```

Only these instructions may affect the Register Extensions. Thus, using the existing 11 instructions will cause wrapping around 16 bits, thus maintaining compatibility with existing programs in the unextended architecture.

DISTINCTION BETWEEN LONG AND SHORT ENTITIES.

As in Craig's Chapter Scheme an Extended Mode (X-Mode) is defined when PS <08> = 1.
In X-Mode, the pointers in memory that are implied by all addressing Modes except Ø (Direct), 2(Autoincrement), 4(Autodecrement) and 6(Indexed) may be either 16 or 28 bits.
When they are 16 bits, the Effective Virtual Address is formed by concatenating these 16 bits with the Register Extension that was used to obtain the pointer itself. This case signifies intra-Chapter references and are considered normal. Therefore, the current modes of existing instructions will have this sense.
If we require indexing across a Chapter boundary, or we need 28 bit pointers for inter-Chapter references we define a Mode 5 escape sequence.
In X-Mode (only) the meaning of Mode 5 is redefined. The use of Mode 5 (for either one of the Operand Addresses) implies that the instruction is extended by one or two words and is to reference 28-bit pointers and indexing constants exclusively for that operand address. The leading 4 bits of the one or two word extension define the Addressing Mode.

It is suggested that the 4 New Mode bits have the following bit significance.

1. Increment or Decrement (by the absolute value of the remaining 12 bits of the word).
2. Before or After Use as a Memory Address.
3. Deferred.
4. Indexed.

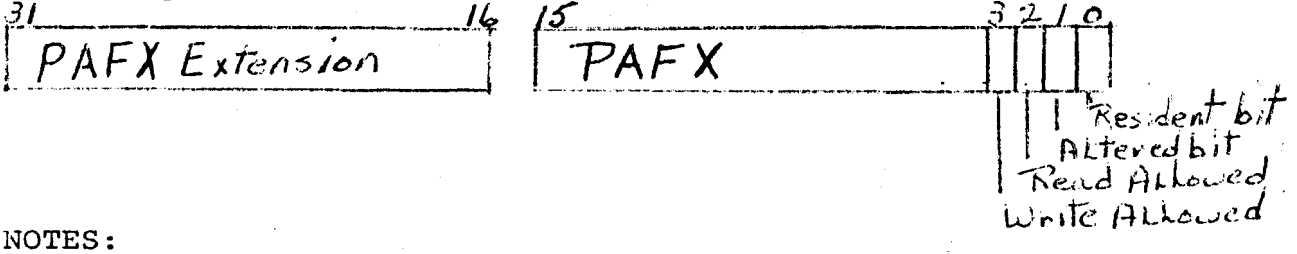
If, and only if the Index Bit is on, is a two rather than one word extension of the instruction implied. Full length indexing capability is essential if linearity is to be provided. In X-Mode too, short indexing is carried out with 16 bit entities that are considered to be 2's complement numbers to permit "negative" indexing.
This is consistent with current definitions, and permits indexing across 32K Virtual Address boundaries with up to 15-bit index constants.
Note also that 28-bit indexing and the use of 28-bit pointers in memory can result in access to a Chapter not defined by any of the currently held Register extensions. Therefore a KTX caching mechanism must be defined to provide an efficient implementation.

STACKING AND UNSTACKING.

No modification to Craig's proposal is required. The mechanism will be identical in all respects with the segmented Chapter scheme in this regard, which does not infringe on linearity at all. This includes the use of PS bits 8 and 9, Subroutine Calling (including JSX) etc.

MEMORY PROTECTION.

On a linear proposal, clearly memory protection has to be provided at the page level.
The protection mechanism will therefore most appropriately be similar to that provided in the present KT.
However, the Page Address Registers for the KTX will be of the following format.



NOTES:

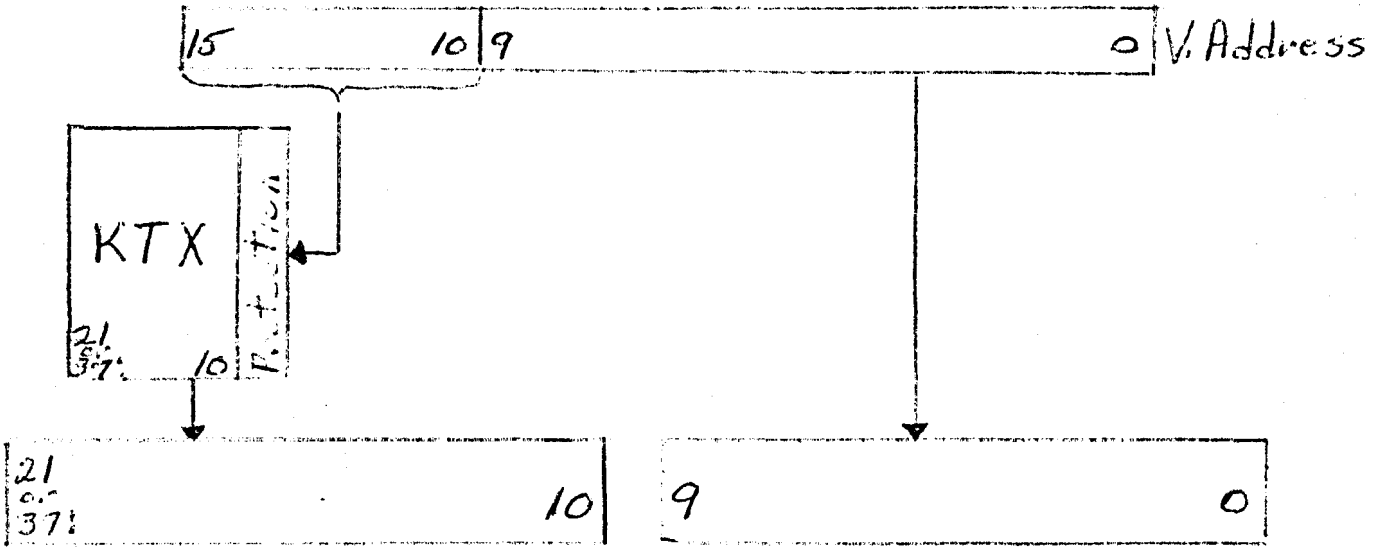
1. Without the PAFX Extension (normal case) the Physical Addressability is 22 bits (12 + 10) or 2 megawords.

2. Resident, Altered (Written Into) and Protection bits are provided to allow this resolution down to 1/2K page size level.
3. Access Rights etc., are therefore determined by the "AND" of those described in the KT entries and KTX entries (except when one or the other is disabled).

4. There will be limited encoding on the Access Rights bits. The "Write Only" case is not useful and will be re-interpreted to indicate that this is a pointer to a Page Table.
5. If an "Execute Only" mode is required further encoding is possible. E.g. Non-Resident AND Altered.

COMMENTARY.

Both the Mapping and Protection mechanism described in this document are fully compatible with the present KT.
However, there is some question as to how important this objective is.
We could gain tremendous simplification by eliminating the current PDR and PAR, if we abandoned KT compatibility.
The KTX Mapping Mechanism would then be as follows.



Note that this is the same as the previous case with the KT disabled.
There is a loss in granularity from 32 words to 512 words.
For this we get simgle word KTX entries (PDR's and PAR's effectively) for 22 bit Physical Addressability.
If granularity is deemed to be important, we could have a length field of up to 10 bits in the high order part of the PAFX Extension. This would give us granularity down to 1 byte at the cost of always needing 2 words for KTX entries, and limiting Physical Addressability to "only" 28 bits. The length field would be compared with the displacement.

digital INTEROFFICE MEMORANDUM

TO: Dragon PSG

CC: Bill Strecker
Craig Mudge
Dave Nelson
Ron Brender

DATE: October 25, 1974

FROM: Bob Gray

DEPT: 11 Engineering

EXT: 3444 LOC: ML5/E54

SUBJ: Considerations in Recommending "Linear".

The 11/VAX subcommittee (Bill Strecker, Ron Brender, Craig Mudge, Mike Garry, Jega Arulpragasam and Dave Nelson), met October 24 to reach concensus on a recommendation between the Segmented VAX scheme proposed earlier by Craig Mudge and a Linear scheme developed by the sub-committee.

The result was a unanimous recommendation for the Linear scheme. The recommendation was offered with the understanding that refinement of the Linear proposal would continue.

In stating their preferences the following considerations were expressed:

Mike Garry: Linear is more favorable "in a practical sense".

Ron Brender: For individual COMMON blocks greater than 32K words, there would be a 5-20 times penalty to try to simulate Linear with a Segmented scheme.

Bill Strecker: FORTRAN drives toward Linear. Also we are unlikely to utilize the superior name space management capability of a Segmented scheme. The Segmented approach is cleaner and simpler to do in hardware.

Dave Nelson: In theory, the Segmented scheme has more potential in the operating system.

Craig Mudge: FORTRAN demands a linear space. We are unlikely to utilize the name space management capability of a Segmented scheme. The Linear scheme is not as efficiently implementable at the low end. Introducing a second addressing architecture adds to uncleanliness. However, increased FORTRAN capability of Linear scheme overrides benefits of Segmented scheme.

Jega Arulpragasam: All VAX schemes based on the PDP11 are unclean. We don't have option to wait for "PDP next". Concerned somewhat whether the Linear scheme is as efficiently implementable at the low end.

In summary the pros and cons of the two approaches are stated on the next page.

PRO	CON
Linear	
FORTRAN demands it.	
Easier to explain at High Level Language.	Harder to explain at assembly language level.
	No name space management.
	New Addressing architecture.
	Less amenable to low-cost (11A05) implementation.
Segmented	
Name space management capability (protection, sharing).	Name space capability unlikely to be supported in DEC software.
No new addressing modes.	FORTRAN COMMON data areas limited to 32K.
Easy to explain at assembly language level.	

Note: Software costs (Mike Garry's Memo 10/3/74) are $680K Linear and $720K Segmented.

/br

digital INTEROFFICE MEMORANDUM

TO: Robin Frith

CC: Distribution

DATE: October 25, 1974

FROM: Craig Mudge

DEPT: 11 Engineering

EXT: 5064 LOC: ML5/E54

SUBJ: 11/VAX - The Removal of the 32K Boundaries.

The segmented version of 11/VAX placed some constraints on the storage of individual large data arrays. These constraints would have been seen by the FORTRAN user as 32K-word limits on individual COMMON areas (Memo 10/8/74: 11/VAX - A User's View of the 32K Boundaries - revision of 9/27/74 memo, Ron Brender and Craig Mudge).

These constraints have been removed by a linear version of 11/VAX. This version was approved by the 11/VAX subcommittee (Jega Arulpragasam, Ron Brender, Mike Garry, Craig Mudge, Dave Nelson and Bill Strecker).

The answers to your questions on 32K boundaries are new revised as follows:

1. Yes, the user can run a sequential program greater than 32K words without explicitly recognizing the 32K boundary.
2. Yes, the user can directly address a data array occupying greater than 32K words.
3. Yes, the user can call multiple subroutines outside a 32K boundary without explicit address manipulation.

In summary the user sees a directly addressable address space of 2^{28} bytes into which he can fit program and data without regard to 32K word boundaries.

The bulk of the segmented 11/VAX proposal, namely those properties which ensure compatibility at both the user program and interrupt structure levels, has been carried over to the linear 11/VAX proposal. Removing the 32K boundary has been done at the cost of 1) some cleanliness in the 11/VAX architecture, and 2) some efficiency in implementation on very small (11A05 - type - cost) machines. However, we firmly believe that it is a worthwhile tradeoff to get the increased FORTRAN capability.

/br

Distribution:

Dragon PSG	Janice Carnes	Bill Strecker
Gordon Bell	Len Hughes	Pete Van Roekens
Ron Brender	John Jones	Larry Wade
John Buckley	Bill McBride	Dave Nelson
		Prod. Line Mgrs.

digital INTEROFFICE MEMORANDUM

TO: Robin Frith

DATE: October 8, 1974

CC: Distribution

FROM: Ron Brender/Craig Mudge

DEPT: Engineering

EXT: 2520/5064 LOC: 3-5/5-5

SUBJ: 11/VAX - a user's view of the 32K boundaries - revision of 9/27/74 Memo.

We have done more work on the implications of a segmented address space for FORTRAN EQUIVALENCE. We have found that the constraints on storage of large data areas are more severe than stated in the Mudge memo of 9/27/74. This follows from the observation that COMMON areas are in fact an implied EQUIVALENCE relationship on data between subroutines.

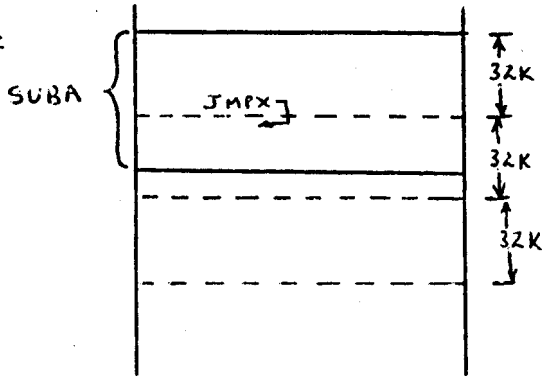
The 32-bit address in 11/VAX is a two-component address (c,d). c, the chapter number, and d, the displacement within chapter, are each 16 bits. 11/VAX purposely treats each 32-bit address as a two-component entity, principally for compatibility with today's 11. Thus the total address space is 2^{16} chapters of 2^{16} bytes each, rather than 2^{32} bytes.

What does this mean to the programmer?

1. "Can the user run a sequential program greater than 32K words without explicitly recognizing the 32K boundary?"

If a routine (subroutine or main program) in a program is 32K then it must explicitly recognize the boundary. It must jump over it. Thus, if we have a 50K subroutine, SUBA, i.e. 50K of instructions, (no data - data are outside of the program chapter), we have

CASE A



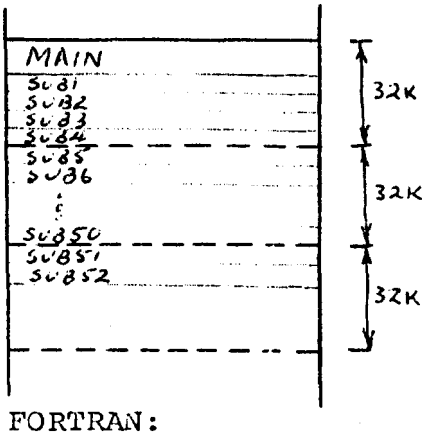
This is the programmer's virtual (or logical) address space.

and there must be a JMPX to perform the interchapter jump.

In practice, however, good programming practice (modularity) excludes this case. Based on data from FORTRAN IV-PLUS, over 5Ø pages of uncommented FORTRAN source statements are required to generate 32K of code.

A programmer writes his program as one main program and many subroutines. Thus his logical address space will be:

CASE B



MACRO and FORTRAN:

Case A: The FORTRAN programmer does not concern himself with the boundary. The compiler will not handle the problem since it is too rare to be worth even understanding how to do. The MACRO programmer must know about the boundary and use JMPX.

Case B: The FTN programmer knows nothing about the boundary. The compiler generates either JSR's or JSRX's.

The MACRO programmer writes JSR for intra-chapter subroutine calls, JSRX for inter-chapter calls.

2. "Can the user program directly address a data array occupying greater than 32K words?"

A FORTRAN compiler would allocate storage across chapter boundaries as needed. The chapter size, however, constrains the maximum size of a dimension. For example, the one-dimension array (a vector) declared in FORTRAN as

```
INTEGER J(10562)
```

would be stored in one chapter. Integer K(100000) could not be. Although a compiler could handle this case transparently, compiler designers would probably choose not to. They would constrain a vector to fit in one chapter. Thus the vector K would have to be split by the programmer into, say,

```
INTEGER K1 (25000)
INTEGER K2 (25000)
INTEGER K3 (25000)
INTEGER K4 (25000)
```

and he would write his program to deal explicitly with the four parts of the vector.

For 16-bit integers and 32-bit floating point numbers the maximum vector sizes would be

0413

```
     INTEGER BIGVI (32768)
and  REAL    BIGVR (16384).
```

The vector,
DOUBLE PRECISION D (3021), a vector of 64-bit entities would be stored in one chapter. The limit would be 8096.

The following variables would be stored together in one chapter

```
     INTEGER K (3021)
     INTEGER I (10)
     REAL    A1(50,100)
```

Implementation strategies for large matrices are considered in two categories: 1) local arrays satisfying certain constraints, and 2) all other arrays.

For arrays local to a program unit, that is arrays that are not in COMMON and which are not passed as an argument in a call (and also are not part of an EQUIVALENCE relationship) there are simple and efficient accessing techniques that would permit such arrays to be stored in multiple chapters.

For example, consider a matrix A which satisfies the constraints and is dimensioned as REAL A(503,3021). It would be stored in 503 chapters, one row per chapter. This storage structure can be exploited in the subscript calculation of the element A (I,J). In a linear space the calculation is,

I x column dimension + J,

giving a one-componenet address. In a segmented space the two component address (c,d) is calculated as

$c_{A(1,1)}$ + I, J

so avoiding a multiplication.
This improvement is estimated to be a 20% reduction in instruction stream words in the inner loop of a matrix multiplication subroutine.

However, most of the time at least one of the above requirements would not be met. In particular, large arrays are in practice almost always declared to be in COMMON for ease of access by multiple subroutines.

The net effect is that most of the time a compiler would be forced to make worst case assumptions about the location and size of any array.

Moreover, the compiled code costs, both in size and performance, for handling arrays which potentially exceed 32K words, is considered to be so high that it would not be acceptable in practice or in the marketplace. Consequently, given a chapter oriented address, the following rule would be imposed on the FORTRAN programmer:

> No single COMMON area, no group of arrays which are EQUIVALENCED together, and no one dimension of a local array may exceed 32K words.

Summarizing the numbers cited earlier, 32K words can contain

32K 16-bit integers
16K 32-bit integers or real numbers
8K 64-bit double precision or complex numbers

We emphasize that this constraint is a major relaxation of the constraint in the existing PDP-11 family where the total of all code and data may not exceed 32K. Without violating the rule above, the programmer would have available as many COMMON areas, equivalence groups, or local arrays as one can conceive of keeping track of, i.e., 32768 chapters worth.

Note that, in practice, for reasons other than machine address space size, a compiler will often put constraints on the maximum size of a dimension. For IBM's PL/I implementations, the maximum subscript on a dimension is 32K. This is because subscripts are held internally as 16-bit signed integers to conserve table space and to exploit the half word instructions of the 360. In Multics PL/I the limit is 24 bits.

3. "If the individual program segment is limited to 32K words, can he call multiple subroutines outside the 32K boundary without any explicit address manipulation"

This is case B under question 1 above, i.e., the compiler or the macro programmer issue JSR and RTS for intra-chapter calls and returns, and JSRX and RTSX for inter-chapter calls and routines.

Summary

1. The FORTRAN programmer would not be aware of the 32K word limit on subroutine size since it would, in practice, be absurd to write a single subroutine that large. He would not be aware of any limit on total code size.

2. The MACRO programmer must be aware of the 32K limit on subroutine size if he is implementing a program that might exceed 32K total.

3. The FORTRAN programmer must be aware of limits on the sizes of individual COMMON areas, equivalence groups, and dimensions of local arrays, but need not be concerned with how they are implemented.

4. The MACRO programmer must be aware of limits of the size of single storage areas, but has available a variety of efficient programming techniques for handling logically coherent very large data areas that can be adapted to the particular application.

/br

Distribution:

Dragon PSC	Janice Carnes	Bill Strecker
Gordon Bell	Len Hughes	Pete VanRoekens
Ron Brender	John Jones	Larry Wade
John Buckley	Bill McBride	Dave Nelson
	Robin Frith	Prod. Line Mgrs.

74JA373-0182

digital INTEROFFICE MEMORANDUM

0415

TO: Distribution

DATE: October 8, 1974

FROM: Jega Arulpragasam

DEPT: 11 Engineering

EXT: 5545 LOC: 5-5

SUBJ: Linear Virtual Space Extension.

OBJECTIVE

The purpose of this document is to present a proposal that meets the preference of Compiler Writers and Marketeers for a Linear Virtual Address space, while capitalizing on the extensive work already done by Craig Mudge on his VAX proposal.

SUMMARY

This proposal is identical in all respects but one with Craig's proposal. That exception is that additional ways in which the Register extensions may be modified are defined, with the sole purpose of removing the watertight logical partitioning of Craig's chapters.

The more obvious limitations (at least) of the linearity thus obtained are identified, and their implications discussed. The potential of these three alternatives is addressed with particular thought to how much of this potential is likely to be realizable in practical terms.

THE BASIC PROPOSAL

First, three things about Craig's proposal are recognized.

A. The ways in which the Register extensions are manipulated are totally independent of the bulk of the work put into Craig's proposal. E.g. Subroutine linkage, Interrupt and Trap procedures and other related "stacking problems".

B. The rigid logical separation of chapters is dependent only on the ways in which the Register Extensions may be modified, and in no way on the other components of the total Chapter Scheme.

C. Craig made the significant decision to effectively limit the use of the Register extensions to addressing entities only. I believe that not compromising or confusing the VAX scheme by trying to devise a general purpose scheme that would facilitate 32-bit arithmetic was a wise one (despite my earlier efforts to push Craig in the opposite direction).

This proposal is different from Craig's Chapter scheme solely in that seven additional ways in which R_nX (Craig's notation) may be modified are defined.

They are;
A. On Autoincrementing) Conditionally on "Carry"
B. On Autodecrementing) out of 16 bits.
C. By the Increment instruction
D. By the Decrement instruction
E. By the Add instruction
F. By the Subtract instruction
and
G. By the Multiply instruction (for subscript handling)

Notes:
A. The instructions can modify only 16 bits unless Destination Mode is zero (i.e. in memory, not more than 16 bits are affected).

B. The instructions do not imply 32-bit Arithmetic. In particular ADD R_m, R_n results in $(R_n)' = (R_m) + (R_n)$ and $(R_nX)' = (R_nX) + C$, where C is the "carry" out of the operation $(R_m) + (R_n)$. R_mX does not participate in the operation.

The other instructions except Multiply act analogously.

The Multiply instruction places the most significant half of the 32-bit product in the extension of the Destination Register (wherever else it may also be placed under current 11 definitions).

C. This set is sufficient for Address manipulation.

D. Since the Multiply in the 11 Instruction set is signed, it is meaningful for subscript handling only if the range of subscripts is limited to 15 bits (in 2-dimensional arrays).

E. Condition Code settings etc. are independent of what happens to the register extensions and are identical with current 11 definitions.

This proposal achieves linearity but has three important limitations.

A. Address manipulation can only take place in Registers,and therefore an additional Store Address (32-bit) instruction needs to be defined.

B. The subscript limitation is more stringent for multi-dimensional Arrays.

The rigorous statement is that while any n-space may have more elements than may be counted in a 16-bit register, every (n-1)-space may have no more elements than may be counted in a 15-bit register.

B. All the limitations of Limited Alternative 1 except for that relating to I-stream bit density still apply.

OUTSTANDING ISSUES

It is not clear to me whether in any of the above 3 proposals, the address located in memory on a Double Indirect such as Autoincrement Deferred should be treated as a 16-bit entity or a 32-bit entity in X-Mode.

I personally favor regarding this as a 32-bit entity, although it would be possible to concatenate a 16-bit entity with the Register Extension that was used to reference it. The latter would limit Autoincrement Deferred to operating with pointers and data in the same chapter. This seems an arbitrary limitation with an inadequate return in savings. However, I would solicit inputs on this question.

This issue arises on all of Addressing Modes 1,3,5 and 7 albeit in X-Mode only (PS<08> = 1), and I trust the same requirement will be imposed on all of them!

IMPLEMENTATION

General

The implementation technique described in the Appendix to Craig's "VAX Summary" of 9/27/74 would no longer be appropriate.

Since R_nX may change and then never be used as an address, changing the KT page tables at the time of R_nX change would be premature. The KT would therefore be loaded when a new value of any R_nX was used to reference memory for the first time.

Such a cache-like scheme would increase the cost of the "KT option" itself.

Also Address translation would take longer when an operand was accessed in a "Chapter" whose KT tables were not in hardware at the time. But it can be seen that the total number of Memory cycles, including those for loading KT tables, is the same as in the segmented Chapter scheme. But these memory cycles will no longer be cleanly collected within the Load Address instruction.

DRAGON Specific

The impact on schedule if it is decided to implement the Basic proposal will be relatively small. While I have not sized it accurately, I would be quite comfortable with one month.

However, Alternatives 1 or 2 impact both the I-Box design and the microcode itself. My guess is that the implication to the schedule would be 3 to 4 months. Hopefully in this case, we would use this time to co-ordinate co-requisite software plans.

C. Displacements appearing in the I-stream, and used in Addressing Modes 6 and 7 are restricted to 16 bits. This is the same limitation as exists in the Chapter scheme, and the extra MOV instruction required to effect A(I)=B(I) appears here too.

This is because the Effective Address will naturally be formed by "High Base Address concatenated with Index in a Register PLUS Low Base Address in the I-stream". Hence two registers are required to hold A-Base and B-Base.

Given only 8 Registers this can make life rather painful.

LIMITED ALTERNATIVE 1

In X-mode i.e. when PS<08>= 1, make the displacements in the I-stream 32 bits.

This removes all the objections springing from limitation C above, at the cost of introducing some of its own:

A. Now we do have 32-bit arithmetic at least in Address formation in Modes 6 or 7, which means putting cost into the Basic machine, contrary to Basic Medium or Small machine philosophy.

B. We pay a penalty on bit density in the I-stream with the concomitant loss in performance implied in additional memory accesses when 16 bits would have sufficed: i.e. most of the time.

C. Note that the subscript limitation still holds unless we permit 32X16 multiplies with further implications to performance and/or cost.

LIMITED ALTERNATIVE 2

It is to be noted that if the 8 present Addressing Modes were to remain as they are, the limitation of bit density in the I-stream would be answered by 2 additional modes indicating "32-bit Index" and "32-bit Index deferred".

These additional modes may be obtained as follows:

In X-Mode, i.e. when PS<08>= 1, Mode 5 is reinterpreted as 32-bit Index". Direct or indirect addressing is specified by one of the 32-bits itself.

Limitations

A. Only 31 bits are available for indexing and presumable for all Virtual Addresses. Indeed only 30 bits would be available if Mode 5 is to be demanded in X-Mode as well. While I do not consider this a real limitation it is listed here for completeness. (In fact, I would propose a 28-bit Virtual Address with a 4-bit Extended Mode).

These schedule guesses are not meant to be commitments, but rather qualitative inputs to help in the decision making process. They are "ballpark" but will be refined if and when 11 Strategy Committee or Dragon PSG decisions demand it.

Jega Arulpragasam.

/br

DISTRIBUTION

Al Avery
Gordon Bell
Frank Bicchieri
Ron Brender
Dick Clayton
Dave Cutler
Jim Davis
Bruce Delagi
Bill Demmer
Robin Frith
Bob Gray
Mike Garry
Wayne Grundy
Brian Fitzgerald
Frank Hassett
Irwin Jacobs
Andy Knowles
Ed Kramer
Tony Lauck
John Levy
Bill Long
Julius Marcus
Mike Mensh
Craig Mudge
Dave Nelson
Bob Misner
Don Street
Bill Planas
John Misialek
Steve Teicher
Brad Vachon
Pete VanRoekens
Garth Wolfendale

Digital INTEROFFICE MEMORANDUM

0420

O: Robin Frith

C: Distribution

DATE: September 27, 1974

FROM: Craig Mudge

DEPT: 11 Engineering

EXT: 5064 LOC: 5/E54

UBJ: 11/VAX - a user's view of the 32K boundaries.

This documents our discussion yesterday on the questions you raised in your memo of 9/20/74.

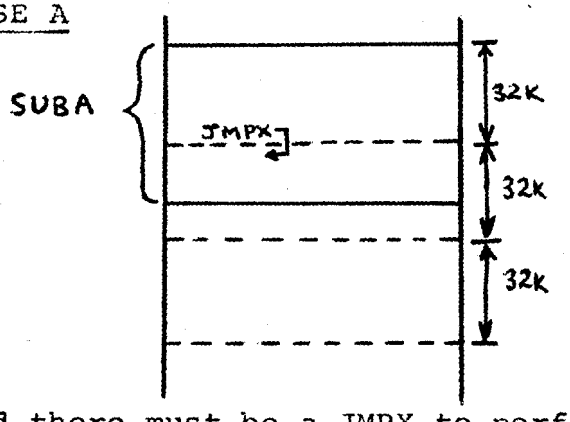
The 32-bit address in 11/VAX is a two-component address (c,d). c, the chapter number, and d, the displacement within chapter, are each 16 bits. 11/VAX purposely treats each 32-bit address as a two-component entity, principally for compatibility with today's 11. Thus the total address space is 2^{16} chapters of 2^{16} bytes each, rather than 2^{32} bytes

What does this mean to the programmer?

1. "Can the user run a sequential program greater than 32K words without explicityly recognizing the 32K boundary?"

If a routine (subroutine or main program) in a program is
> 32K then it must explicitly recognize the boundary. I must jump over it. Thus, if we have a 50K subroutine, SUBA, i.e. 50K of instructions, (no data - data are outside of the program chapter), we have

CASE A



This is the programmer's virtual (or logical) address space.

and there must be a JMPX to perform the interchapter jump.

In practice, however, good programming practice (modularity) excludes this case. A programmer writes his program as one main program and many subroutines. Thus his logical address space will be:

The vector:

DOUBLE PRECISION D (3021), a vector of 64-bit entities would be stored in one chaper. The limit would be 8096.

The following variables would be stored together in one chaper

```
INTEGER  K (3021)
INTEGER  I (10)
REAL     A1(50,100)
```

For a matrix if the toal matrix is over 32K, e.g.,

```
REAL  A2 (503,3021)
```

then it would be stored one row per chapter (503 chapters in this example). (Since FORTRAN stores by columns, it would allocate one chapter per column.)

Note that, in practice, for reasons other than machine address space size,a compiler will often put constraints on the maximum size of a dimension. For IBM's PL/I implementations, the maximum subscript on a dimension is 32K. This is because subscripts are held internally as 16-bit signed integers to conserve table space and to exploit the half word instructions of the 360. In Multics PL/I the limit is 24 bits.

3. "If the individual program segment is limited to 32K words, can he call multiple subroutines outside the 32K boundary without any explicit address manipulation".

This is case B under question 1 above, i.e., the compiler or the macro programmer issue JSR and RTS for intra-chapter calls and returns, and JSRX and RTSX for inter-chapter calls and routines.

Summary

1. The FORTRAN programmer is not aware of the 32K word boundary in data arrays or subroutine length.
2. The MACRO programmer must be aware of the boundary.
3. For point 1 to hold, the FORTRAN compiler must put constraints on the dimension limits on arrays. I feel these constraints are reasonable.
4. Constraints must also be put on FORTRAN EQUIVALENCE statements. I need to do more work with Ron Brender (FORTRAN IV PLUS) to assess the reasonableness of these constraints.

/br

Distribution:

Dragon PSC	Janice Carnes	Bill Strecker
Gordon Bell	Len Hughes	Pete Van Roekens
Ron Brender	John Jones	Larry Wade
John Buckley	Bill McBride	Product Line Managers
		Dave Nelson

7. Interrupt Structure

0423

All trap and interrupt vectors are in the p_0c_0 address space. Kernel mode is implied. The page table for this address space is always loaded in a dedicated part of the RT11 and is selected by the interrupt sequence.

a. Interrups and traps:

11/45	11/VAX
(CM = current mode map)	($M = p_0c_0$ map)
$temp_1 \leftarrow$ PS $temp_2 \leftarrow$ PC PC $\leftarrow$ vector PS $\leftarrow$ vector + 2	$temp_1 \leftarrow$ PCX $temp_2 \leftarrow$ PS $temp_3 \leftarrow$ PC PCX $\leftarrow$ 0 PC $\leftarrow$ vector PS $\leftarrow$ vector + 2
	if PS<09> then $\downarrow(SP)_M \leftarrow temp_1$
$\downarrow(SP)_{CM} \leftarrow temp_1$ $\downarrow(SP)_{CM} \leftarrow temp_2$	$\downarrow(SP)_M \leftarrow temp_2$ $\downarrow(SP)_M \leftarrow temp_3$
or, less rigorously:	
$\downarrow(SP) \leftarrow$ PS	$\downarrow(SP) \leftarrow$ PCX only if the new PS<09> is one
$\downarrow(SP) \leftarrow$ PC PC $\leftarrow$ vector PS $\leftarrow$ vector + 2	$\downarrow(SP) \leftarrow$ PS $\downarrow(SP) \leftarrow$ PC PCX $\leftarrow$ 0 PC $\leftarrow$ vector PS $\leftarrow$ vector + 2

b. RTI and RTT

11/45	11/VAX
PC $\leftarrow (SP)\uparrow$ PS $\leftarrow (SP)\uparrow$	PC $\leftarrow (SP)\uparrow$ PS $\leftarrow (SP)\uparrow$ if PS<09> then PCX $\leftarrow (SP)\uparrow$

Note that the unstacking of PCX on an RTI or RTT is conditional on the setting of PS<09>. This is so that existing code containing "fake RTI's" will run unmodified on 11/VAX. A fake RTI is a common 11 programming technique used to transfer control.

The following sequence of instructions is executed.

```
MOV  NEWPS, - (SP)
MOV  NEWPC, - (SP)
RTI
```

It is "fake" in the sense that the RTI in the sequence has no matching interrupt.

The conditional (according to PS <09>) unstacking during RTI on 11/VAX satisfies the two situations which follow:

a. The RTI is a true return from interrupt

The interrupted process's PCX is on the stack having been put there during the hardware interrupt sequence, and because PS <09> is set it will be unstacked.

b. The RTI is a fake RTI

On 11/VAX this will be an intra-chapter jump, i.e., PCX must remain unchanged and we must unstack no more than PC and PS.

Notice that on 11/VAX, extended fake RTI's are possible:

```
MOV  NEWPCX, - (SP)
MOV  NEWPS , - (SP)
MOV  NEWPC , - (SP)
RTI
```

In the first version of the working notes, PS<08> served to control the conditional unstacking during an RTI as well as its usual function. It cannot do both functions: in the case where the interrupted process is non-X, PS <08> will be 0 and yet a PCX will have been stacked (all interrupts whether interrupting an X or non-X mode program must stack PCX) and must be unstacked.

Warning: the conditional unstacking of PCX depends on the assumption that old programs load a PS with a zero in PS <097. How realistic is this assumption?

digital INTEROFFICE MEMORANDUM

TO: Distribution

DATE: 5/3/74

FROM: Craig Mudge

DEPT: 11 Engineering

EXT: 5064 LOC: 1-2

SUBJ:

Attached are my working notes (as of May 2, 1974) on the chapter scheme for extending the virtual address space.

I would greatly appreciate your comments.

DISTRIBUTION

Jega Arulpragasam
Ron Brender
Dave Cutler
Bruce Delagi
Bill Demmer
Lloyd Dickman
Bob Gray
Tom Hastings
Len Hughes
John Levy
Ed Marison
Dave Rodgers
Bob Stewart
Bill Strecker
Nate Teichholtz

pl

EXTENDING THE PDP-11 VIRTUAL ADDRESS SPACE

- Working Notes on the Chapter Scheme -

Craig Mudge

May 2, 1974

CONTENTS

I. Terminology

11/VX - A machine architecture; PDP-11 architecture with an extended virtual address space using the chapter scheme.

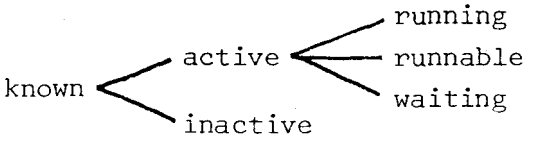
11/44 with KT11X - A particular implementation of 11/VX; the principal design goal of the KT11X option is to provide an extended VAS with minimum impact on the cost of the base 11/44 CPU.

Process - Informal definition: "the execution of a program." The distinction between a process and a program becomes clearer if one thinks about a reentrant program with several concurrent activations.

- Formal definition (Dennis and van Horn): "a locus of control within an instruction sequence. That is, a process is that abstract entity which moves through the instructions of a procedure as the procedure is executed by a processor."

- Task is a synonym used by RSX11-D, RSX11-M, and OS/360.

States of a process:



Known - All processes known to the system.

Active - All processes known to the process manager (or task dispatcher)

Running - A process in control of a CPU. At any one instant in time there is just one running process; it is the process whose context block is loaded in the processor registers.

Runnable - Process able to run but blocked because some higher priority task is running.

Waiting - Blocked awaiting the occurrence of some event; e.g., I/O completion, timer interrupt.

II. Introduction

Why extend the virtual address space?

1. Today's pressure.

   11/45 market pressure leading to supporting I & D space in RSX11-D.

2. Trends which will increase the pressure.

   a. More programming in higher level languages to increase programmer productivity.

      - Compilers, being rich in function, are large programs.

      - Compiled object code is larger than hand code.

   b. Increased physical address space.

   c. Cheaper main memory.

   d. CCD's.

Design philosophy, assumptions.

1. Changing such a fundamental architectural parameter as virtual address length is justifiable from a DEC business point of view, not necessarily from an aesthetic view.

2. The market needs justify extensive design effort and some (optionable) hardware to affect a clean, compatible address space extension. Similarly, the work to generate a clear definition of interfaces to run existing code is justified.

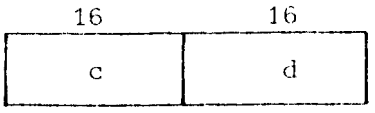
3. The extension should be a big jump (not just an extra bit or two) so that

   a. New uses are possible; e.g., those that follow from true segmentation;
   b. It will survive pressures (memory technology and programming trends) for several years.

III. Summary of the Chapter Scheme

1. A program's VAS is a set of 64K-byte chapters.

2. A VA is of the form



c = chapter number
d = displacemnt within chapter

giving a 32-bit VAS. Today's 11 VAS is 16 bits.

3. The space is a segmented address space; segments (called chapters) are independent.

4. A VA is always mapped to a physical memory address. The physical address space is of the order of 24 bits in the class of machines considered.

5. Each general register, R0-R7, is extended to 32 bits, so exploiting the fact that a register always takes part in an address formation on the 11.

6. New instructions are added to the 11 instruction set to load and store 32-bit addresses and to transfer control between chapters.

7. Address specification is efficient. Full 32-bit addresses will appear in the instruction stream much less frequently than 16-bit addresses, which, in turn, appear much less frequently than 3-bit register addresses (specifying address-holding registers).

8. A CTBR (Chapter Table Base Register) tied to process # facilitates map loading, hence context switch time is improved.

9. The chapter scheme is compatible with today's 11. This has two aspects:

   a. The extensions are compatible

      (i) Existing instructions are not redefined.

      (ii) New instructions are consistent with 11 style.

   b. Code written for today's 11/45 will run unmodified.

10. The compatibility, even at the assembler level, follows from

   a. A mode bit, PS<08>, to distinguish X-mode from non-X mode. Its principal function is to indicate that PCX is the chapter number for any (16-bit) address generated by a non-X mode program.

b. A set of software conventions to be followed when an X-mode program calls a non-X mode program.

c. Existing instructions have not been redefined.

d. The structure placed on a chapter is identical to the KT11 MMU (3 | 7 | 6).

11. Quite general sharing/protection mechanisms (at the chapter level) are possible.

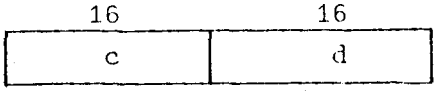
12. Although existing software, both user and system, will run on 11/VX, to exploit its capabilities, e.g., dynamic linking and demand paging, a new executive would be required.

0450

IV. The Chapter Scheme

1. The Address Space

Each process has a 32-bit virtual address space. This space is two dimensional - an address specifies a chapter number and a displacement within a chapter:



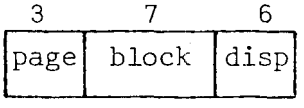
The chapters are independent (a carry out of the displacement field does not propagate into the chapter field).

An address space so structured is usually called a segmented address space. I have used the term chapter instead of segment because DEC documentation has sometimes misused the term segmentation and has sometimes used the terms segment and page interchangeably.

The best known example of a segmented address space is in the Multics system.

The maximum VA on 11/VX is 4096 Mbytes -- 2^{16} chapters of 2^{16} bytes each.

The structure on a chapter is that defined by the KT11 Memory Management Unit, namely



2. Motivation

Denning nicely motivated the concept:

Segmentation

Programmers normally require the ability to group their information into content-related or function-related blocks, and the ability to refer to these blocks by name. Modern computer systems have four objectives, each of which forces the system to provide the programmer with means of handling the named blocks of his address space:

- *Program modularity.* Each program module constitutes a named block which is subject to recompilation and change at any time.
- *Varying data structures.* The size of certain data structures (e.g. stacks) may vary during use, and it may be necessary to assign each such structure to its own, variable size block.
- *Protection.* Program modules must be protected against unauthorized access.
- *Sharing.* Programmer A may wish to borrow module S from programmer B, even though S occupies addresses which A has already reserved for other purposes.

These four objectives, together with machine independence and list processing, are not peculiar to virtual memory systems. They were fought for in physical storage during the late 1950s [W5]. Dynamic storage allocation, linking and relocatable loaders [M3], relocation and base registers [D11], and now virtual memory, all result from the fight's having been won.

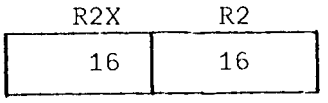
The *segmented address space* achieves these objectives. Address space is regarded as a collection of named *segments*, each being a linear array of addresses. In a segmented address space, the programmer references an information item by a *two-component* address (s, w), in which s is a segment name and w a word name within s. (For example, the address (3, 5) refers to the 5th word in in the 3rd segment.) We shall discuss shortly how the address map must be constructed to implement this.

By allocating each program module to its own segment, a module's name and internal addresses are unaffected by changes in other modules; thus the first two objectives may be satisfied. By associating with each segment certain *access privileges* (e.g. read, write, or instruction-fetch), protection may be enforced. By enabling the same segment to be known in different address spaces under different names, the fourth objective may be satisfied.

ACM CompSurveys Sept 70

3. 11/VX Registers

Each general register is 32 bits wide. The low order 16 bits is called Ri, the high order RiX. Thus, if R2 has been loaded with an address (done by a new instruction, LA, load address),



then R2X holds the chapter number.

Once the extended address has been loaded, then operand addresses are formed through it in the standard 11 way: ( ), ( )+, -( ), @( )+, @-( ), x( ), @x( ). For example, to sum a vector of 16-bit integers, use

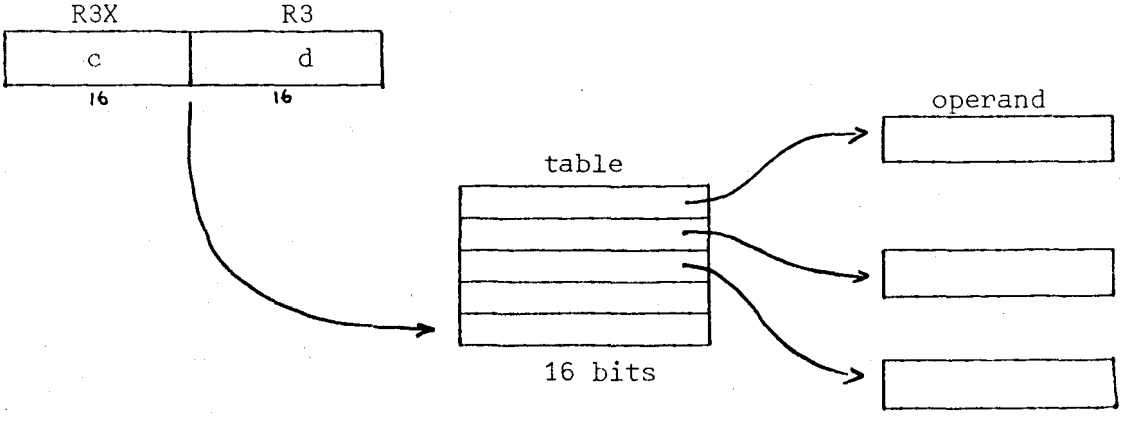
```
    LA  # VEC, R3
┌─► ADD (R3) +, SUM
└── SOB
```

The standard 11 code for this is

```
    MOV # VEC, R3
┌─► ADD (R3) +, SUM
└── SOB
```

That is, when an address is being loaded by a standard 11 instruction, it is a within-chapter address and fills Ri. RiX holds the current chapter number.

Deferred and index modes are defined to use 16-bit quantities in the address-formation process. For example, @ (R3) +:



If 32-bit quantities were allowed in the table, or 32-bit index constants were allowed, we would face the problem of disambiguating 16 and 32-bit quantities in memory. Constraining these quantities to 16 bits is no loss because the long address is needed only as the base; i.e., in the register.

The program counter PC is, of course, also extended; PCX≡ R7X.

0452

4. Address Mapping

Page tables and chapter tables are stored in memory. Each active process has a process # which locates the first entry of the chapter table for that process. For the running process, this chapter table base is held in the CTBR (Chapter Table Base Register).

See Figures 1 and 2.

CTBR is an 11/VX register whose address is in the I/O page.

5. X-Mode

PS <08> holds the mode bit of the running process. When zero, PCX is used as the chapter number for any (16-bit) address generated by a non-X mode program. It is used to obtain compatibility. An example of its use is given in Section V. below.

6. New Instructions

LA — Load Address

```
LA   SS,  R
```

loads 32-bit address into specified destination register

STA — Store Address

```
STA  R,  DD
```

stores the 32-bit address at the specified destination

JSL — Jump and Stack Link

```
JSL  DD
```

(Interchapter JSR PC with PC implied)

↓(SP) ← PC
↓(SP) ← PCX
PCX ← dest
PC ← dest

RET — Return and Unstack Link

```
RET
```

PCX ← (SP)↑
PC ← (SP)↑

JMPX — Jump Extended

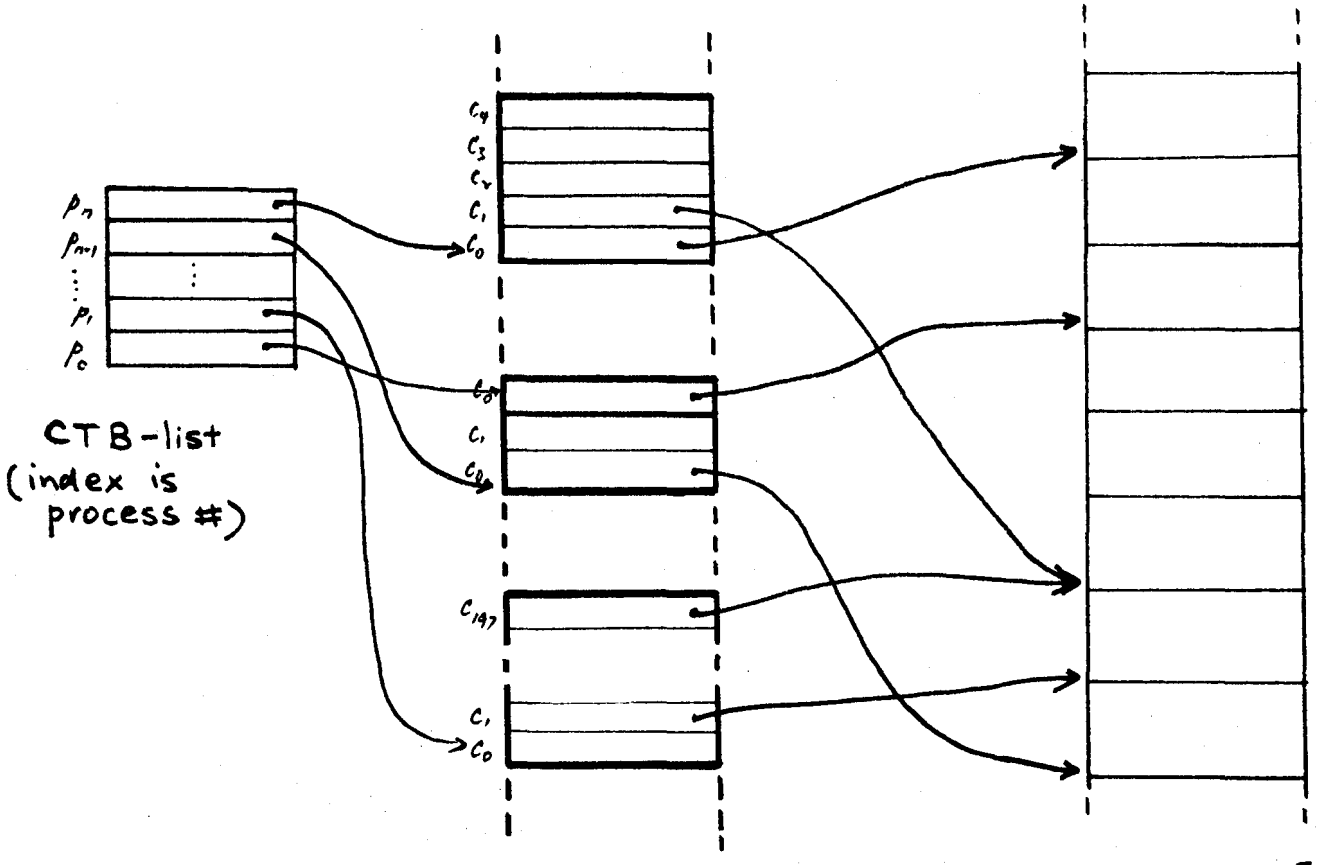
```
JMPX  SS
```

For inter-chapter jumps; an assembler macro which generates

LA SS, R7

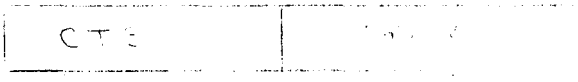
FIG 1

MAP TABLES IN MEMORY



Format of table entries:

CTB-list entry



chapter table entry

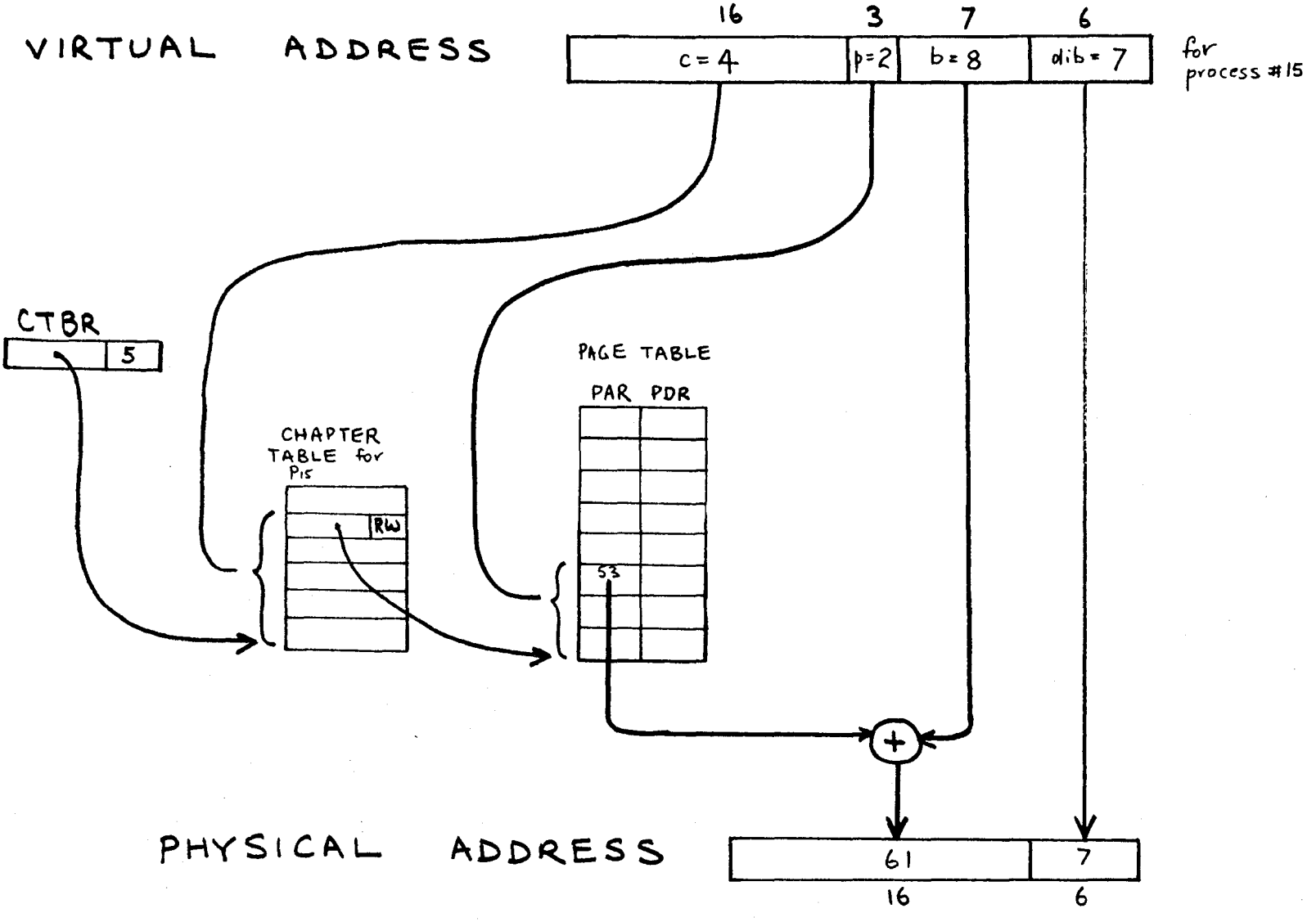
page table entry

FIG 2

ADDRESS TRANSLATION EXAMPLE

0454

32-bit virtual to 22-bit physical



7. Interrupt Structure

All trap and interrupt vectors are in the $p_o c_o$ address space. Kernel mode is implied. The page table for this address space is always loaded in a dedicated part of the KT11 and is selected by the interrupt sequence.

a. Interrupt:

11/45 (CM = current mode map)

$temp_1 \leftarrow PS$
$temp_2 \leftarrow PC$
$PC \leftarrow vector$
$PS \leftarrow vector + 2$

$\downarrow(SP)_{CM} \leftarrow temp_1$
$\downarrow(SP)_{CM} \leftarrow temp_2$

11/VX ($M = p_o c_o$ map)

$temp_1 \leftarrow PCX$
$temp_2 \leftarrow PS$
$temp_3 \leftarrow PC$
$PCX \leftarrow 0$
$PC \leftarrow vector$
$PS \leftarrow vector + 2$

$\downarrow(SP)_M \leftarrow temp_1$
$\downarrow(SP)_M \leftarrow temp_2$
$\downarrow(SP)_M \leftarrow temp_3$

or, less rigorously:

$\downarrow(SP) \leftarrow PS$
$\downarrow(SP) \leftarrow PC$
$PC \leftarrow vector$
$PS \leftarrow vector + 2$

$\downarrow(SP) \leftarrow PCX$
$\downarrow(SP) \leftarrow PS$
$\downarrow(SP) \leftarrow PC$
$PCX \leftarrow 0$
$PC \leftarrow vector$
$PS \leftarrow vector + 2$

b. RTI/RTT:

¬X mode (as determined by current PS)

$PC \leftarrow (SP)\uparrow$
$PS \leftarrow (SP)\uparrow$

Thus, RTI/RTT is an intra-chapter return.

X mode

$PC \leftarrow (SP)\uparrow$
$PS \leftarrow (SP)\uparrow$
$PCX \leftarrow (SP)\uparrow$

8. Process Dispatching and Context Switching

   a. Select p_j, the process to be run next

   b. Restore R0-R5 for p_j
      Restore R6
      Restore FPP

   c. Load CTBR with CTB(p_j)

   d. Stack PCX(p_j)
      Stack PS (p_j)
      Stack PC (p_j)

   e. RTI

Note that this sequence has less instructions than today's RSX11-D sequence; although stage d has one more instruction, stage c has 17 instructions less.

9. New Trap Conditions

   a. Chapter # bounds

   b. Null chapter # (for dynamic linking)

   c. Access control.

V. Compatibility

1. Introduction

Typical problems encountered in extending the virtual address space are (1) specifying extended addresses in instructions, and (2) passing extended addresses between subroutines and between processes. It is necessary to examine where addresses are manipulated as operands - explicitly by program instructions, e.g., (Rn)+, and implicitly, e.g., when addresses are stacked on an interrupt.

These manipulations occur in

.loading an address into a register
.storing an address
.incrementing and decrementing an address in a register
.deferred addressing
.pushing and popping of addresses on the stack:

- instructions
  JSR, RTS, MARK, MTPI, MTPD, MFPI, MFPD

- interrupts
  I/O interrupts
  EMT, TRAP, BPT, IOT,RTI, RTT

As well as the 16-bit length of an address, the structure placed on the 11's 16-bit address space must be considered. Two examples are the KT11 memory management scheme's structure

3	7	6

and the wrap around from 177777 to 0. Other structure is established by users, for example, register-usage conventions in operating systems and compilers.

Because an address space is so fundamental to an architecture, an extension to it must be strictly compatible; the extension must subset to an 11.

11 instructions and behaviour can be classified into:

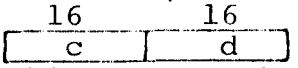
Class A: instructions whose domain is one 32K-word address space, e.g., arithmetic and I/O instructions.

Class B: instructions whose domain is a multiple address space machine, e.g., MFPI.

Behaviour which is K, S, U-mode deriv ed fall into Class B. So also do interrupts - in a multi-chapter address space machine, implementation efficiency demands that some part of the total address space be reserved. For example, process 0, chapter 0 for the interrupt vectors.

2. Class A instructions

These instructions remain unchanged - they specify 16-bit addresses, in particular the displacement field, d, in the full 32-bit space



A program which knows about the existence of RiX is o.k. It issues existing 11 instructions for most of its work and occasionally uses LA to set up a full 32-bit address in a register.

However, a program written for today's machine does not know about RiX. The X-mode bit in the PS takes care of this - it forces the program to run as it was intended, i.e., as a one-chapter program. The remaining question is how does an X-mode program use an existing non-X routine, say SUBNX, to work on data outside of SUBNX's chapter. Appendix A shows how the X-mode bit, together with the general mapping concept, effects this.[1]

[1]Note that the case of non-X calling X is a non-problem - for a non-X program to issue a JSL or JMPX it must know about 32-bit addresses and would no longer be a non-X program.

3. Class B instructions
(This is a topic list only - most problems have been solved but text not yet written.)

a. K,S,U-mode derived address space selection is subsumed by chaptered space structure, except for Kernel space holding interrupt vectors. P_0KC_0 is explicitly recognized; page table always loaded; only place K-notion acknowledged.

b. MFPI and MTPI
supported

c. I and D space
Not supported on 11/VX
Unnatural notion
Very bad decision to put it into 11/45, unclean, bought very little, cost much.
DEC software does not support it.
If we have to put it in, then one way is to use bit0 of the chapter field to indicate it.

d. Questions
K,S,U-mode forces hierarchy on SPL, RESET, HALT
- implications for 11/VX?

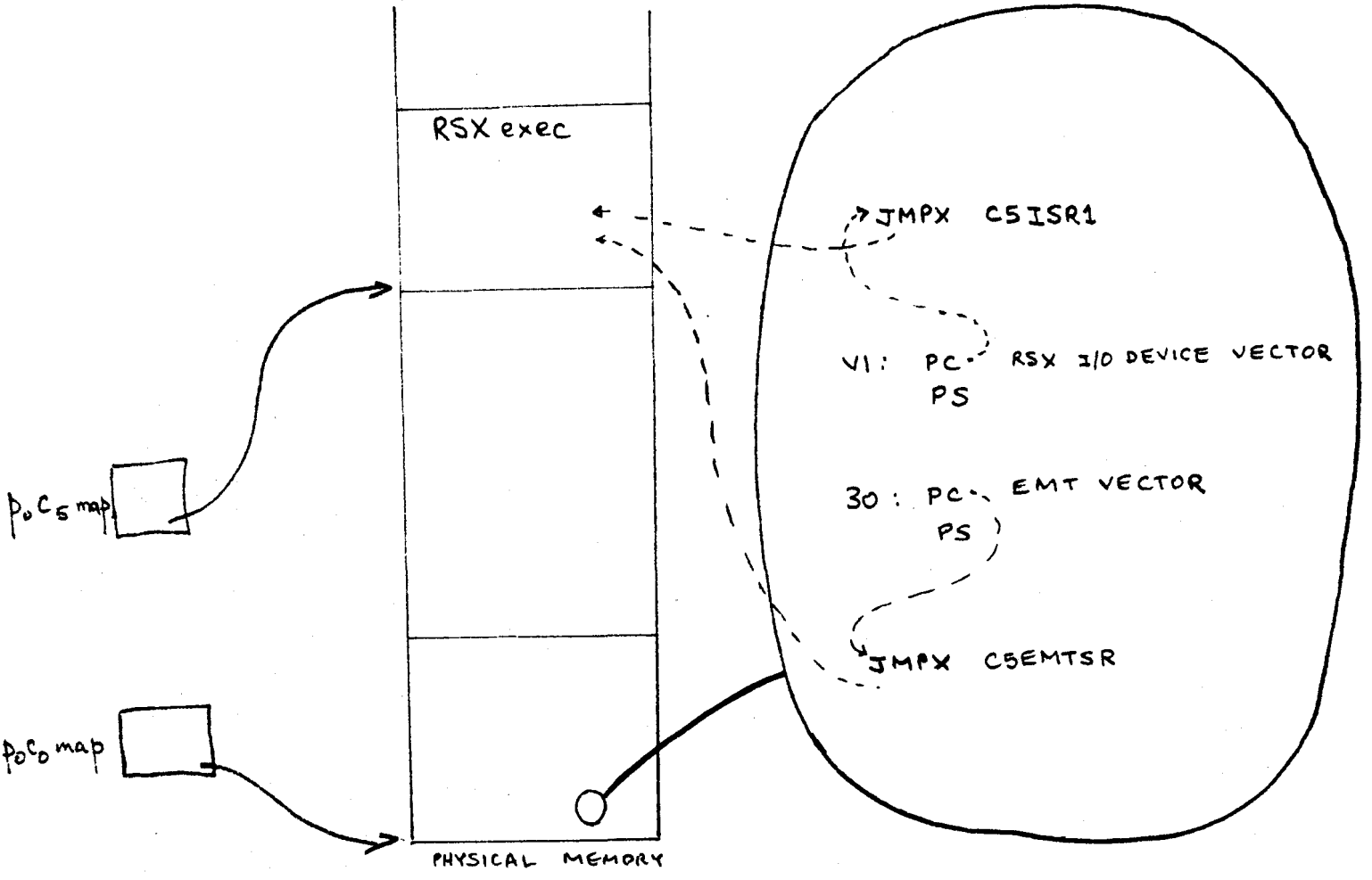
e. Running existing operating systems:—

If RSX11D (say) occupies Chapter 0, then PCX←0 is o.k., but if it is in some other chapter then a nonzero PCX is required to get to the ISR (interrupt service routine) in that chapter. Rather than burden 11/VX with an extended interrupt vector pair, the nonzero chapter will be reached by one level of indirection, i.e., the interrupt vector PC transfers control to a chapter Ø connection section which loads PCX with the right chapter #.

Example 1

RSX11D exec in C_5

0460



Example 2

RSX11D exec in C_5
RSTS exec in C_7
No shared i/o devices simplification (more for resource management)
Sync traps, e.g., EMT, require the connection code in chapter Ø to use process # of interrupted process to decide which opsys to direct interrupt to.
If RSX RSTS need to have access to the same system stack then map them that way.

VI. KT11-X -- an implementation

Figure 3 presents a tentative implementation. It is a starting point; we will modify it as we work on the partitioning of function and registers between the 11/44 basic CPU and the KT11-X.

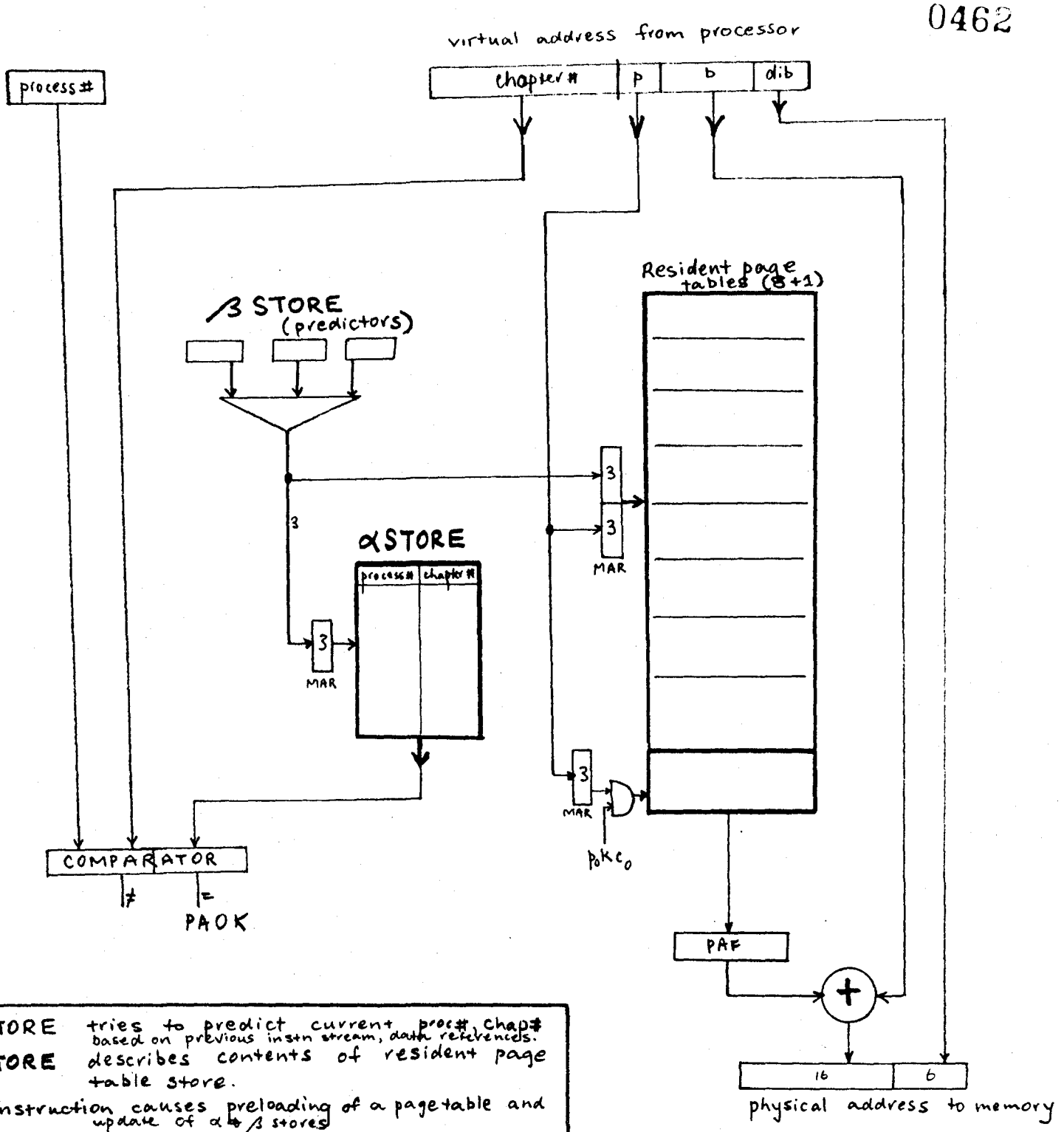
A more expensive implementation would have a larger resident page table store and the α and β stores would be replaced by an associative memory.

A cheaper implementation would delete process # from the α store and hold only the running process's chapter numbers. α store could then be the actual RiX, i=1,...8. Since LA causes preloading, this scheme would never fault. However, it would not have the nice context slaving properties of the Figure 3 scheme which decreases context switch time by retaining page tables for processes other than the running process.

Simulation with 11/VX programs must be done to test the effectiveness of the β store predictor scheme.

KT11-X

0462



1) β STORE tries to predict current proc#, chap# based on previous instn stream, data references.
2) α STORE describes contents of resident page table store.
3) LA instruction causes preloading of a page table and update of α & β stores
4) Comparator output:
   - =: [allow PA formation to proceed
   - ≠: [search α store for match on proc#, chap#;
     if found then set β store;
     else do;
       load page table from memory;
       update α store;
       update β store;
     end;
     repeat translation;

VII. Design Decisions

1. "Linear" vs. "symbolic" ("two-dimensional") address space
   c and d must be independent!
   a. Only way to enforce one chapter programs - hence need it for compatibility.
   b. better for programming.

2. Relation to integer 32 data type

   Since c and d independent can't use addressing mechanism to give 32-bit arithmetic for free.

3. Instruction format of new instructions.

   No 3-bit opcodes left, hence can't have MOVA SS, DD which would subsume LA and STA.

4. Chapter 0
   Making it special justified on grounds of interrupt responce.

5. Mode bit

6. 32→16

7. Process management (using process #) not to be assisted.
   e.g. ATL scan, insertion, deletion of nodes not to be assisted

8. Questions:
   JSL, RET not necessary if we pay time penalty and use [redacted] (SP)+ and LA.

VIII. Programming examples

1. FORTRAN array addressing
   array > 32K words

2. passing parameters
   a la old 11 style ok

3. dynamic linking

   Linktime: a. chapter # allocated
   b. chapter table entry:
   PTB set to null;
   access rights set as required

   Execution time: if null PTB then search system-wide link table;
   else ok;

X. Previous Work

1. Atlas, Dennis, 360/67, Multics

2. (DEC) Strecker, Rodgers, Mudge, Burness

Acknowledgement: Ed Marison's contribution to 11/VX

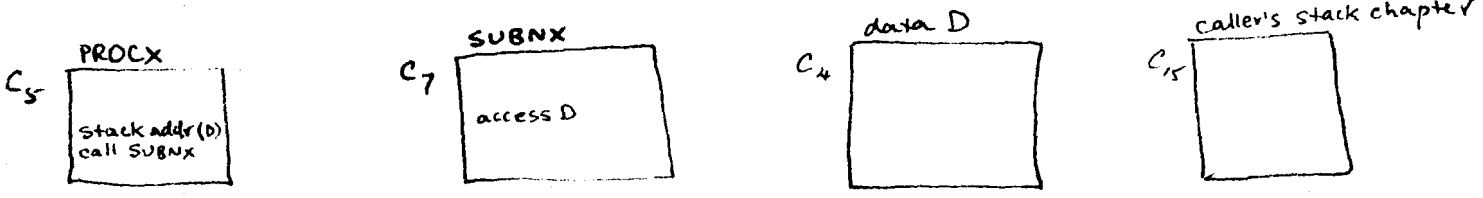
APPENDIX A

X-mode program calling an existing non-x mode program

PROCX, the caller, ~~code~~ wishes to invoke SUBNX, a subroutine written for today's 11/45, to operate on some data D.

Without loss of generality, assume (1) SP is used for communication between PROCX and SUBNX, and (2) PROCX, SUBNX, D, and the stack occupy separate chapters.

Thus we have



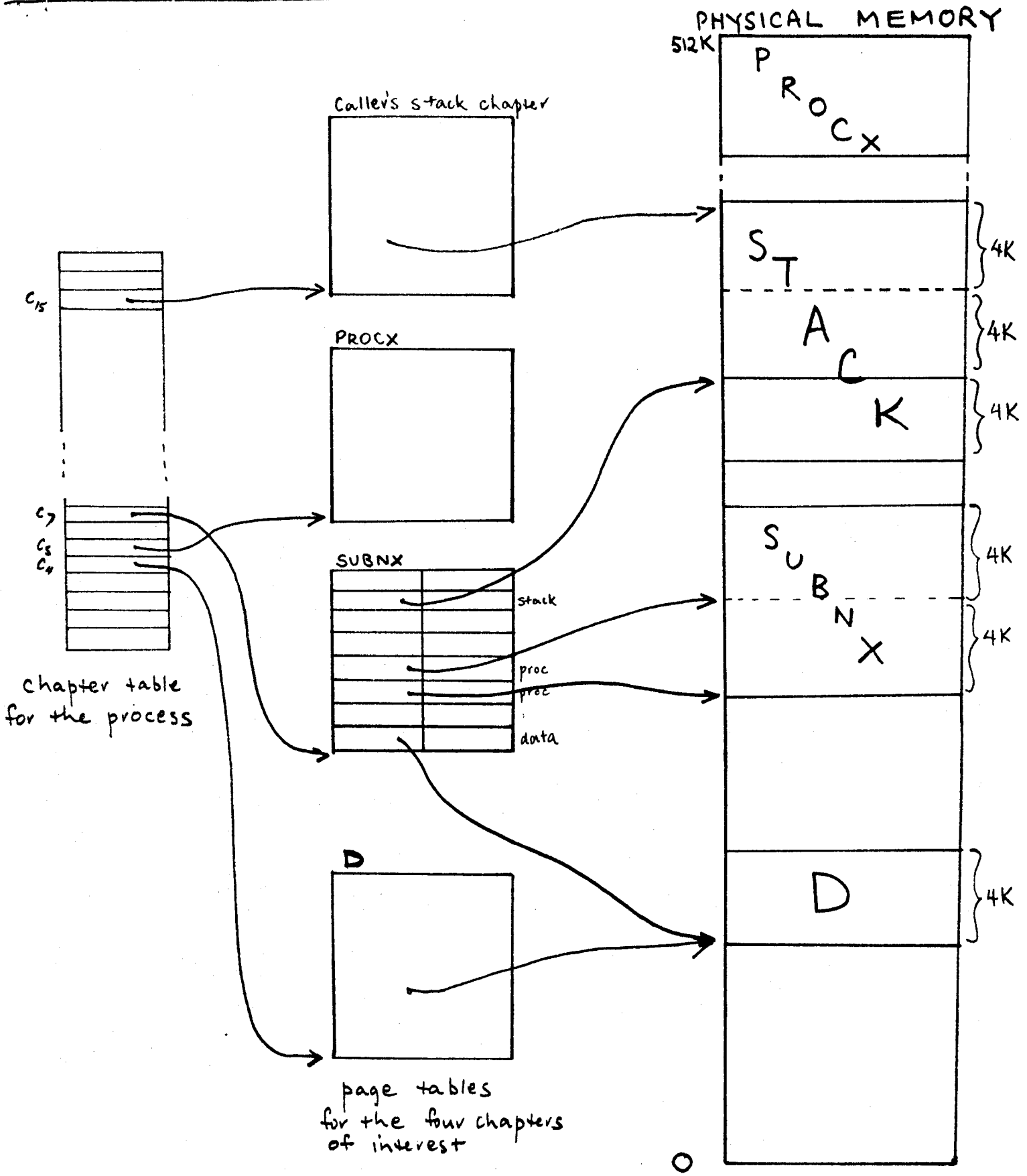
C_7 is prefixed by an interface routine, IFR. This IFR will switch the PS to non-x mode and so force all references to remain within C_7 (PCX had been set to 7 by the ~~[illegible]~~ interchapter jump from PROCX to SUBNX). All that remains is to ensure that data areas referenced by both PROCX and SUBNX, namely D and the stack, appear within C_7's 16-bit VAS. This, of course, is done by mapping.

Figure 4 shows the mapping. The example is slightly more complicated than necessary ~~[illegible]~~ — it shows SUBNX mapped to a subset of the caller's stack.

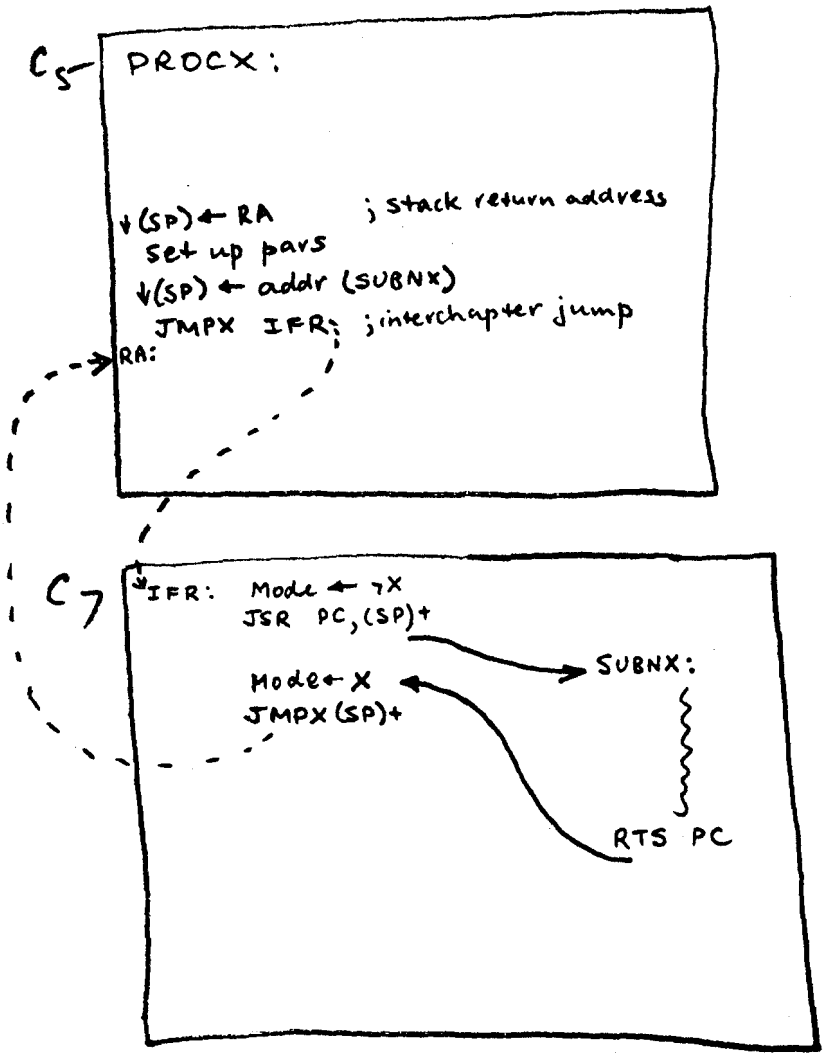
The IFR and JMPX to invoke SUBNX are as

FIG 4

MAPPING FOR PROCX CALLING SUBNX EXAMPLE



follows:



The IFR, which will handle all non-X mode subroutines in chapter 7, can be prefixed to a chapter load image by the linker/loader — SUBNX needs to be relinked only, not reassembled as well.

Compatibility derives from:

a. PS<08>, the X-mode bit
b. KT11X ≡ KT11C
c. IFR

digital INTEROFFICE MEMORANDUM

TO: Craig Mudge

CC: Jega Arulpragasam
Bill Demmer
John Levy

DATE: May 13, 1974

FROM: Bruce Delagi

DEPT: 11 Engineering

EXT: 3563 LOC: 1-2

0469

SUBJ:

I've some uncertainties about the 5/3/74 chapter scheme:

1. Why is it "no loss" to prohibit index constants to be used as base addresses e.g. OPR TABLE (R2) where R2 holds a 16-bit displacement and TABLE is a 32-bit base address?

2. How will the assembler expand "JMPX K(RN)"

   Possibly:
   ```
   STA RN, -(SP)
   ADD #K, 2 (SP)
   ADC (SP)
   [DEC (SP); if K < ϕ]
   LA (SP) +
   ```

   Note that JMP K(RN) ≠ MOV K(RN), PC

3. Does "X-MODE" refer to the state of a bit in the PS word? If so, is it changed on interrupts and traps. If it is changed, the RTI/RTT can't tell from the PS word whether to pop 3 words and restore PCX or not. If it is not changed, then interrupt and trap service routines must be rewritten and take proper action for both 16-bit and 32-bit pre-interrupt environments.

4. Trap service routines sometimes expect arguments on the stack at a fixed position relative to the top-of-stack. If PCX is pushed, the old relationships between top-of-stack and arguements is messed up and the old trap service routines must be rewritten. If PCX is not pushed, then the old trap service routines have to be rewritten to handle 16-bit and 32-bit calling environments.

5. A call to a 16-bit subroutine from a 32-bit environment requires a call to the operating system to map a segment of the chapter holding an argument into a segment of the subroutine's chapter.

   a. This seems to take 1 page entry per argument. What happens for subroutines that require more than 6 arguments?

6. If an argument is a pointer, then must the page holding the pointer and the page holding the data both be mapped. Suppose the pointer points to a pointer. Does the call site have to trace down the chain and call the operating system at each level? Suppose the subroutine uses arguments in address calculations (like adding 2 arguments) what can be done at the call site then?

11/XX — Reply to Bruce's memo of 5/13/74

1. Yes

The claim that there is no loss to restrict index constants to 16 bits is false

Since chapter scheme has only 16-bit index constants, to effect the equivalent of op A(R2) where A is a base address (c, d)

we must load R2X with c_A and then execute op d_A(R2)

The losses include

a. this addressing of A could be perceived to be unnatural

b. sometimes an extra index register must be used, e.g., the code for the FORTRAN statement

A(I) = B(I)

when A and B are in different chapters

2. No

JMPX K(R_n) ≡ LA K(R_n), PC

because we have only 16-bit index constants.

3. X-mode bit (PS<08>) has two uses

1. When X=0 it forces R_iX to be read as = PCX
2. On RTI it determines whether there is a PCX to unstack

4. No

We cannot reasonably be expected to cater to Mesa programming practice ie. indexing of the top of the system stack by a process which can be interrupted.

5. YES and no. Bruce's remark highlights a limitation of using old 16-bit routines.
If ≤6 distinct chapter addresses are being passed then no "call to the operating system" is needed (static allocation of the KT map)

If >6 then the KT map must be changed dynamically

But this is expecting the old 16-bit subroutine to do more than it could before, ie. see more than 32K virtual address at one instant in time

6. I don't think so
— we need elaboration

Craig
6/28/74

digital INTEROFFICE MEMORANDUM

0472

TO: Distribution List

DATE: 5/9/74

FROM: Ed Marison

DEPT: 11 Software Engineering

EXT: 4868 LOC: 3-5

SUBJ: SOFTWARE PLAN FOR THE 11/44

This is not a "Plan" in the usual sense, but a statement of concerns and assumptions about system software support for the PDP-11/44. It is based upon my knowledge, conceptions, and misconceptions of the PDP-11/44, and the software systems which will run on it.

Uni-Processor Systems:

Time to Market goal Q2 or Q3 of FY76

The PDP-11/44 comes in three basic configurations which the following adjectives describe, Small, Medium, and Large. The characteristics which differentiate the configurations are memory size and management.

- SMALL ≤ 28K no KT
- MEDIUM ≤ 124K with KT
- LARGE > 124K with KT and Uni-bus Map

Small Systems:

- Direct replacement for the PDP-11/40
- EIS maybe standard
- FIS optional
- Parity non PDP-11/40 compatible

As can be seen from the above the RT-11 and RSX-11M unmapped systems should run without modifications if parity is disabled. New code will be needed to support the '44's parity option.

Medium Systems:

- Same features as small systems plus
- KT - program compatible with PDP-11/40, but may have "D" space. Also, all bits in the PAR's will be implemented (ala the PDP-11/55).
- Null Uni-bus map (ie. transparent to software).
- FPP - Program compatible with the PDP-11/45's FPP, but maybe synchronous (also, may never happen).

With the exception of parity as noted in Small Systems the following software systems should run without modification.

- RSX-11M
- RSX-11D
- TSOS

However, in systems of this size memory parity should be supported to allow gracefull system degradation.

Large Systems:

- Same features of Medium Systems plus
- Uni-bus map - program compatible with the PDP-11/55

Given the above and the assumption that the PDP-11/55 is supported fully then all systems supporting the 11/55 will run unmodified, except for parity, on the 11/44. RSX-11D and TSOS should fall into this category.

In effect, given the above, all uni-processor PDP-11/44 systems can be supported with minimal system software effort.

Multi-Processor Systems:

. Time to Market goal Q4 FY77 - Q1 FY78

Only medium and large systems are considered here with the following goal being explicitly stated -

- No new Operating System should be written to support multi-processors, only modifications to existing OS's should be done.

Given the appropriate hardware support to prevent race conditions in the software between processors the following systems should be considered candidates for multi-processors.

- RSX-11M
- RSX-11D
- TSOS

To meet increased reliability requirements the goal should be to produce symmetric multi-processor systems. However, if we introduce "USER" micro-code then we may get systems with processors having different capabilities. Therefore, we will need the software capability for a Task to declare which processor it requires, and the hardware capability (processor number) to differentiate between processors (ie. asymmetric systems).

Firmware ("USER" micro-code) options for High Level languages:

. Time to Market goal - ?

This is an area where a high degree of cooperation between the hardware and software groups is a must. Just what languages should be

aided by firmware is a marketing concern. The following is a list of possible candidates:

- F4+
- F4S
- BASIC
- COBOL (the CIP - Commercial Instruction Set Processor)

Notes:

1. The areas of online error-logging, and diagnostics are ones which the various operating systems must address independently of the 11/44.

2. Networks - This is also an independent concern. However, since the PDP-11/44 is a PDP-11 it should fit very nicely into a DEMOS net with available 11 software at its time of introduction.

3. Virtual Address Space extension (VAS) is not covered in this plan. However, it should be noted that to support an extended virtual address (chapter scheme) would require extensive software effort in the order of five (5) to seven (7) man-years per system and 18 to 24 months of calendar time.

dml

Craig Mudge

digital INTEROFFICE MEMORANDUM

0475

TO: Distribution

DATE: August 21, 1974

FROM: Garth Wolfendale GW

DEPT: RSX11D Development

EXT: 3959 LOC: 3-5

SUBJ: RSX11D Group's Position/Thoughts/Concerns on VX Proposal

SUMMARY

The mapping scheme proposed is a good cost-effective scheme for achieving a greatly expanded task (process) addressability.

We could support such a scheme for a cost of approximately $150K for executive and related developments (e.g. task/process builder).

This cost does not include that involved in upgrading compilers, etc.

HOWEVER

In order to support larger processes and at the same time maintain an effective operating system which can handle multi-users with effective response times, we are concerned that the chapter and segment handling will present possibly grave limitations on the number of "large" processes that the system can handle and also maintain responsiveness.

The biggest problems here are: -

Real memory management...the ability to find and allocate real memory for a loading task.

And

Fragmented process loading and recording involving many segments.

Specific proposals for alleviating these problems are being discussed and the corresponding trade-offs are still being evaluated.

digital INTEROFFICE MEMORANDUM

TO: Extended Addressing Review List Attendees

DATE: March 18, 1975

MAR 21 1975

FROM: Tom Hastings

DEPT: -1Ø Software Engineering

EXT: 6512 LOC: MR 1-2/E37

SUBJ: Minutes of 11 March Review of Extended Addressing

We finished the wide review of Extended Addressing from the user programmers point of view. Most everyone is generally satisfied with the concepts and the details of the specification. Only the monitor interface (PXCT) specification remains. Only one ECO for the 1Ø8Ø is required. Four or five boards will have to be relayed out for the 2Ø series. These can be phased into subsequent 1Ø8Øs, so that we can achieve the goal of the 1Ø8Ø being a strict subset of the 2Ø1Ø in function and hardware.

Minutes: The agenda items are indicated in parenthesis.

1. Problem: (1Øv) The items Local and Global address will be confused with software use of these terms.

Solution: Use terms Short and Extended address instead.

Why: These terms are almost self explanatory. We are not inventing new jargon.

2. Problem: (1Øa) Should the spec be changed to say that KI compatible effective address computation is determined by PC bits 6-17, instead of VMA bits 6-17?

(1Øb) There are multiple uses of zero: They are : (1) Short address in XRs, (2) KI-compatible section in EFIWs, and (3) hardware ACs on UUOs and MOVXA.

Solution: Leave KI-compatible test on VMA rather than PC.

Why: Effective address computation works the way the caller expects. Once an effective address computation gets into section Ø, it remains there.

Solution: Make the section independent extended address of the hardware ACs be 1,,AC instead of Ø,,AC. Effectively section 1 must be a code section in the extended machine with the first 2Ø locations always being the hardware ACs. Sections 2 and greater can be used for code or data. Hardware ACs (in all sections) are used if VMA 6-31 = 1,,Ø or (VMA 18-31 = Ø and SA = 1). (SA is

Short Address Flag). When the PC is in a non-zero section, MOVXA will generate 1,,AC whenever a hardware AC is specified (whether Short or Extended effective address). An EFIW of Ø,,AC will reference memory in section Ø. When the PC is in section Ø, MOVXA will follow the KI rule: the LH will always be Ø.

Why: Then the monitor and user code can generate an address with MOVXA which will uniquely specify a section independent address for all locations in the machine. This address can be copied to other sections (except from Ø to non-zero sections) and it will mean the same location.

Solution: We decided to leave the double use of Ø to mean Short indexing in XRs and KI compatible section in EFIWs.

Why: We are not anticipating code in Extended sections referencing code in KI compatible sections, especially not general purpose subroutines. Exceptions will be specially written code, such as debuggers and compatibility packages which will be written to run in a non-zero section and operate on programs in section Ø only. They must reference section Ø with indirection (EFIWs) rather than indexing or with indexing in which the program has set bits 1-5 non-zero. The monitor has the same problem. The solution will be worked out during PXCT review.

3. Problem: (1Øb) Double Word Byte pointer is too complicated.

Solution: If bit 12=1 (when byte is in non-zero sections, not PC) the 2nd word is an indirect word (EFIW or IFIW). Bits 13-17 of the first word must be zero and are reserved for future hardware. However the hardware will not trap. Bits 18-35 are reserved for software forever (in DWBP with bits 18-35 when bits 13-17 zero). We will decide for future machines what happens to bits 18-35 when bits 13-17 are non-zero. Overflow into bits Ø-5 in EFIW, or Ø-17 in IFIWs can occur and no trap will occur. Since section 7777 will be illegal by software convention, accidental overflow seems unlikely.

Why: Software can use RH for counters, pointers while leaving a hook for future hardware.

4. Problem: (1Øc) Should incrementing a byte pointer with indirection be fixed to increment the last word in the indirect chain, instead of the first word of an SWBP or the second word of a DWBP?

Solution: No.

Why: Subroutine argument which are byte pointers which point to other sections should always be DWBPs. Then callee can pickup. Don't introduce a potential incompatibility with KI if don't need to.

0478

5. Problem: (1Ød) Is there any solution to the problem that a byte pointer with an Extended Address in the XR, wants the RH of the byte pointer to be unsigned, instead of signed?

Solution: No. All Extended Indexing is restricted to a $\pm 2^{17}$ offset.

6. Problem: (1Øf) Should BLT backwards be decomitted? It is breaking KI programs. The manual says what happens if destination is greater than termination address.

Solution: Decommit it. Put it in EXTEND-BLT.

7. Problem: If effective address of a BLT is extended, can it cross ore section boundary, or should it always wrap around in a section?

Solution: Doesn't matter for software, so wrap around. If easier to do the other way, we will change it.

8. Problem: (1Øh) ADJSP - how can we make it check for over-flow and underflow with an Extended Stack pointer?

Solution: After adjusting the stack pointer, ADJSP will pretend to do a write into the top of the stack. This means that the stack can be completely protected by having 256 non-existent or read-only pages at either end of the stack.

9. (1Øk) What is effect of spec on 1Ø8Ø ship date? Only one ECO is needed (to access 6 ACs for EXTEND).

10. (1Øm) What should hardware/micro-code do with unused fields? There are 3 choices:

SOFT	1.	Give to software (forever)
MBZ	2.	Must be zero. Reserve for future hardware, but not trap if non-zero.
MBZ(T)	3.	Must be zero. Reserve for future hardware and trap if non-zero.

a. Double Word Byte Pointer
Bits 13-17 of 2nd word - MBZ, reserved for future hardware but does not trap. When 13-17=Ø, bits 18-35 of 2nd word are available to software forever (for counts, pointers, etc).

b. Bits 1-5 of Index Register
When bit Ø=Ø, bits 1-5 are available to software forever. However the programmer must be aware that if these bits are non-zero, Extended Indexing is called for (instead of Short Indexing even though bits 6-17 are zero). Also if the programmer wishes to indirect through the index register,

bits 1-5 will be interpreted by the hardware.

c. Unimplemented section number (bits 6-12 in KL).
(My notes are hazy for item c. The following is results of discussion with R. Reid and D. Murphy). In the KL whenever the hardware fetches a word from memory, a trap will occur if VMA bits 6-12 are non-zero. The trap is the same as for unallocated section number. Thus the hardware appears to implement all 4Ø96 sections. (Since the VMA is only 23 bits long on the KL, there will probably be a flag saying whether VMA 6-12 are Ø or not).

d. Bits 6-12 in index registers.
Not looked at when index register is fetched. Instead checked as in c above. When XR contains an Extended Address programmers must keep bits 6-12 zero for future hardware which may implement more than 32 sections.

e. Bits 2-35 in IFIW when bits Ø-1 = 11. MBZ(T). Microcode will give illegal instruction trap.

f. Bits Ø-5 in EXTEND-BLT pointers. Available to software forever (like index register bits 1-5).

g. Bits Ø-5 when restoring PC in POPJ, and RPCF.
There was no agreement. Some said reserved to software, as with bits 1-5 in XRs. Some said reserved for future hardware. Some said hardware should trap.

Since then the proper goal for PC words has been found. A PC word must be the same as an EFIW in KL and in all future machines, since programs indirect through PC words. Therefore the hardware cannot ever use bits Ø-5 in future machines. Furthermore software is warned not to use them, unless the indirection property is not needed. Therefore the hardware will not check bits Ø-5 on restoration of the PC.

11. Review of opcode names

The new opcodes will be:

RPCF	Restore PC and Flags
RPCFD	Restore PC and Flags then Dismiss
EPCF	Exchange PC and Flags
SFM	Save Flags in Memory
MOVXA	Move Extended Address

12. The following items were deferred:

a. (1Øi, 1Øj) EXT-BLT specification (needs a separate spec)
b. (1Øg) Rules for writing section-independent subroutines (T. Hastings will write-up)
c. (1Ør) Rules for writing subroutines which will run in section Ø, extended sections, and KA and KI.
d. (1Øs) PXCT
e. (1Øt) User interface to the monitor (do with PXCT renew).

Attendees:

Tony Fong
Gordon Benedict
Mary Cole
Lloyd Dickman
Bob Stewart
Al Kotok
Tom Hastings
Paul Guglielmi
Walt Luse
Dan Murphy
Dave Rodgers
Robert Reid
Jud Leonard

/lo

FREDERICK M. TRAPNELL, JR. Computer Systems Consultant

6321 Hardley Drive
La Jolla, Calif. 92037
(714) 459-5530

THE NEED FOR EXTENDED MEMORY ADDRESSING IN THE PDP-11

The Reason for the Need

The principle change in the mini-computer market over the next several years will be the broad realization by users and manufacturers alike that the mini-computer technology is applicable to business problems which are in no sense "small". Indeed the embodiment of that technology in the PDP-11, the Interdata 7/32 and the Hewlett-Packard 3000 yield capabilities which cannot be matched by the 360 series below the level of the model 50. The PDP-11 family offers instruction execution rates which range from 200K to nearly 1 million instructions per second and memory bandwidths from 18 to 80 megabits per second. By standards of five years ago this is not a "small" computer. The man who needed an IBM 360 model 50 at model 30 prices to do a job now has something even better.

Thus, this new market will arise in the traditional business application area. But, it will provide for the automation of applications which are not now done on a computer for reasons of cost. The "mini" computer will change all that; it will open to system designers vistas that could not be considered five years ago! It will make practical the broad use of on-line, real time business control systems which could only be justified in rare cases in the past

used programs resident to main memory. Ideally the main thread of these applications should not require disk access to any programs. This has the dual benefit of increasing performance by doing away with all but essential disk accesses to data and of reducing overhead by eliminating the need for complex overlay control mechanisms. This requires a large memory which can be accessed directly by the user.

This reason for increasing memory size is markedly different from the conditions which led to ever larger memories on commercial data processors during the past decade. This came about by users trying to get the least job cost out of a given system. The availability of advanced operating systems allowed them to buy the largest core memory for the system and to multiprogram a number of separate jobs on one computer. The end result is that today most data processing users run as many multiprogrammed batch applications as possible on their computer, and most small-intermediate D.P. computers (e.g. 360, models 30, 40, 50) use the largest available memory.

Mini-computer costs today are such that there is much less to be gotten from multiprogramming unrelated jobs on one computer; it is nearly as cheap to have two dedicated ones. Thus, time sharing of unrelated applications will become increasingly unattractive. To be more relevant, the need for address extension on the PDP-11

Likewise, a facility for relocating programs must be considered. The key question is whether or not relocation at load time is sufficient. If it is, then relocation can be left as a software problem; if not, then additional hardware will be required. My own feeling is that for this kind of system, load-time relocation by software is adequate. But, it is very cumbersome, and it may be an uphill selling job against a product which provides it.

However, there is a far more important point related to these issues of extended addressing, protection and relocation. It is the second design guideline:

2) From the viewpoing of system architecture (and hence the user) extended addressing, protection and relocation are three separate issues. A mechanism that provides one of these facilities (like page registers to provide extended addressing) may also be used to provide others (like protection and relocation). The extent to which this is done confuses these issues and diminishes the usefulness of the system; the decision to do this is a design compromise.

Impact on Existing Programs

Another constraint on the design of an extended addressing mechanism comes from the impact these changes will have on the investment in DEC's and user's existing programs for the PDP-11. The next guideline is:

3) The solution to extended addressing will be better to the extent

that it does not require: (listed in order of importance)

A. DEC to rewrite assemblers, compilers, and operating systems-

1) to run existing programs in the environment of the new facilities.

2) to take advantage of the new facilities.

B. Users to rewrite code-

1) to run existing programs in the environment of the new facilities.

2) to take advantage of the new facilities.

In addition, some marketing benefit can be obtained if users are not required to re-link or recompile programs.

Ease of Use

Whatever mechanism is provided in hardware for memory extension, the user must be given the facility to be sure that his programs contain no addressing errors by the time the linking process is done. For example, if DEC were able to extend the address length in some magic way from 16 to say 32 bits, then the user could simply use the new "magic" addresses and be sure that his program would work. However, if the address extension scheme involves dynamic user management of memory pages, then this may not be the case: If not, this is not an acceptable solution. This gives us the fourth guideline:

4) At assembly and/or link time the user must be able to code instructions and declarations which define the paging dynamics well enough so that errors can be detected by the assembler and linker. The rules for this coding should be easy to understand and simple

Others are faster on the Interdata machine; for example, the JSR takes 3.5 microseconds while the branch and link on the 7/32 takes 2.0 microseconds. Memory bandwidth of the PDP-11/40 is 17.8 megabits per second, while that of the 7/32 is either 16 or 21.3 megabits per second. The Interdata 7/32 can directly address 1,048,576 bytes or 512K words. This means that users who find either 32K words per user or 128K total memory words a limitation now have an alternative.

Thus the first mini-computer with this capability is now on the market; others will surely follow. When these machines will impact DEC business, I cannot say. Suffice it to say that announcing extended addressing on the PDP-11 today would not make DEC first in the market place with such a product.

The Segmentation Approach

Suppose we try to build on the memory management system which is currently available on the PDP-11/45 (Assume that the Page Address Field is extended to give 20+ bits of real address). Suppose we then try to concoct a hardware/software scheme which meets the requirements set down in "The Need for Extended Memory Addressing on the PDP-11"; namely:

1) Must be fast to change maps.
2) Must be easy to use and error-free once linking is complete.
3) Must give protection

How would we then proceed to get a practical solution which had minimum hardware impact?

N.B. This solution is mixed hardware and software with some attempt (not guaranteed) to distinguish between them explicitly at this stage.

1) All programs, data, store areas and for buffers, etc. will be coded or allocated in segments. A segment is defined as follows:

a) A segment is an area of code, data, or storage space which can be mapped into no more than 28,672 words of contiguous virtual addresses (7 APR's). A segment is made up of one or more modules which have been assembled or compiled. These are then built into a segment by the Segment Builder (called hereafter Segbldr). This program operates like the linker to resolve all internal globals to make segment a program segment a self contained unit of code. The access control for a segment is the same for all pages in it and for all user programs which

in physical memory
segments are relocatable; in fact the pages within segments are too.

Global symbols which define locations to be referenced from outside the segment are called external globals or externs. Externs for each segment must be explicitly declared at assembly time. The access control code which applies to a segment must be declared at segbld time.

Completely built segments are put together into working user programs by the segment mapper, called hereafter segmap. Segmap's function is to map the segments into pseudo-physical memory locations. ~~pseudo locations this~~ This is the equivalent of a contiguous area of physical memory; however, it can be relocated at load time.

Segmap also constructs a separate

module which contains the ~~capabilities~~ capabilities to access ~~all~~ ~~eptems~~ each segment. These capabilities are 3 word quantities (probably) giving the starting physical address of the segment, its length, and the other ~~bit~~ data needed for the PDR's which will reference it.

To address an eptem the user must cause to be loaded (or be sure it is already loaded) ~~the~~ into the active page registers, the information needed to access the segment containing the eptem. He does this with a ~~TRAP or EMT~~ macro which expands to an EMT or TRAP plus a specification of the capability to be loaded and its starting virtual address.

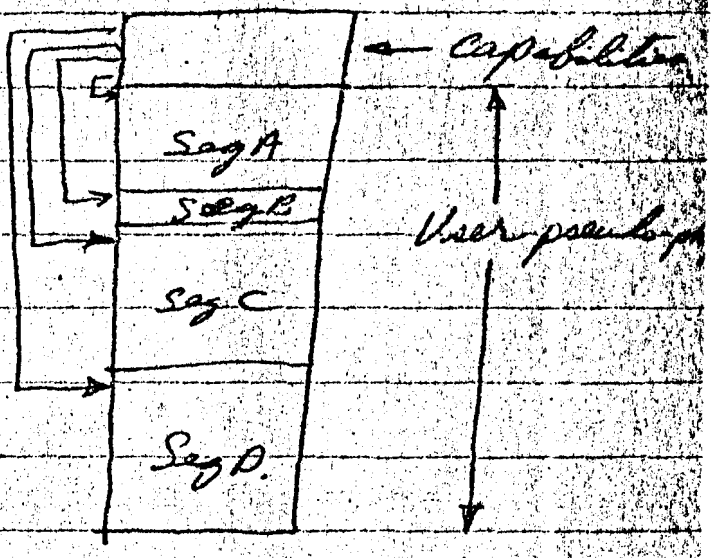
NB. Here's where this one gets sloppy.

That means that either the programmer or the assembler has to be smart enough to know what segments are loaded where (in virtual memory) at execute time. An limited version of the

assembler can tell, but not always! For example, a called subroutine may have either APR's or capability pointers passed to it as parameters. Or, the path of execution to the point of accessing an extern may be conditional upon execute time situation. Therefore, there is no good way to let the assembler work out the true situation. Maybe the best that can be hoped for is that the ~~user has to~~ assembler will point out to the user where potential pitfalls are.

I really don't like this because it expects too much of the programmer. I'm sure this is a fundamental weakness of any scheme which doesn't use O'Halloran's scheme or which doesn't map virtual to physical at execute time. Anyway let's keep going; maybe a solution will come to mind or perhaps someone else has got one.

We now have: (memory map)



The root segment of a user program must contain user page 0. This must hold his stack, ~~and the~~ entry for his interrupt service routine entry points, etc. It can never be overwritten (the assembler prohibits it).

Some new instructions:

	New Mnemonic	Counterparts
Jump segment	JMS	JMP
~~Jump subsegment~~		
Jump segment subroutine	JSS	JSR
Return segment subroutine	RSS	RTS

The functions of these new instructions are the same as for their counterparts, except:

a) all three of these permit specified active page registers to be cleared. ~~(not page zero)~~ (The assembler won't let page 0 be cleared).

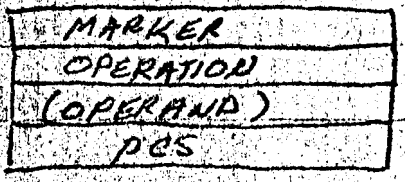
b) JSS can also cause specified page registers to saved on the stack.

c) RSS will automatically restore APR's saved by a JSS.

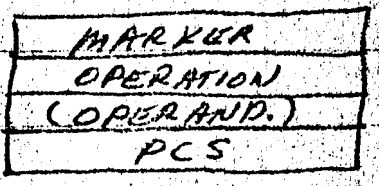
JMS and JSS are ~~three~~ two or ~~three~~ four word instructions; RSS requires two. Their operation and operand words are identical to ~~their~~ their respective counterparts. They are preceded by a marker which identifies them.

Formats are:

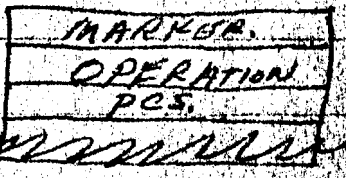
JMS



JSS



RSS



The PCS stands for Page Control Specification. ~~It's~~ format is two bytes, each with ~~a flag specifying~~ eight one bit flags associated with the eight APR's. The low byte specifies those APR's which are to be cleared during execution of any of these instructions. The high byte of the PCS is used to control saving and restoring in JSS and RSS.

JSS uses its PCS as follows:

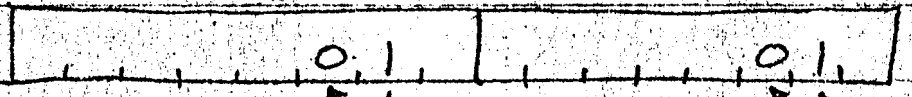
~~1)~~ The APR's which have a flag set in the highbyte of the PCS are dumped onto the user stack. The PCS itself is then dumped to the stack. The physical address of the destination of the jump is computed. The APR's specified in the lowbyte of PCS are cleared; and ~~the~~ a JSR is executed.

(9)

RSS uses the PCS's as follows:

The APR's which have a flag set in the lobyte of its PCS are cleared. Then an RTS is executed. The ~~saved~~ APR's specified in the highbyte of the PCS on the top of the stack are restored.

PCS format.



a) This says don't save (or restore) APR reg 2

b) This say save APR reg 1 (JSS) or restore APR reg 1 (if old stack in RSS)

This says don't save APR reg 2

This says retain APR 1

With these instructions (plus the "load segment" macros) we have clean, protected access to segments.

After we've done all this where are we?

1) It looks like we have good, protected access to segments. No segment can get at something it is not either coded to access or passed by a calling segment. Protection Segments are protected by their capability pointers, which the user cannot access directly. Thus from a protection viewpoint it looks good. Also you can get all the address range you need by extending the PAF to 14 or more bits.

2) Programming this is a beast, and it is prone to undetected errors. As I said earlier maybe we need a new idea to clean this up. If we could find one, this could be a good scheme.

The Magic Address approach:

Suppose we could in some way "magically" extend the PDP-11 addresses from 16 to 24+ bits. How would we do it? What problems would arise? How bad a compromise is it?

Objectives:

1) Minimize architectural changes required.
2) Minimize (a) recoding (b) reassembly, if possible
3) End up with the user having an unsegmented address space as big as the address ~~space~~ range with no user or supervisor gimmicks (eg no dynamic page changing)

Some considerations:

1) We should allow both kinds of addresses, (16 bit conventional and 24 bit), called a short address and a long address respectively.

2) If we restrict the user to assembling (compiling) modules of no greater than 64K bytes, then the assembler should not need to know the type of address it resolves what it can; the rest are globals.

3) The assembler can't know whether a global is to be a short or long address. It must allocate space in the program the same way for both. Let's proceed on the assumption that it allocates space as it does now: one word for each operand. (With luck the assembler won't have to be changed!)

4) The big question is what to do with the general registers. Sometimes they hold addresses; sometimes data.

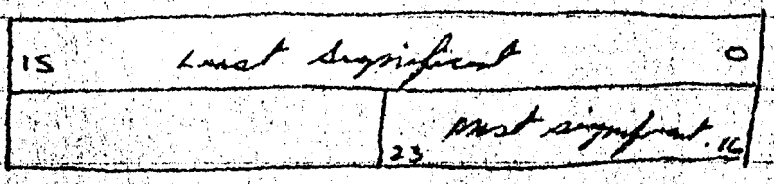
X) Let's keep data the same. We're not designing a 32 bit machine; we're trying to put long addresses on a 16 bit machine. We want to keep everything else the same, if possible.

X) Let's see what happens if we let the registers hold both kinds of quantities: 16 bit short addresses and data as well as 24 bit long addresses.

Changes:

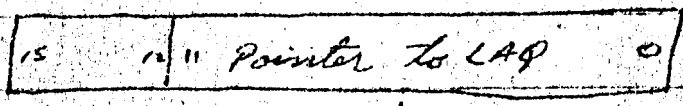
1) Define a new, two word (32 bit) quantity in memory called a Long Address Quantity or LAQ. These are made up of two adjacent words: the lowest address word holds the least significant part of the address, the other word hold the most significant part.

An LAQ:

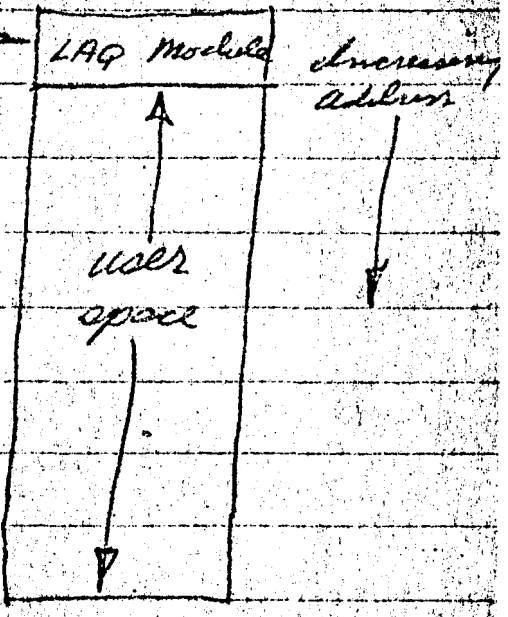


LAQ's are never contained in code. The linker builds a special module containing them. Indeed, the assembler doesn't know whether an address in the code is to be long or short; only the linker can know. It decides when it builds the complete program which ~~global~~ references must be ~~put~~ long addresses.

2) Define a new one word (16 bit) quantity called a Long Address Vector or LAV. In its basic form bits 0-11 hold a pointer to one of (possibly) 4096 LAQ's stored in the LAQ module built by the linker. The other bits are reserved for special functions (later).

LAV: 

Memory Map:



LAV's are found in two places: (1) as operands of instructions when the linker determines that the desired global is beyond the range of a short address and (2) as the indirect address, ~~in an instruction which specifies indirect~~, when one is specified

and when a short address has insufficient range

Here we run into a problem. ~~Short address not~~ We must have some way of knowing whether an indirect address is a short address or an LAV. It is not good enough to know before the indirection has started, because in the general case neither the assembler or linker can tell precisely which address the indirection will go through; this may only be clear at execute time. Therefore, we must provide a way to identify in the indirect address whether it is a short address or an LAV. Two solutions:

1) Make all of them LAV's - may be expensive in space or time. For example, even a vector table with no globals would have to be treated this way.

2) Restrict assembled modules to 32K bytes. Therefore the M.S. bit is always zero ~~never used~~ in a short indirect address. Let it

address or an LAV. - A bit of a perversion for this architecture, but on the face of it not too unclean. Let's proceed assuming this.

When a LAV is an operand then ~~the instruction must be specify either~~ the address specification for in the instruction must call for either (a) a reg = 7 address or (b) a mode 6 (index) or 7 (index deferred) address. ~~In case (a) the~~ linker (a)

In case (b):
The linker will convert that address spec into a mode = 0, reg = 7. It will then insert into bits 12-15 of the LAV (a) the register originally specified in the addr. spec. (3 bits) and one bit to say whether the original mode was 6 or 7.

In case (a):
The linker will convert that address spec into a mode = 1, reg = 7. It will then insert into bits 12-15 of the LAV ~~and~~ an indicator of the original mode specified (2, 3, 6, 7).

3) Change the genl. registers so that they are 24 bits long. All ~~defun~~ currently defined operations continue to operate on the L.S. 16 bits. ~~only, except that~~ carries and borrow carries and borrows, however propagate into the upper 8 bits. MOV and CLR with d=reg clear the upper 8 bits. Shifts + Rotates operate on lower 16 bits only. Present condition codes operate on the lower 16 bits only.

Whenever a register is a source or a destination, it will be treated as above. When, however, it is used in any but mode=0, it is treated as an unsigned 24 bit quantity.

4) Add two new condition codes to PSW. These correspond to C and Z, ~~they~~ are called C' and Z'; they work on the 24 bit unsigned quantities.

5) Need some new ~~branch~~ instructions which ~~work~~ are explicitly designed to ~~work with~~ EA manipulate and move EAQ's (or the 24 bit ~~portion~~ long form of the general registers. It is proposed that this be a strictly limited subset of the 16 bit manipulating instructions which currently exist. After all it is still a 16 bit machine; these are only for manipulating LAQ's.

~~If~~ These new instructions will have ~~the~~ op codes, etc identical to their ~~own~~ counterparts which process 16 bit operands. They are distinguished by a unique 16 bit marker which precedes them in the code. (This is generated by the assembler). (Now we've blown it, we've changed the assembler! But, only for these special instructions; hang in there, baby!) This marker tells the hardware that a long address manipulating instruction follows; this, in turn, causes the (newly defined) ~~Ebit~~ L bit to be set temporarily in the PSW (or suspend interrupts til finished).

New instructions are:

a) Branches which operate on C' and Z'

		Br if
Branch if LA higher	(BAH)	C'VZ'=0
Br if LA lower or same	(BALS)	C'VZ'=1
Br if LA higher or same	(BAHS)	C'=0
Br if LA lower	(BAL)	C'=1
Br if LA equal (to 3rd)	(BAEQ)	Z'=0
Br if LA not equal (to 3rd)	(BANE)	Z'=1

b) Long address manipulates

Move Long Address (MLA)
Both source and destination are treated as 24 bit long addresses, in LAQ format if in core. - unfortunately we need this one, ~~because there is no way to tell~~ see below.

~~CLA~~
(Compare Long Address - this is not required provided the hardware is made to do both a short and long comparison and to set all condition codes accordingly (including C' and Z').)

We need the MCA because, in general, only the programmer knows whether he is moving an address or data. For example, the well known save and restore convention...

```
MOV  Rn, -(SP)
  :
  :
MOV  (SP)+, Rn
```

This leaves you clueless as to whether the programmer means to save and restore an address or data. Or if you think the above should always move 32 bit quantities by MOV R0, (R6) where R6 points to a table entry (of addresses or data?) Therefore, the proposition is that we only add these seven new mnemonics to the Assembler repertoir.

6) Some oldies that execute differently:

JSR, RTS, RTI dump and restore two word LAQ quantities. ~~Interrupt~~ Interrupt vectors have two word addresses. Actually, you might be able to exempt RTI and the interrupts if you insist that the interrupt processing routines are in lower 32K words of store. This is no serious problem if the machine has one relocation register.

7) About the LAQ module:

This is built by the linker as it resolves out-of-range globals. There will be one LAQ in it for every such global; all programs which need a reference that global will share use the one LAQ. There is address range in the LAV for 4096 of these, which should be enough.

8) After we've done all this where are we

a) it looks like it should work and if you'd never seen a PDP-11 before these changes were made you'd probably learn this ok.

b) it looks like it really does get pretty close to the "magic address" addition: it gives you full addressing up to 24 (or more if you want it) bits for any user. It needs some protection added to it, and every machine ought to have a relocation register which is part of every address calculation (but that's not a new problem.

However, there we've left a few well concealed land mines around for the unwary programmer who thinks that "magic" is really magic and that all his existing programs will run unchanged.

a) Any subroutine called by a JSR and which expects to find parameters under the return address on the stack now must look one word deeper.

b) It's possible that ~~data~~ 16 bit calculations on a register could leave trash in the upper byte, since carries and borrows will now propagate up there. Currently this trash migrates out through the C or V bits and gets lost. If you now use that register as an address, it sure won't work like it used to!

c) This one is analogous to (b). Consider the instruction:

MOV STAR (Ra), Rb

In this case you don't really know whether to treat Ra as a short or long quantity. If STAR is the name of a table and Ra is an offset into it, then you would like to treat Ra as a 16 bit quantity and get rid of the any trash. However, if STAR is an offset

in say a linked list block and Ra is the variable index to the blocks, then you want to treat it as a 24 bit quantity.

d) Also, I have the awful feeling that somebody will come along with a valid reason why you have to add and subtract address quantities. Somebody must calculate table lengths by subtracting base from limit. I may have done this myself, it just won't give a correct ~~valid answer. Hard to tell how bad this is.~~ answer for long addresses. But existing programs shouldn't create long addresses!

digital INTEROFFICE MEMORANDUM

TO: Jega Arulpragasam

DATE: October 25, 1974

FROM: Bill Strecker

DEPT: R & D Group

EXT: 4207 LOC: 3-4

NOV 05 1974

SUBJ: F4P Degradation on 11/VAXL

Summary

Based on making the worst case assumptions about the design of the 11/VAXL extension and its use by F4P the 11/VAXL extension appears to increase, for the program measured, the program static size by 40% and the number of I-Stream references by 36%. By being more careful about the design of 11/VAXL and its use by F4P I believe that the percentages for both cases can easily be reduced to the range of 10% to 20%. Since the function of this memo is neither to design 11/VAXL or redesign F4P, I have not documented this here, although I will be happy to do so.

Program Details

The program analyzed is a matrix multiply subroutine given in Figure 1. The symbolic object code produced by F4P is given in Figure 2 and Figure 3. It consists of 4 PSECTs. PSECT $IDATA contain the array descriptor block (ADB) templates for the arrays A, B, and C. Bytes 0 - 73 of PSECT $CODE1 contain code to build the ADBs. Bytes 74 - 322 contain the matrix multiply code. PSECT $VARS contains a partial address calculation which is invariant across the innermost loop. The current size of these PSECTS is given at the end of Figure 3. The number of I-Stream references of the innermost loop (bytes 110 - 233 of PSECT $CODE1) are written in the first column to the right of each instruction.

VAXL Considerations

A slightly modified 11/VAXL proposal (as I understand it) is assumed. In particular there are 5 address operators; Load Address, Store Address, Add 28 bit Address, Multiply Address, and Add 16 bit Address. The first three use a 28 bit operand and a 28 bit register while the last two use a 16 bit operand and a 28 bit register. Indexing and indirection is 28 bits for these instructions. The instruction memonics are LDA STP, ADA28, MPA, ADA16 respectively. For all current instructions there is a mode 5 escape sequence which provides a 28 bit index and a 28 bit indirect.

For the code examples that follow I use [source or destination] to indicate the mode 5 escape. It is clear that each use of [ ] adds one static program word. It also adds one I-Stream reference and two in the case of indirection.

Static Program Analysis

Since the three ADBs must now contain a double word base PSECT $IDATA is increased by three words. The code which generates the ADBs must be modified to pass double word addresses: The code to do this would replace bytes 0 - 23 of PSECT $CODE1 and would look something like this:

```
                                        Additional inline words
MOV     [@2(R5)] , -(SP)                        +1
MOV     [@2(R5)] , -(SP)                        +1
MOV     6(R5), -(SP)                            +Ø
MOV     10(R5), -(SP)                           +2
MOV     #   , -(SP)                             +Ø
MOV     #   , -(SP)                             +2
JSX     PC, [MKA2$]                             +1
                                                 7
```

Since this must be done three times it adds 21 additional program words. The additional words for the remainder of $CODE1 are indicated in the second comumn to the right of each instruction in Figure 2 and Figure 3. The total word increase for $CODE1 is 55. PSECT $VARS is unaffected while PSECT $TEMPS is increased by one word to hold a double word address. The summary of the incrases is given in Figure 3: The program size goes from 144 to 203 words - a 40% increase.

Dynamic Program Analysis

The inner loop of the program would look as follows:

```
                                        Additional I-Stream
                                        Word References
L$FAAL: MOV     [K] , R1                        +1
        MPA     [ ] , R1                        +1
        ADA16   [I] , R1                        +1
        MPA     #4  , R1                        +Ø
        LDA     [ ] , RØ                        +2
        ADA16   [K] , RØ                        +1
        MPA     #4  , RØ                        +Ø
        ADA28   [ ] , R1                        +2
        ADA28   [ ] , RØ                        +2
        LDF     @R1, FØ                         +Ø
        MULF    @RØ, FØ                         +Ø
        ADDF    [S] , FØ                        +1
        STF     FØ, [S]                         +1
        INC     [K]                             +1
        CMP     [K] , [@2(R5)]                  +3
        BLE     L$FAAL                          +Ø
                                                 16
```

The number of additional I-Stream references is given to the right of each instruction. Comparing this to the I-Stream references in the original program, the number foes from 45 to 61 - a 36% increase.

/br

0515

UNRELEASED SYSTEM, SUPPORTED FOR FIELD TEST PURPOSES ONLY

```
0001            SUBROUTINE MATMUL(N,A,B,C)
0002            REAL A(N,N),B(N,N),C(N,N)
0003            DO 10 I=1,N
0004            DO 10 J=1,N
0005            S=0
0006            DO 5 K=1,N
0007      5     S=S+A(I,K)*B(K,J)
0008     10     C(I,J)=S
0009            RETURN
0010            END
```

Fig. 1.

UNRELEASED SYSTEM, SUPPORTED FOR FIELD TEST PURPOSES ONLY

Static Increase

```
                .TITLE  MATMUL
                .IDENT  29AUG
000000          .PSECT  $IDATA
000000          .WORD   1,0,1,0
000010          .WORD   0
000012          .WORD   120000
000014          .WORD   0,0
000020          .WORD   31004,0,0
000026          .WORD   1,0,1,0
000036          .WORD   0
000040          .WORD   120000
000042          .WORD   0,0
000046          .WORD   31004,0,0
000054          .WORD   1,0,1,0
000064          .WORD   0
000066          .WORD   120000
000070          .WORD   0,0
000074          .WORD   31004,0,0
000000          .PSECT  $CODE1
000000  MATMUL:
000000          MOV     @2(R5),-(SP)
000004          MOV     @2(R5),-(SP)
000010          MOV     4(R5),-(SP)
000014          MOV     #$IDATA+14,-(SP)
000020          JSR     PC,MKA2$
000024          MOV     @2(R5),-(SP)
000030          MOV     @2(R5),-(SP)
000034          MOV     6(R5),-(SP)
000040          MOV     #$IDATA+42,-(SP)
000044          JSR     PC,MKA2$
000050          MOV     @2(R5),-(SP)
000054          MOV     @2(R5),-(SP)
000060          MOV     10(R5),-(SP)
000064          MOV     #$IDATA+70,-(SP)
000070          JSR     PC,MKA2$
000074          MOV     #1,I
000102  LSFAGG:
000102          MOV     #1,J
000110  LSFAFI:
000110          SETF
000112          CLRF    S
000116          MOV     #1,K
000124          MOV     J,R1
000130          MUL     $IDATA+50,R1
000134          MOV     R1,$TEMP5
000140  LSFAAL:
000140          MOV     K,R1
000144          MUL     $IDATA+22,R1
000150          ADD     I,R1
000154          ASL     R1
000156          ASL     R1
000160          MOV     $TEMP5,R0
000164          ADD     K,R0
```

3

21

1

1

1

1

1

I-Stream

3

3

3

1

1

3

3

1

1

1

1

1

1

1

1

1

1

Fig. 2

55

RELEASED SYSTEM, SUPPORTED FOR FIELD TEST PURPOSES ONLY

```
000170          ASL     R0                      1
000172          ASL     R0                      1
000174          ADD     $IDATA+16,R1            3       1
000200          ADD     $IDATA+44,R0            3       1
000204          LDF     @R1,F0                  2
000206          MULF    @R0,F0                  2
000210          ADDF    $,F0                    3       1
000214          STF     F0,$                    3       1
000220          INC     K                       3       1
000224          CMP     K,@2(R5)                6       3
000232          BLE     LSFAAL                  1
000234          MOV     J,R1                    45      1
000240          MUL     $IDATA+76,R1                    1
000244          ADD     I,R1                            1
000250          ASL     R1
000252          ASL     R1
000254          ADD     $IDATA+72,R1                    1
000260          MOV     $,(R1)+                         1
000264          MOV     $+2,@R1                         1
000270          INC     J                               1
000274          CMP     J,@2(R5)                        3
000302          BLE     LSFAFI
000304          INC     I                               1
000310          CMP     I,@2(R5)                        3
000316          BLE     LSFAGG
000320          RTS     PC
                .GLOBL  $OTSV
                .END
```

SECTION	SIZE			
$CODE1	105	+ 55	=	160
$IDATA	33	+ 3	=	36
$VARS	5	+ 0	=	5
$TEMPS	~~1~~ 144	+ 1	=	2
				203

,LP:/LI:3=VECMUL./-T R

Fig. 3