

digital

INTEROFFICE MEMORANDUM

TO: Dragon PSG Members

DATE: October 24, 1974

0402

CC: Gordon Bell
Dick Clayton
Bruce Delagi
Bill Demmer

FROM: Jega Arulpragasam

DEPT: 11 Engineering

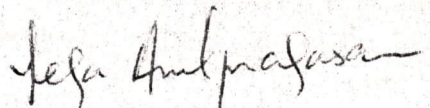
EXT: 5545 LOC: ML5/E54

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.



/br

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.

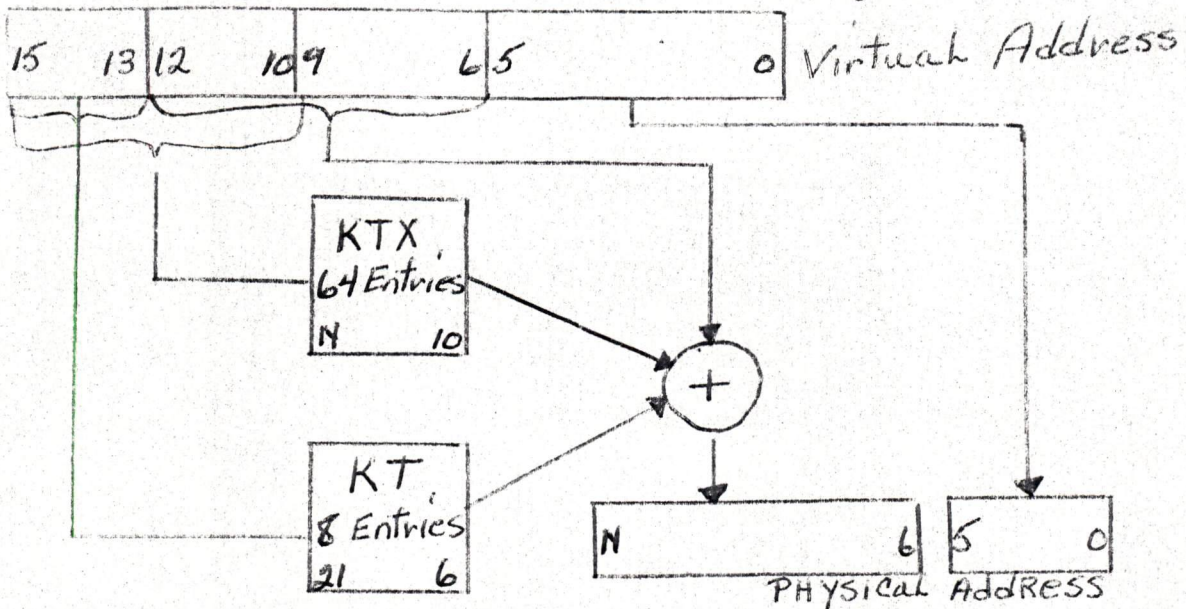
THE MAPPING MECHANISM.

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 instructions 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 R₁ S₂ S₂. 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, Auto-increment 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 0 (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.

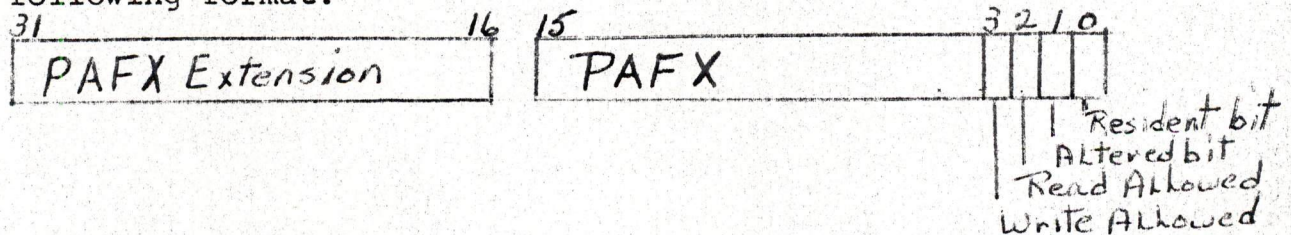
No modification to Craig's proposal is required. The mechanism will be identical in all respects with the segmented Chapter scheme in this regard, which does not infringe on linearity at all. This includes the use of PS bits 8 and 9, Subroutine Calling (including JSX) etc.

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:

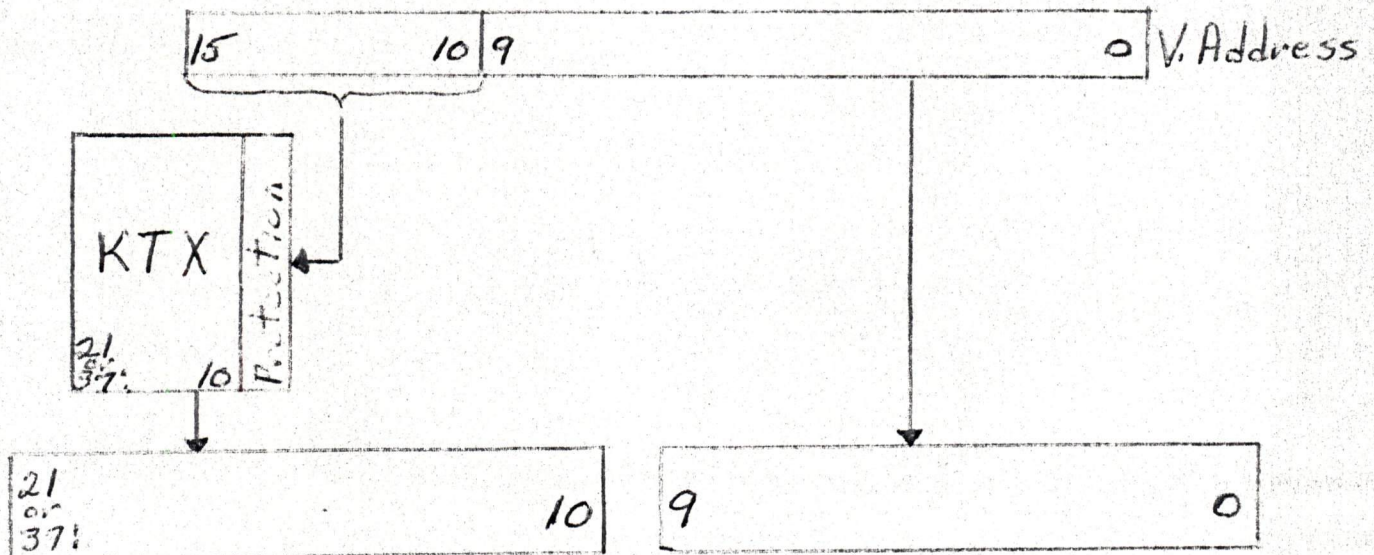
1. Without the PAFX Extension (normal case) the Physical Addressability is 22 bits (12 + 10) or 2 megawords.
2. Resident, Altered (Written Into) and Protection bits are provided to allow this resolution down to 1/2K page size level.
3. Access Rights etc., are therefore determined by the "AND" of those 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 single 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.

THE EVOLUTION OF VIRTUAL MACHINE ARCHITECTURE*

by J.P. Buzen and U.O. Gagliardi

Honeywell Information Systems, Inc.
Billerica, Massachusetts

and

Harvard University
Cambridge, Massachusetts

*This work was sponsored in part by the Electronic Systems Division,
U.S. Air Force, Hanscom Field, Bedford, Massachusetts under Contract
Number F19628-70-C-0217.

This paper will be presented at the AFIPS National Computer
Conference, June 4-8, 1973, New York, New York.

INTRODUCTION

In the early 1960's two major evolutionary steps were taken with regard to computing systems architecture. These were the emergence of I/O processors and the use of multiprogramming to improve resource utilization and overall performance. As a consequence of the first step computing systems became multiprocessor configurations where non-identical processors could have access to the common main memory of the system. The second step resulted in several computational processes sharing a single processor on a time-multiplexed basis while vying for a common pool of resources.

Both these developments introduced very serious potential problems for system integrity. An I/O processor executing an "incorrect" channel program could alter areas of main memory that belonged to other computations or to the nucleus of the software system. A computational process executing an "incorrect" procedure could cause similar problems to arise. Since abundant experience had demonstrated that it was not possible to rely on the "correctness" of all software, the multi-processing/multiprogramming architectures of the third generation had to rely on a completely new approach.

DUAL STATE ARCHITECTURE

The approach chosen was to separate the software into two classes: the first containing a relatively small amount of code which was presumed to be logically correct, the second containing all the rest. At the same time the system architecture was defined so that all functionality which

could cause undesirable interference between processes was strictly denied to the second class of software.

Essentially, third generation architectures created two distinct modes of system operation (privileged/non-privileged, master/slave, system/user, etc.) and permitted certain critical operations to be performed only in the more privileged state. The critical operations restricted to privileged state typically include such functions as channel program initiation, modification of address mapping mechanisms, direct monitoring of external interrupts, etc. Experience has shown that this solution can be quite effective if the privileged software is limited in quantity, is stable in the sense that few changes are made over long periods of time, and is written by skilled professional programmers.

While this architectural principle has proven its value by fostering the development of computing systems with true simultaneity of I/O operations and high overall resource utilization, it has generated a whole host of problems of its own. These problems arise from the fact that the only software which has complete access to and control of all the functional capabilities of the hardware is the privileged software nucleus.

Probably the most serious difficulty arises in the area of program transportability since non-privileged programs are actually written for the extended machine formed by the privileged software nucleus plus the non-privileged functions of the hardware. These extended machines are more difficult to standardize than hardware machines since it is relatively easy to modify or extend a system whose primitives are in part implemented

in software. This has frequently resulted in a multiplicity of extended machines running on what would otherwise be compatible hardware machines. A user who wishes to run programs from another installation which were written for a different extended machine is faced with either scheduling his installation to run the "foreign" software nucleus for some period of time or converting the programs to his installation's extended machine. Neither of these alternatives is particularly attractive in the majority of cases.

Another problem is that it is impossible to run two versions of the privileged software nucleus at the same time. This makes continued development and modification of the nucleus difficult since system programmers often have to work odd hours in order to have a dedicated machine at their disposal. In addition to the inconvenience this may cause, such procedures do not result in very efficient utilization of resources since a single programmer who is modifying or debugging a system from a console does not normally generate a very heavy load.

A final problem is that test and diagnostic software has to have access to and control of all the functional capabilities of the hardware and thus cannot be run simultaneously with the privileged software nucleus. This in turn severely curtails the amount of testing and diagnosis that can be performed without interfering with normal production schedules. The ever increasing emphasis on computer system reliability will tend to make this an even more serious problem in the future.

THE VIRTUAL MACHINE CONCEPT

Figure 1 illustrates the conventional dual state extended machine architecture which is responsible for all the difficulties that were cited in the preceding section. As can be seen in the Figure, the crux of the problem is that conventional systems contain only one basic machine interface* and thus are only capable of running one privileged software nucleus at any given time. Note, however, that conventional systems are capable of running a number of user programs at the same time since the privileged software nucleus can support several extended machine interfaces. If it were possible to construct a privileged software nucleus which supported several copies of the basic machine interface rather than the extended machine interface, then a different privileged software nucleus could be run on each of the additional basic machine interfaces and the problems mentioned in the preceding section could be eliminated.

*A basic machine interface is the set of all software visible objects and instructions that are directly supported by the hardware and firmware of a particular system.

A basic machine interface which is not supported directly on a bare machine but is instead supported in a manner similar to an extended machine interface is known as a virtual machine. As illustrated in Figure 2, the program which supports the additional basic machine interfaces is known as a virtual machine monitor or VMM. Since a basic machine interface supported by a VMM is functionally identical to the basic machine interface of the corresponding real machine, any privileged software nucleus

which runs on the bare machine will run on the virtual machine as well. Furthermore, a privileged software nucleus will have no way of determining whether it is running on a bare machine or on a virtual machine. Thus a virtual machine is, in a very fundamental sense, equivalent to and functionally indistinguishable from its real machine counterpart.

In practice no virtual machine is completely equivalent to its real machine counterpart. For example, when several virtual machines share a single processor on a time-multiplexed basis, the time dependent characteristics of the virtual and real machine are likely to differ significantly. The overhead created by the VMM is also apt to cause timing differences. A more significant factor is that virtual machines sometimes lack certain minor functional capabilities of their real machine counterparts such as the ability to execute self-modifying channel programs. Thus the characterization of virtual machines presented in the preceding paragraph must be slightly modified in many cases to encompass all entities which are conventionally referred to as virtual machines.

Perhaps the most significant aspect of virtual machine monitors is the manner in which programs running on a virtual machine are executed. The VMM does not perform instruction-by-instruction interpretation of these programs but rather allows them to run directly on the bare machine for much of the time. However, the VMM will occasionally trap certain instructions and execute them interpretively in order to insure the integrity of the system as a

whole. Control is returned to the executing program after the interpretive phase is completed. Thus program execution on a virtual machine is quite similar to program execution on an extended machine: the majority of the instructions execute directly without software intervention, but occasionally the controlling software will seize control in order to perform a necessary interpretive operation.

VIRTUAL MACHINES AND EMULATORS

Figure 2 is not intended to imply that the basic machine interface supported by the VMM must be identical to the interface of the bare machine that the VMM runs on. However, these interfaces often are identical in practice. When they are not, they are usually members of the same computer family as in the case of the original version of CP-67¹, a VMM which runs on an IBM 360 Model 67 (with paging) and supports a virtual IBM 360 Model 65 (without paging) beneath it.

When the two interfaces are distinctly different the program which supports the virtual interface is usually called an emulator² rather than a virtual machine monitor. Aside from this comparatively minor difference, virtual machines and emulators are quite similar in both structure and function. However, because they are not implemented with the same objectives in mind, the two concepts often give the appearance of being markedly different.

Virtual machine monitors are usually implemented without adding special order code translation firmware to the bare machine. Thus, most

VMM's project either the same basic machine interface or a restricted subset of the basic machines interface that they themselves run on. In addition, VMM's are usually capable of supporting several independent virtual machines beneath them since many of the most important VMM applications involve concurrent processing of more than one privileged software nucleus. Finally, VMM's which do project the same interface as the one they run on must deal with the problem of recursion (i.e., running a virtual machine monitor under itself). In fact, proper handling of exception conditions under recursion is one of the more challenging problems of virtual machine design.

Emulators, by contrast, map the basic machine interface of one machine onto the basic machine interface of another and thus never need be concerned with the problem of recursion. Another point of difference is that an emulator normally supports only one copy of a basic machine interface and thus does not have to deal with the scheduling and resource allocation problems which arise when multiple independent copies are supported. Still another implementation difference is that emulators must frequently deal with more complex I/O problems than virtual machine monitors do since the emulated system and the system that the emulator is running on may have very different I/O devices and channel architecture.

Modern integrated emulators³ exhibit another difference from the virtual machine monitor illustrated in Figure 2 in that an integrated emulator runs on an extended machine rather than running directly on a

ware machine. However, it is possible to create virtual machine monitors which also run on extended machines as indicated in Figure 3. Goldberg⁴ refers to such systems as Type II virtual machines. Systems of the type depicted in Figure 2 are referred to as Type I virtual machines.

It should be apparent from this discussion that virtual machines and emulators have a great deal in common and that significant interchange of ideas is possible. For a further discussion of this point, see Mallach⁵.

ADDITIONAL APPLICATIONS

It has already been indicated that virtual machine systems can be used to resolve a number of problems in program portability, software development, and "test and diagnostic" scheduling. These are not the only situations in which virtual machines are of interest, and in fact virtual machine systems can be applied to a number of equally significant problems in the areas of security, reliability and measurement.

From the standpoint of reliability one of the most important aspects of virtual machine systems is the high degree of isolation that a virtual machine monitor provides for each basic machine interface operating under its control. In particular, a programming error in one privileged software nucleus will not affect the operation of another privileged software nucleus running on an independent virtual machine controlled by the same monitor. Thus virtual machine monitors can localize and control the impact of operating system errors in much the same way that conventional systems localize and

control the impact of user program errors. In multiprogramming applications where both high availability and graceful degradation in the midst of failures are required, virtual machine systems can, for a large class of utility functions, be shown to have a quantifiable advantage over conventionally organized systems.⁶

The high degree of isolation that exists between independent virtual machines also makes these systems important in certain privacy and security applications^{7,8}. Since a privileged software nucleus has, in principle, no way of determining whether it is running on a virtual or a real machine, it has no way of spying on or altering any other virtual machine that may be coexisting with it in the same system. Thus the isolation of independent virtual machines is important for privacy and security as well as system reliability.

Another consideration of interest in this context is that virtual machine monitors typically do not require a large amount of code or a high degree of logical complexity. This makes it feasible to carry out comprehensive checkout procedures and thus insure high overall reliability as well as the integrity of any special privacy and security features that may be present.

The applications of virtual machines to the measurement of system behavior are somewhat different in nature. It has already been noted that existing virtual machine monitors intercept certain instructions for interpretive execution rather than allowing them to execute directly

on the bare machine. These intercepted instructions typically include I/O requests and most other supervisory calls. Hence, if it is desired to measure the frequency of I/O operations or the amount of supervisory overhead in a system, it is possible to modify the virtual machine monitor to collect these statistics and then run the system under that modified monitor. In this way no changes have to be made to the system itself. A large body of experimental data has been collected by using virtual machine monitors in this fashion^{9,10,11}.

EARLY VIRTUAL MACHINES

Virtual machine monitors for computers with dual state architecture first appeared in the mid 1960's. Early VMM's^{12,13} were most noteworthy for the manner in which they controlled the processor state, main memory and I/O operations of the virtual machines which ran under their control. This section presents a brief description and analysis of the special mapping techniques that were employed in these early systems.

Processor state mapping

The mapping of processor state was probably the most unusual feature of early virtual machine monitors. If a VMM did not maintain proper control over the actual state of the processor, a privileged software nucleus executing on a virtual machine could conceivably enter privileged mode and gain unrestricted access to the entire system. It would then be able to interfere at will with the VMM itself or with any other virtual machine present in the system. Since this is obviously an unacceptable situation, some mapping of virtual processor state to actual processor state was required.

The solution that was adopted involved running all virtual machine processes in the non-privileged state and having the virtual machine monitor maintain a virtual state indicator which was set to either privileged or non-privileged mode, depending on the state the process would be in if it were executing directly on the bare machine. Instructions which were insensitive to the actual state of the machine were then allowed to execute directly on the bare machine with no intervention on the part of the VMM. All other instructions were trapped by the VMM and executed interpretively, using the virtual system state indicator to determine the appropriate action in each case.

The particular instructions which have to be trapped for interpretive execution vary from machine to machine, but general guidelines for determining the types of instructions which require trapping can be identified¹⁴. First and most obvious is any instruction which can change the state of the machine. Such instructions must be trapped to allow the virtual state indicator to be properly maintained. A second type is any instruction which directly queries the state of the machine, or any instruction which is executed differently in privileged and non-privileged state. These instructions have to be executed interpretively since the virtual and actual states of the system are not always the same.

Memory mapping

Early virtual machine monitors also mapped the main memory addresses generated by processes running on virtual machines. This was necessary

because each virtual machine running under a VMM normally has an address space consisting of a single linear sequence that begins at zero. Since physical memory contains only one true zero and one linear addressing sequence, some form of address mapping is required in order to run several virtual machines at the same time.

Another reason for address mapping is that certain locations in main memory are normally used by the hardware to determine where to transfer control when an interrupt is received. Since most processors automatically enter privileged mode following an interrupt generated transfer of control, it is necessary to prevent a process executing on a virtual machine from obtaining access to these locations. By mapping these special locations in virtual address space into ordinary locations in real memory, the VMM can retain complete control over the actual locations used by the hardware and thus safeguard the integrity of the entire system.

Early VMM's relied on conventional paging techniques to solve their memory mapping problems. Faults generated by references to pages that were not in memory were handled entirely by the VMM's and were totally invisible to processes running on the virtual machines. VMM's also gained control after faults caused by references to addresses that exceeded the limits of a virtual machine's memory, but in this case all the VMM had to do was set the virtual state indicator to privileged mode and transfer control to the section of the virtual machine's privileged

software nucleus which normally handles out-of-bounds memory exceptions. These traps were thus completely visible to the software running on the virtual machine, and in a sense they should not have been directed to the VMM at all. More advanced virtual machine architectures permit these traps to be handled directly by the appropriate level of control. 15, 16

It should be noted that the virtual machines supported by early VMM's did not include paging mechanisms within their basic machine interfaces. In other words, only privileged software nuclei which were designed to run on non-paged machines could be run under these early virtual machine monitors. Thus these VMM's could not be run recursively.

I/O mapping

The final problem which early VMM's had to resolve was the mapping of I/O operations. As in the case of main memory addresses, there are a number of reasons why I/O operations have to be mapped. The primary reason is that the only addresses which appear in programs running on virtual machines are virtual (mapped) addresses. However, existing I/O channels require absolute (real) addresses for proper operation since timing considerations make it extremely difficult for channels to dynamically look up addresses in page tables as central processors do. Thus all channel programs created within a particular virtual machine must have their addresses "absolutized" before they can be executed.

The VMM performs this mapping function by trapping the instruction

which initiates channel program execution, copying the channel program into a private work area, absolutizing the addresses in the copied program, and then initiating the absolutized copy. When the channel program terminates, the VMM again gains control since all special memory locations which govern interrupt generated transfers are maintained by the VMM. After receiving the interrupt, the VMM transfers control to the address which appears in the corresponding interrupt dispatching location of the appropriate virtual machine's memory. Thus I/O completion interrupts are "reflected back" to the virtual machine in the same manner that out-of-bounds memory exceptions are.

One of the drawbacks of copying channel programs into private work areas and executing the absolutized copies is that channel programs which dynamically modify themselves during execution sometimes do not operate correctly. Hence it was not possible to execute certain self-modifying channel programs in early VMM's. However, since the majority of commonly used channel programs are not self-modifying, this lack of functionality could frequently be tolerated without serious inconvenience.

Channel program absolutization is not the only reason for VMM intervention in I/O operations. Intervention is also needed to maintain system integrity since an improperly written channel program can interfere with other virtual machines or with the VMM itself. The need for intervention

also arises in the case of communication with the operator's console. This communication must clearly be mapped to some other device since there is normally only one real operator's console in a system.

A final point is that VMM intervention in I/O operations makes it possible to transform requests for one device into requests for another (e.g., tape requests to disk requests) and to provide a virtual machine with devices which have no real counterpart (e.g., a disk with only five cylinders). These features are not essential to VMM operation, but they have proven to be extremely valuable by-products in certain applications.

Summary

In summary, early VMM's ran all programs in non-privileged mode, mapped main memory through paging techniques, and performed all I/O operations interpretively. Thus they could only be implemented on paged computer systems which had the ability to trap all instructions that could change or query processor state, initiate I/O operations, or in some manner be "sensitive" to the state of the processor¹⁴. Note that paging per se is not really necessary for virtual machine implementation, and in fact any memory relocation mechanism which can be made invisible to non-privileged processes will suffice. However, the trapping of all sensitive instructions in non-privileged mode is an absolute requirement for this type of virtual machine architecture. Since very few systems provide all the necessary traps, only a limited number of these VMM's have actually been constructed^{12,13,17,19}.

PAGED VIRTUAL MACHINES

It has already been noted that early VMM's did not support paged virtual machines and thus could not be run on the virtual machines they created. This lack of a recursive capability implied that VMM testing and development had to be carried out on a dedicated processor. In order to overcome this difficulty and to achieve a more satisfying degree of logical completeness, CP-67 was modified so that it could be run recursively¹⁸.

The major problem which had to be overcome was the efficient handling of the additional paging operation that took place within the VMM itself^{18,20}. To put the problem in perspective, note that early VMM's used their page tables to map addresses in the virtual machine's memory into addresses in the real machine's memory. For example, virtual memory address A' might be mapped into real memory address A". However, processes running on paged virtual machines do not deal with addresses which refer directly to the virtual machine's memory the way address A' does. Rather, an address A used by such a process must be mapped into an address such as A' by the page table of the virtual machine. Thus, in order to run a process on a paged virtual machine, a process generated address A must first be mapped into a virtual machine memory address A' by the virtual machine's page table, and then A' must be mapped into a real address A" by the VMM's page table.

In order to carry out this double mapping efficiently, the VMM constructs a composed page table (in which virtual process address A is mapped into real address A") and executes with this map controlling the address translation hardware. When the VMM transfers a page out of

memory, it must first change its own page table and then recompute the composed map. Similarly, if the privileged software nucleus changes the virtual machine's page table, the VMM must be notified so that the composed map can be recomputed.

This second consideration poses some difficulties. Since the virtual machine's page tables are stored in ordinary (virtual) memory locations, instructions which reference the tables are not necessarily trapped by the VMM. Thus changes could theoretically go undetected by the VMM. However, any change to a page table must in practice be followed by an instruction to clear the associative memory since the processor might otherwise use an out of date associative memory entry in a subsequent reference. Fortunately, the instruction which clears the associative memory will cause a trap when executed in non-privileged mode and thus allow the VMM to recompute the composed page table. Therefore, as long as the privileged software nucleus is correctly written, the operation of a virtual machine will be identical to the operation of the corresponding real machine. If the privileged software nucleus fails to clear the associative memory after changing a page table entry, proper operation cannot be guaranteed in either case.

TYPE II VIRTUAL MACHINES

VMM's which run on an extended machine interface are generally easier to construct than VMM's which run directly on a bare machine. This is because Type II VMM's can utilize the extended machine's instruction repertoire when carrying out complex operations such as I/O. In

addition, the VMM can take advantage of the extended machine's memory management facilities (which may include paging) and its file system. Thus Type II virtual machines offer a number of implementation advantages.

Processor state mapping

Type II virtual machines have been constructed for the extended machine interface projected by the UMMPS operating system²¹. UMMPS runs on an IBM 360 Model 67, and thus the VMM which runs under UMMPS is able to utilize the same processor state mapping that CP-67 does. However, the instruction in the VMM which initiates operation of a virtual machine must inform UMMPS that subsequent privileged instruction traps generated by the virtual machine should not be acted on directly but should instead be referred to the VMM for appropriate interpretation.

Memory mapping

The instruction which initiates operation of a virtual machine also instructs UMMPS to alter its page tables to reflect the fact that a new address space has been activated. The memory of the virtual machine created by the VMM is required to occupy a contiguous region beginning at a known address in the VMM's address space. Thus UMMPS creates the page table for the virtual machine simply by deleting certain entries from the page table used for the VMM and then subtracting a constant from the remaining virtual addresses so the new address space begins at zero. If the virtual machine being created is paged, it is then necessary to compose the resulting table with the page table that appears in the memory of the virtual machine. This latter operation is completely analogous to the creation of paged virtual machines under CP-67.

I/O mapping

I/O operations in the original UMMPS Type II virtual machine were handled by having UMMPS transfer control to the VMM after trapping the instruction which initiated channel program execution. The VMM translated the channel program into its address space by applying the virtual machine's page map if necessary and then adding a constant relocation factor to each address. After performing this translation the VMM called upon UMMPS to execute the channel program. UMMPS then absolutized the channel program and initiated its execution.

In addition to the overhead it entailed, this mapping procedure made it impossible for the virtual machine to execute a self-modifying channel program. A recent modification to the UMMPS virtual machine monitor has been able to alleviate this situation²². This modification involves positioning the virtual machine's memory in real memory so that the virtual and real address of each location is identical. This eliminates the need for channel program absolutization and thus improves efficiency while at the same time making self-modification of channel programs possible.

One of the difficulties that had to be overcome when making this change to the VMM was that the real counterparts of certain virtual machine memory locations were already being used by UMMPS. The solution that was adopted was to simply re-write the virtual machine's privileged software nucleus so that most of these locations were never used. A more detailed discussion of this point is provided by Srodawa and Bates²². Parmelee¹¹ describes a similar modification that has been made to CP-67.

SINGLE STATE ARCHITECTURE

One of the more unusual approaches to the problem of creating virtual machine architectures is based on the idea of eliminating privileged state entirely^{15,23}. The proponents of this approach argue that the primary -- and in fact only essential -- function of privileged state is to protect the processor's address mapping mechanism. If the address mapping mechanism were removed from the basic machine interface and thereby made totally invisible to software, there would be no need to protect the mechanism and therefore no need for privileged state.

In these single state architectures all software visible addresses are relative addresses and the mechanism for translating these relative addresses to absolute addresses always concealed. That is, each software level operates in an address space of some given size and structure but has no way of determining whether its addresses correspond literally to real memory addresses or whether they are mapped in some fashion. Since all addressing including I/O is done in this relative context, there is really no need for software to know absolute address and thus no generality is lost.

The central feature of this architecture is the manner in which software level N creates the address space of software level N+1. Basically, level N allocates a portion of its own address space for use by level N+1. The location of the address space of level N+1 is thus specified in terms of its relative address within level N. After defining the new address space, the level N software executes a special transfer of control instruction which changes the address mapping mechanism so that addresses will be

translated relative to the new address space. At the same time, control passes to some location within that new space.

Note that this special instruction need not be privileged since by its nature it may only allocate a subset of the resources it already has access to. Thus it cannot cause interference with superior levels. Level N can protect itself from level $N+1$ by defining the address space of level $N+1$ so that it does not encompass any information which level N wishes to keep secure. In particular, the address map that level N sets up for level $N+1$ is excluded from level $N+1$'s address space.

When an addressing fault occurs, the architecture traps back to the next lower level and adjusts the address map accordingly. Thus the system must retain a complete catalog of all active maps and must be able to compose and decompose them when necessary. This is relatively easy to do when only relocation/bounds maps are permitted¹⁵ but more difficult when segmentation is involved²³.

Since each level sees the same bare machine interface except for a smaller address space, each level corresponds to a new virtual machine. Mapping of processor state is unnecessary, mapping of memory is defined by the level N VMM relative to its own address space and is completely invisible to level $N+1$, and mapping of I/O is treated as a special case of mapping of memory. The two published reports on this architecture are essentially preliminary documents. More details have to be worked

out before a complete system can be defined.

THE VIRTUAL MACHINE FAULT

The single state architecture discussed in the preceding section provides a highly efficient environment for the creation of recursive virtual machine systems. However, the basic machine interface associated with this architecture lacks a number of features which are useful when writing a privileged software nucleus. These features, which are present to varying degrees in several late third generation computer systems, include descriptor based memory addressing, multi-layered rings of protection and process synchronization primitives.

A recent analysis²⁴ of virtual machine architectures for these more complex systems is based on an important distinction between two different types of faults. The first type is associated with software visible features of a basic machine interface such as privileged/non-privileged status, address mapping tables, etc. These faults are handled by the privileged software nucleus which runs that interface. The second type of fault appears only in virtual machine systems and is generated when a process attempts to alter a resource map that the VMM is maintaining or attempts to reference a resource which is available on a virtual machine but not the real system (e.g., a virtual machine memory location that is not in real memory). These faults are handled solely by the VMM and are completely invisible to the virtual machine itself.*

*Faults caused by references to unavailable real resources were not clearly identified in this paper. The distinctions being drawn here are based on a later analysis by Goldberg¹⁶.

Since conventional architectures support only the former type of fault, conventional VMM's are forced to map both fault types onto a single mechanism. As already noted, this is done by running all virtual machine processes in non-privileged mode, directing all faults to the VMM, and having the VMM "reflect" all faults of the first type back to the privileged software nucleus of the virtual machine.

An obvious improvement to this situation can be realized by creating an architecture which recognizes and supports both types of faults. A preliminary VMM design for a machine with this type of architecture has been proposed²⁴. The design relies on static composition of all resource maps and thus requires a trap to the VMM each time a privileged process attempts to alter a software visible map. However, the privileged/non-privileged distinction within a virtual machine is supported directly by the bare machine and a privileged process is allowed to read all software visible constructs (e.g., processor state) without generating any type of fault. The major value of this design is that it can be implemented on an existing system by making only a relatively small number of hardware/firmware modifications.

DYNAMIC MAP COMPOSITION - THE HARDWARE VIRTUALIZER

The clear distinction between virtual machine faults (handled by the VMM) and process exceptions (handled by the privileged software nucleus of the virtual machine) first appeared in a Ph.D. thesis by Goldberg¹⁶. One of the essential ideas of the thesis is that the various resource maps which have to be invoked in order to run a process

on a virtual machine should be automatically composed by the hardware and firmware of the system. Since map composition takes place dynamically, this proposal eliminates the need to generate a virtual machine fault each time a privileged process running on a virtual machine alters a software visible map. Thus the only cause of a virtual machine fault is a reference to a resource that is not present in a higher level virtual or real machine.

The thesis contains a detailed description of a "hardware virtualizer" which performs the map composition function. It includes a description of the virtualizer itself, the supporting control mechanisms, the instructions used for recursive virtual machine creation, and the various fault handling mechanisms. These details will not be considered here since they are treated in a companion paper²⁵.

It is interesting to note that the work on single state architecture^{15,23} can be regarded as a special case of the preceding analysis in which process exceptions caused by privileged state are completely eliminated and only virtual machine faults remain. Similarly, the earlier work of Gagliardi and Goldberg²⁴ represents another special case in which map composition is carried out statically by the VMM and where additional virtual machine faults are generated each time a component of the composite map is modified. By carefully identifying the appropriate functionality and visibility of all the maps involved in virtual machine operation, Goldberg's later analysis provides a highly valuable model for the design of virtual machine architectures and for the analysis of additional problems in this area.

CONCLUSION

A number of issues related to the architecture and implementation of virtual machine systems remain to be resolved. These include the design of efficient I/O control mechanisms, the development of techniques for sharing resources among independent virtual machines, and the formulation of resource allocation policies that provide efficient virtual machine operation. Many of these issues were addressed at the ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems held recently at Harvard University's Center for Research in Computing Technology*.

*Proceedings²⁶ may be ordered from ACM Headquarters in New York City.

In view of the major commitment of at least one large computer manufacturer to the support of virtual machine systems²⁷, the emergence of powerful new theoretical insights, and the rapidly expanding list of applications, one can confidently predict a continuing succession of virtual machine implementations and theoretical advances in the future.

ACKNOWLEDGEMENT

We would like to express our appreciation to Dr. R. P. Goldberg for generously providing us with much of the source material that was used in the preparation of this paper.

REFERENCES

0386

- 1 IBM CORPORATION
Control program-67/Cambridge monitor system
IBM Type III release no 360D-05.2.005 IBM Program Information Department
Hawthorne New York
- 2 E G MALLACH
Emulation: a survey.
Honeywell Computer Journal Vol 6 No 4 1973
- 3 G ALLRED
System/370 integrated emulation under OS and DOS
Proceedings AFIPS Spring Joint Computer Conference 1971
- 4 R P GOLDBERG
Virtual Machines: semantics and examples
Proceedings IEEE International Computer Society
Conference Boston Massachusetts 1971
- 5 E G MALLACH
On the relationship between emulators and virtual machines
Proceedings ACM SIGOPS-SIGARCH Workshop on Virtual Computer
Systems Cambridge Massachusetts 1973
- 6 J P BUZEN P P CHEN R P GOLDBERG
Virtual machine techniques for improving software reliability
Proceedings IEEE Symposium on Computer Software Reliability New York 1973
- 7 C R ATTANASIO
Virtual machines and data security
Proceedings ACM SIGOPS-SIGARCH Workshop on Virtual Computer
Systems Cambridge Massachusetts 1973
- 8 S E MADNICK J J DONOVAN
Virtual machine approach to information system security and isolation
Proceedings ACM SIGOPS-SIGARCH Workshop on Virtual Computer
Systems Cambridge Massachusetts 1973
- 9 V CASAROSA
VHM: a virtual hardware monitor
Proceedings ACM SIGOPS-SIGARCH Workshop on Virtual Computer
Systems Cambridge Massachusetts 1973
- 10 Y SARD
Performance criteria and measurement for a time-sharing system
IBM Systems Journal Vol 10 No 3 1971
- 11 R P PARMELEE
Preferred virtual machines for CP-67
IBM Cambridge Scientific Center Report No G320-2068
- 12 R ADAIR R U BAYLES L W COMEAU R J CREASY
A virtual machine system for the 360/40
IBM Cambridge Scientific Center Report No G320-2007 1966
- 13 R A MEYER L H SEAWRIGHT
A virtual machine time-sharing system
IBM Systems Journal Vol 9 No 3 1970

- 14 R P GOLDBERG
Hardware requirements for virtual computer systems
Proceedings Hawaii International Conference on System Sciences
Honolulu Hawaii 1971
- 15 H C LAUER C R SNOW
Is supervisor-state necessary?
Proceedings ACM AICA International Computing Symposium Venice Italy 1972
- 16 R P GOLDBERG
Architectural principles for virtual computer systems
Ph.D. Thesis Division of Engineering and Applied Physics Harvard
University Cambridge Massachusetts 1972
- 17 D SAYRE
On virtual systems
IBM T J Watson Research Laboratory Yorktown Heights 1966
- 18 R P PARMELEE T I PETERSON C C TILLMAN D J HATFIELD
Virtual storage and virtual machine concepts
IBM Systems Journal Vol 11 No 2 1972
- 19 A AUROUX C HANS
Le concept de machines virtuelles
Revue Francaise d'Informatique et de Recherche Operationelle Vol 15
No B3 1968
- 20 R P GOLDBERG
Virtual machine systems
MIT Lincoln Laboratory Report No MS-2687 (also 28L-0036) Lexington
Massachusetts 1969
- 21 J HOGG P MADDEROM
The virtual machine facility - - how to fake a 360
University of British Columbia - - University of Michigan
Computer Center Internal Note
- 22 R J SRODAWA L A BATES
An efficient virtual machine implementation
Proceedings AFIPS National Computer Conference 1973
- 23 H C LAUER D WYETH
A recursive virtual machine architecture
Proceedings ACM SIGOPS-SIGARCH Workshop on Virtual Computer
Systems Cambridge Massachusetts 1973
- 24 U O CAGLIARDI R P GOLDBERG
Virtualizeable architectures
Proceedings ACM AICA International Computing Symposium Venice Italy 1972
- 25 R P GOLDBERG
Architecture of virtual machines
Proceedings AFIPS National Computer Conference 1973

- 26 R. P. GOLDBERG EDITOR
Proceedings ACM SIGOPS-SIGARCH Workshop on Virtual Computer
Systems Cambridge Massachusetts 1973
- 27 IBM CORPORATION
IBM virtual machine facility/370: planning guide
Publication Number GC20-1801-0 1972

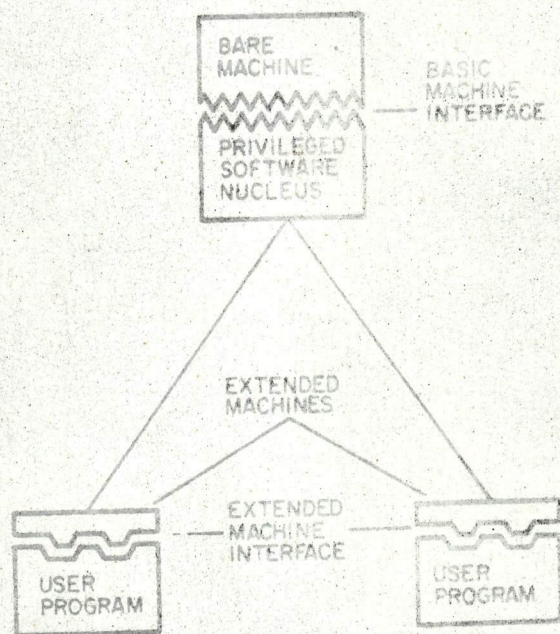


Figure 1
Conventional Extended Machine Organization

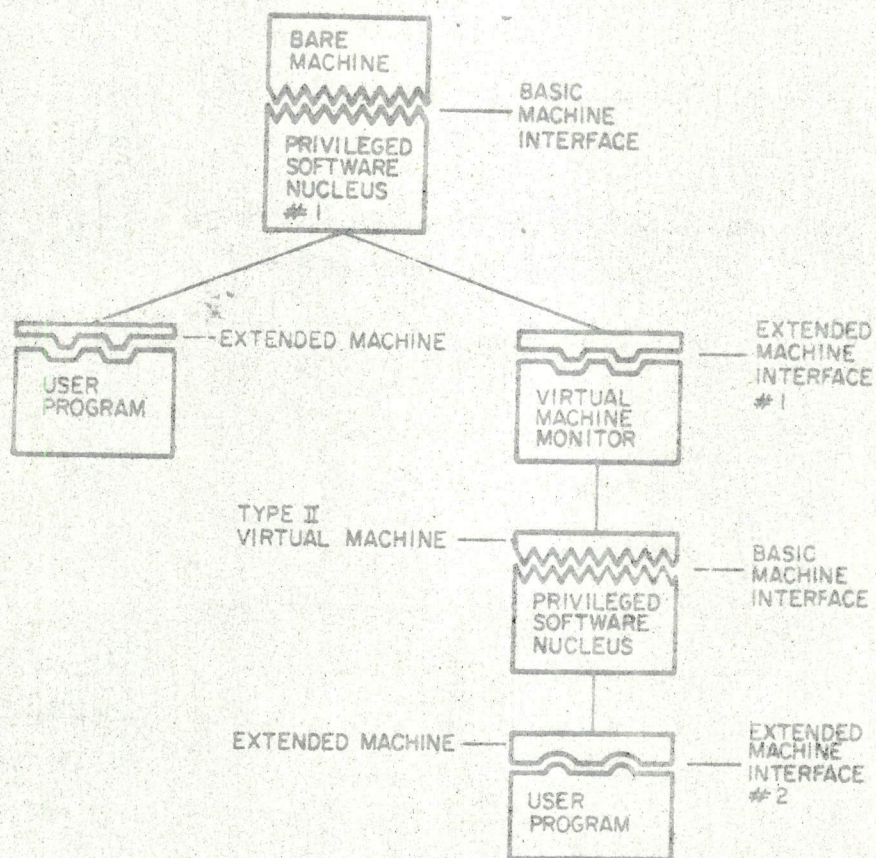


Figure 3
Type II Virtual Machine Organization

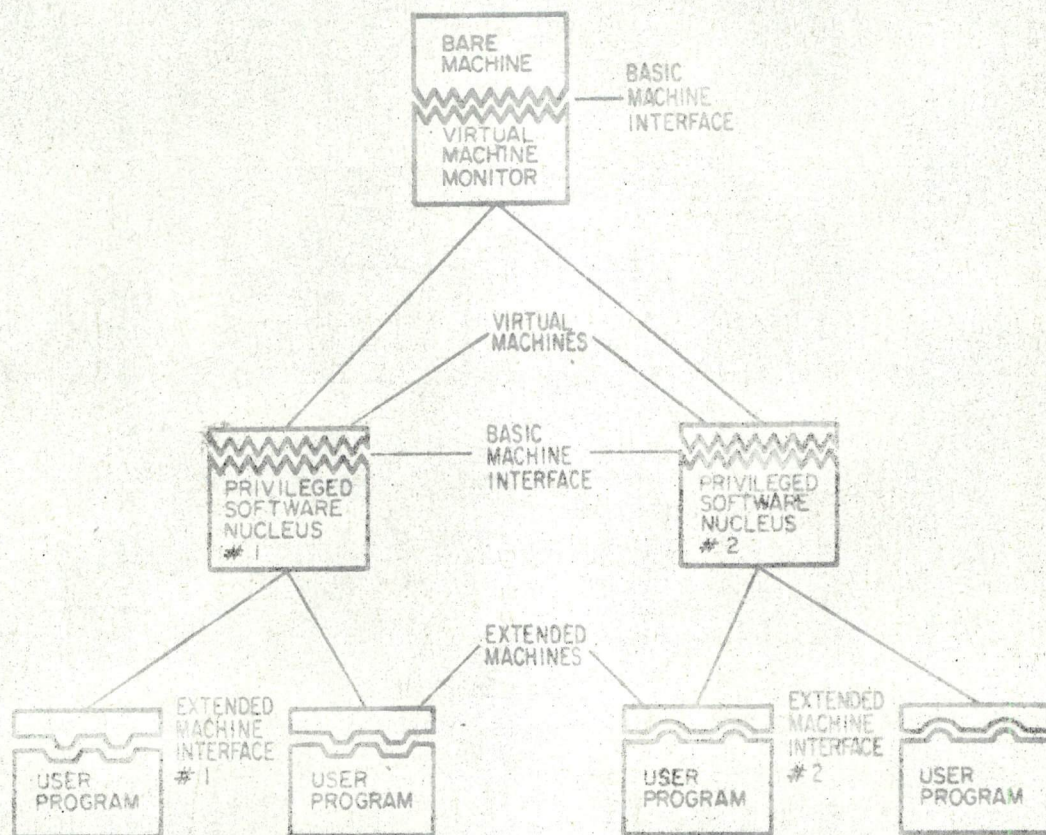


Figure 2
Virtual Machine Organization

digital INTEROFFICE MEMORANDUM

TO: Dragon PSG
 CC: Bill Strecker
 Craig Mudge
 Dave Nelson
 Ron Brender

DATE: October 25, 1974
 FROM: Bob Gray *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 consensus 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

FORTTRAN 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).

No new addressing modes.

Easy to explain at assem-
bly language level.

Name space capability un-
likely to be supported in
DEC software.

FORTTRAN COMMON data areas
limited to 32K.

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
 Gordon Bell
 Ron Brender
 John Buckley

Janice Carnes
 Len Hughes
 John Jones
 Bill McBride

Bill Strecker
 Pete Van Roekens
 Larry Wade
 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.

OCT 10 1974

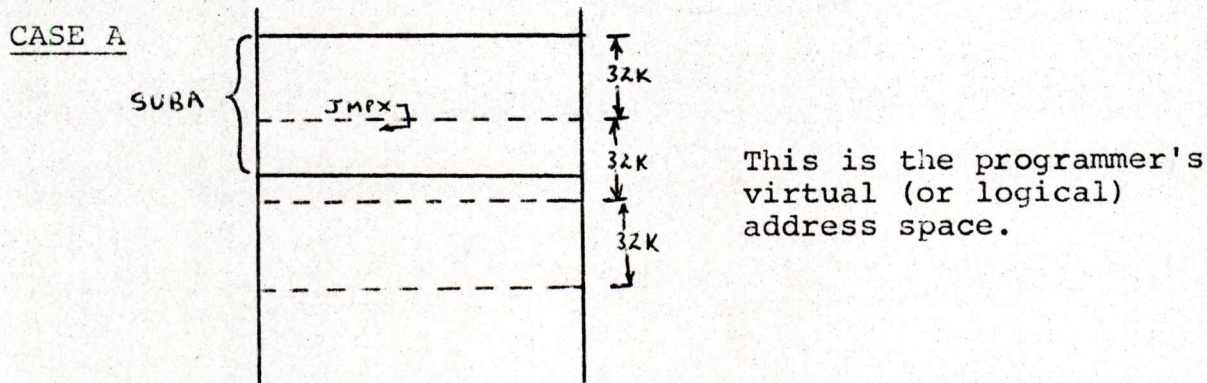
We have done more work on the implications of a segmented address space for FORTRAN EQUIVALENCE. We have found that the constraints on storage of large data areas are more severe than stated in the Mudge memo of 9/27/74. This follows from the observation that COMMON areas are in fact an implied EQUIVALENCE 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?"

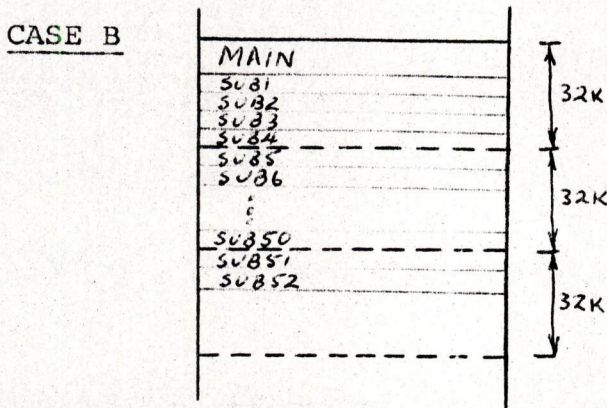
If a routine (subroutine or main program) in a program is 32K then it must explicitly recognize the boundary. It must jump over it. Thus, if we have a 50K subroutine, SUBA, i.e. 50K of instructions, (no data - data are outside of the program chapter), we have



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 50 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:



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

INTEGER BIGVI (32768)
and REAL BIGVR (16384).

0413

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,

$I \times \text{column dimension} + J,$
giving a one-component 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 instruc-
tion stream words in the inner loop of a matrix multiplication
subroutine.

However, most of the time at least one of the above require-
ments 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 con-
sidered to be so high that it would not be acceptable in prac-
tice 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
Gordon Bell
Ron Brender
John Buckley

Janice Carnes
Len Hughes
John Jones
Bill McBride
Robin Frith

Bill Strecker
Pete VanRoekens
Larry Wade
Dave Nelson
Prod. Line Mgrs.

Gordon Bell
12-1

digital INTEROFFICE MEMORANDUM

0415

TO: Distribution

DATE: October 8, 1974

FROM: Jega Arulpragasam

DEPT: 11 Engineering

EXT: 5545 LOC: 5-5

OCT 10 1974

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 Auto-increment 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 ($PS\langle 08 \rangle = 1$), 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 $PS\langle 08 \rangle = 1$, 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 32×16 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 $PS\langle 08 \rangle = 1$, 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 ll 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

Gordon Bell
12-1



INTEROFFICE MEMORANDUM

0420

OCT 03 1974

TO: Robin Frith
 CC: Distribution

DATE: September 27, 1974
 FROM: Craig Mudge
 DEPT: 11 Engineering
 EXT: 5064 LOC: 5/E54

SUBJ: 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.

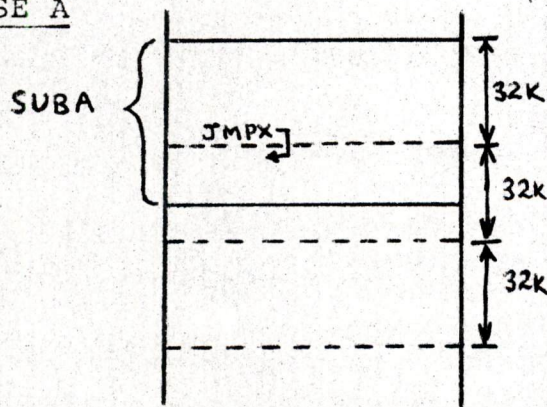
The 32-bit address in 11/VAX is a two-component address (c,d). c, the chapter number, and d, the displacement within chapter, are each 16 bits. 11/VAX purposely treats each 32-bit address as a two-component entity, principally for compatibility with today's 11. Thus the total address space is 2^{16} 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 explicitly 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



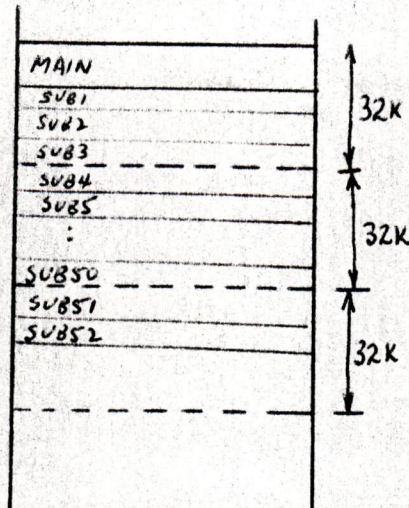
This is the programmer's virtual (or logical) address space.

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:

CASE B

0421



MACRO and FORTRAN:

Case A: The FORTRAN programmer does not concern himself with the boundary. The compiler will insert the JMPX.

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(100,000) 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 explicitly deal with the four parts of the vector.

For 16-bit integers and 32-bit floating point numbers the maximum vector sizes would be

```
and      INTEGER BIGVI (32768)
        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)
```

For a matrix if the total 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
Gordon Bell
Ron Brender
John Buckley

Janice Carnes
Len Hughes
John Jones
Bill McBride

Bill Strecker
Pete Van Roekens
Larry Wade
Product Line Managers
Dave Nelson

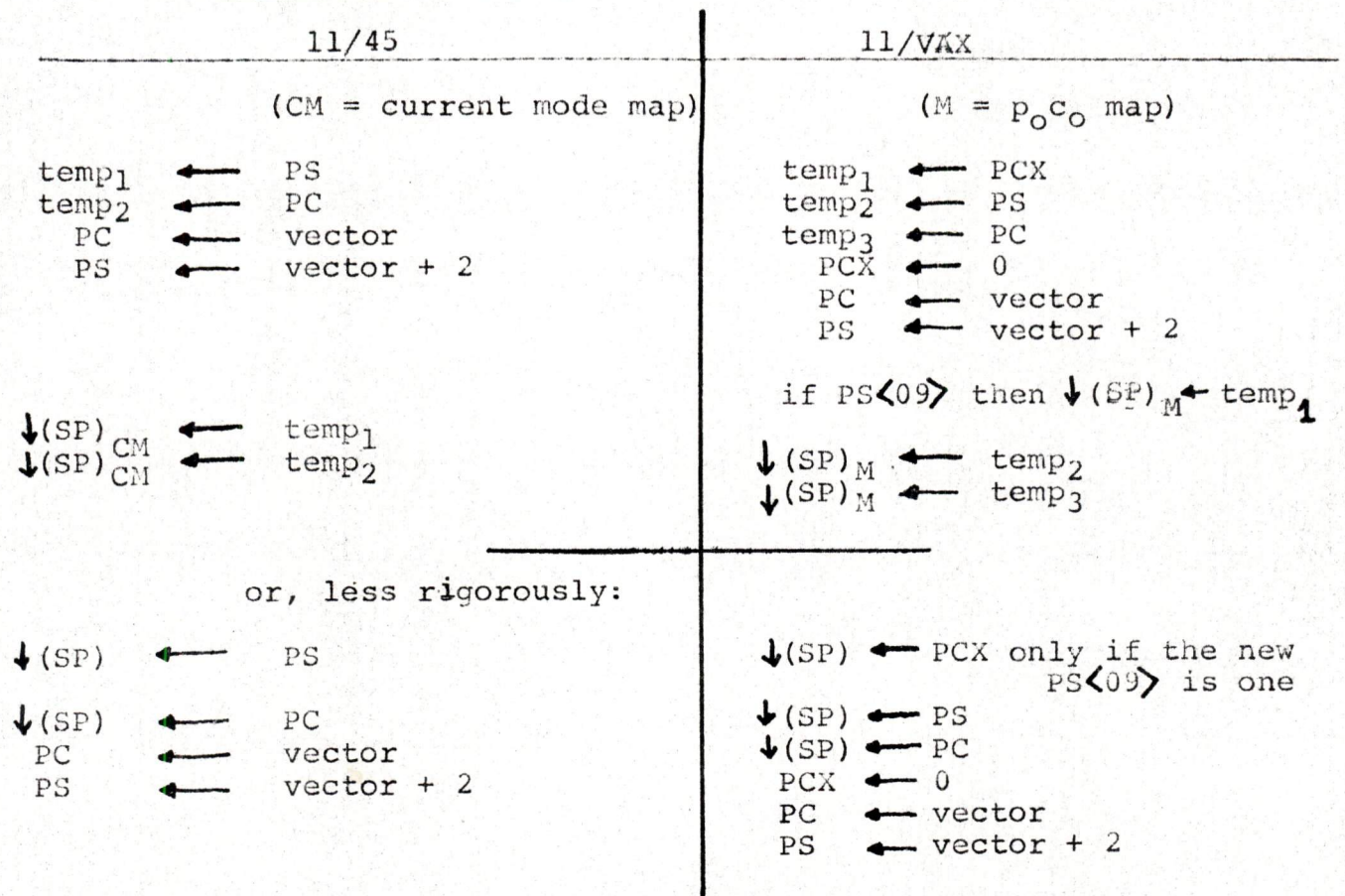
7th Sept 13, 1974 Version 2 Craig Mudge 9/74

0423

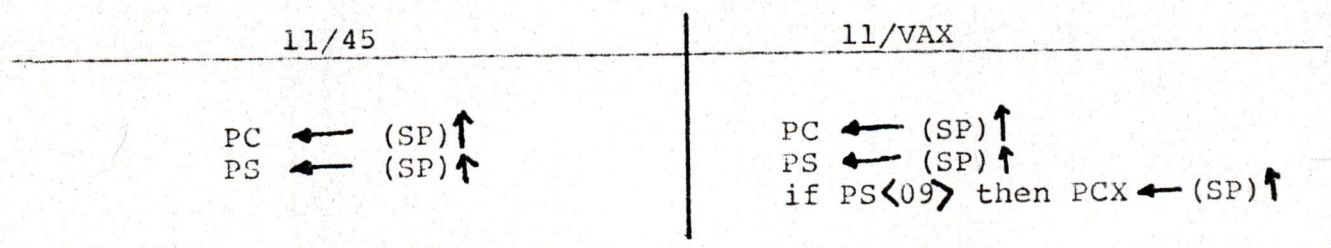
7. Interrupt Structure

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 KTL1 and is selected by the interrupt sequence.

a. Interrupts and traps:



b. RTI and RTT



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 <09>. 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
- II. Introduction
- III. Summary of the Chapter Scheme
- IV. The Chapter Scheme
 - 1. The Address Space
 - 2. Motivation for Segmentation
 - 3. 11/VX Registers
 - 4. Address Mapping
 - 5. X Mode
 - 6. New Instructions
 - 7. Interrupt Structure
 - 8. Process Dispatching and Context Switching
 - 9. New Error Conditions
- V. Compatibility
- VI. KT11-X--An Implementation
- VII. Design Decisions
- VIII. Programming Examples
- IX. Systems Software
- X. Previous Work

I. Terminology

11/VX - A machine architecture; PDP-11 architecture with an extended virtual address space using the chapter scheme.

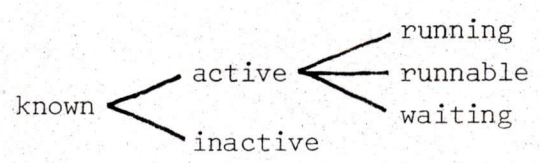
11/44 with - A particular implementation of 11/VX; the principal design goal of the KT11X option is to provide an extended VAS with minimum impact on the cost of the base 11/44 CPU.

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.

0447

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.

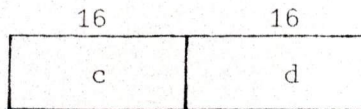
Design philosophy, assumptions.

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

0448

1. A program's VAS is a set of 64K-byte chapters.
2. A VA is of the form



c = chapter number
 d = displacement 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.

0449

- 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 (

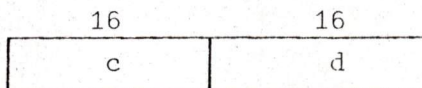
3	7	6
---	---	---

).
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.

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:



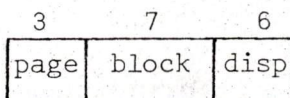
The chapters are independent (a carry out of the displacement field does not propagate into the chapter field).

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.

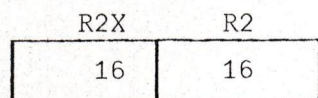
These four objectives, together with machine independence and list processing, are not peculiar to virtual memory systems. They were fought for in physical storage during the late 1950s [W5]. Dynamic storage allocation, linking and relocatable loaders [M3], relocation and base registers [D11], and now 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.

3. 11/VX Registers

Each general register is 32 bits wide. The low order 16 bits is called R_i , the high order R_iX . Thus, if R_2 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

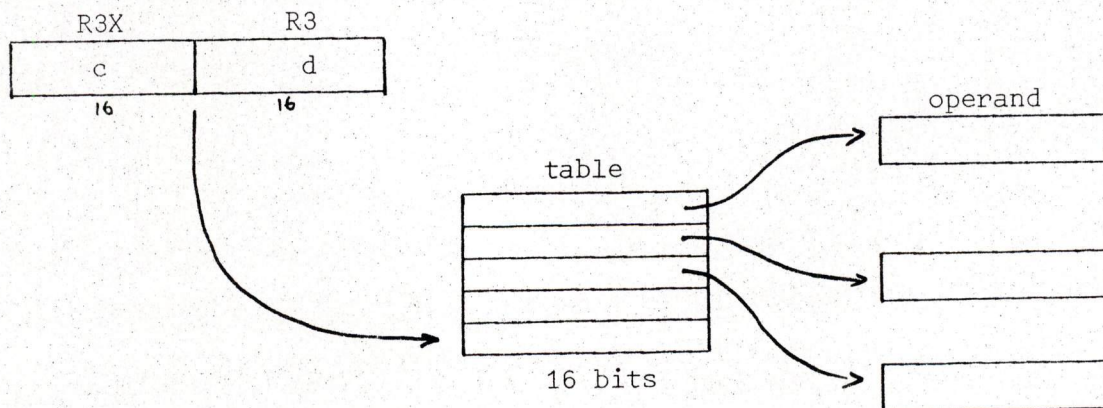
```

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 R_i . R_iX 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

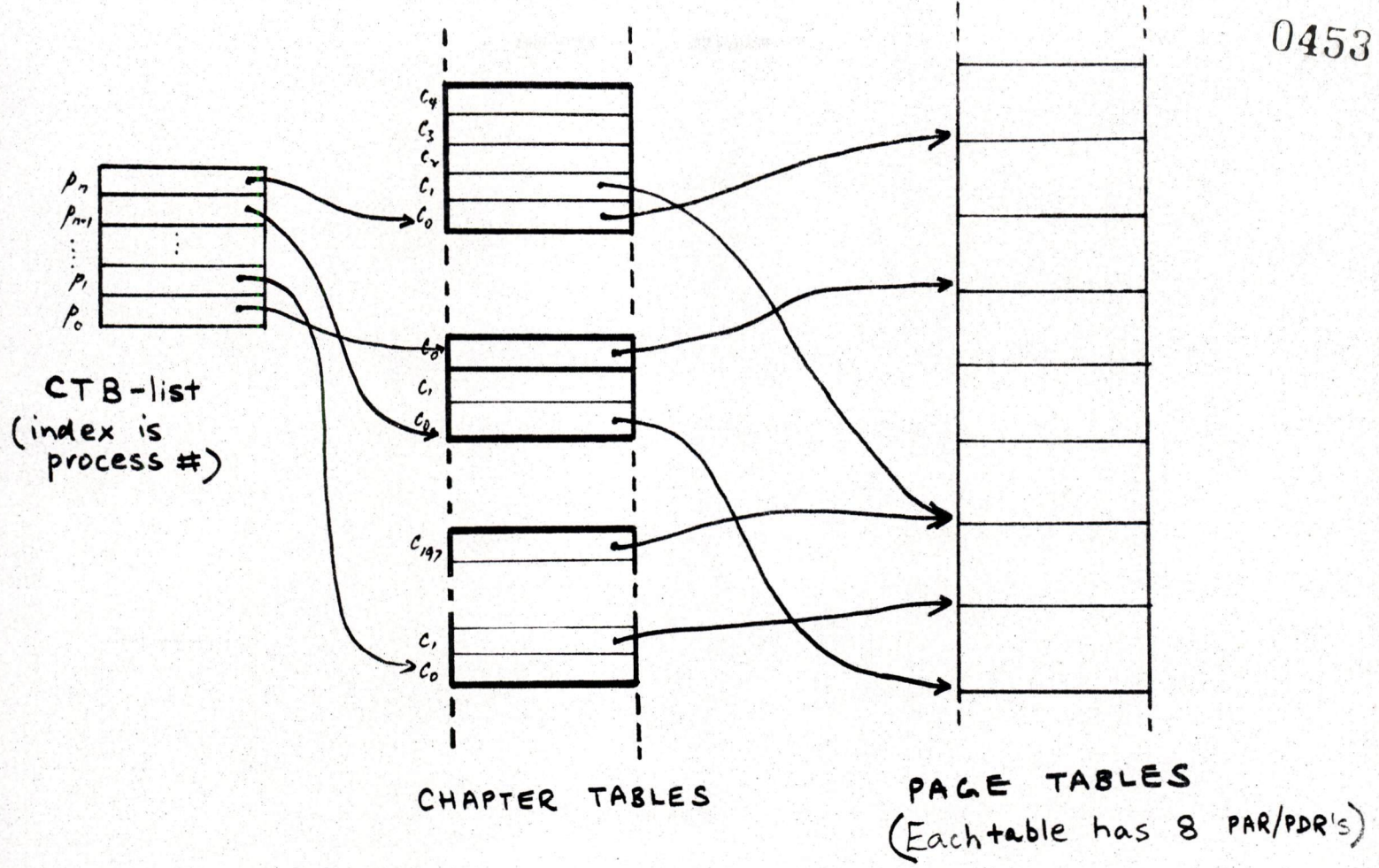
PS <08> holds the mode bit of the running process. When zero, PCX is used as the chapter number for any (16-bit) address generated by a non-X mode program. It is used to obtain compatibility. An example of its use is given in Section V. below.

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

0453



Format of table entries:

CTB-list entry

CTB	index
-----	-------

chapter table entry

CTS	index
-----	-------

page table entry

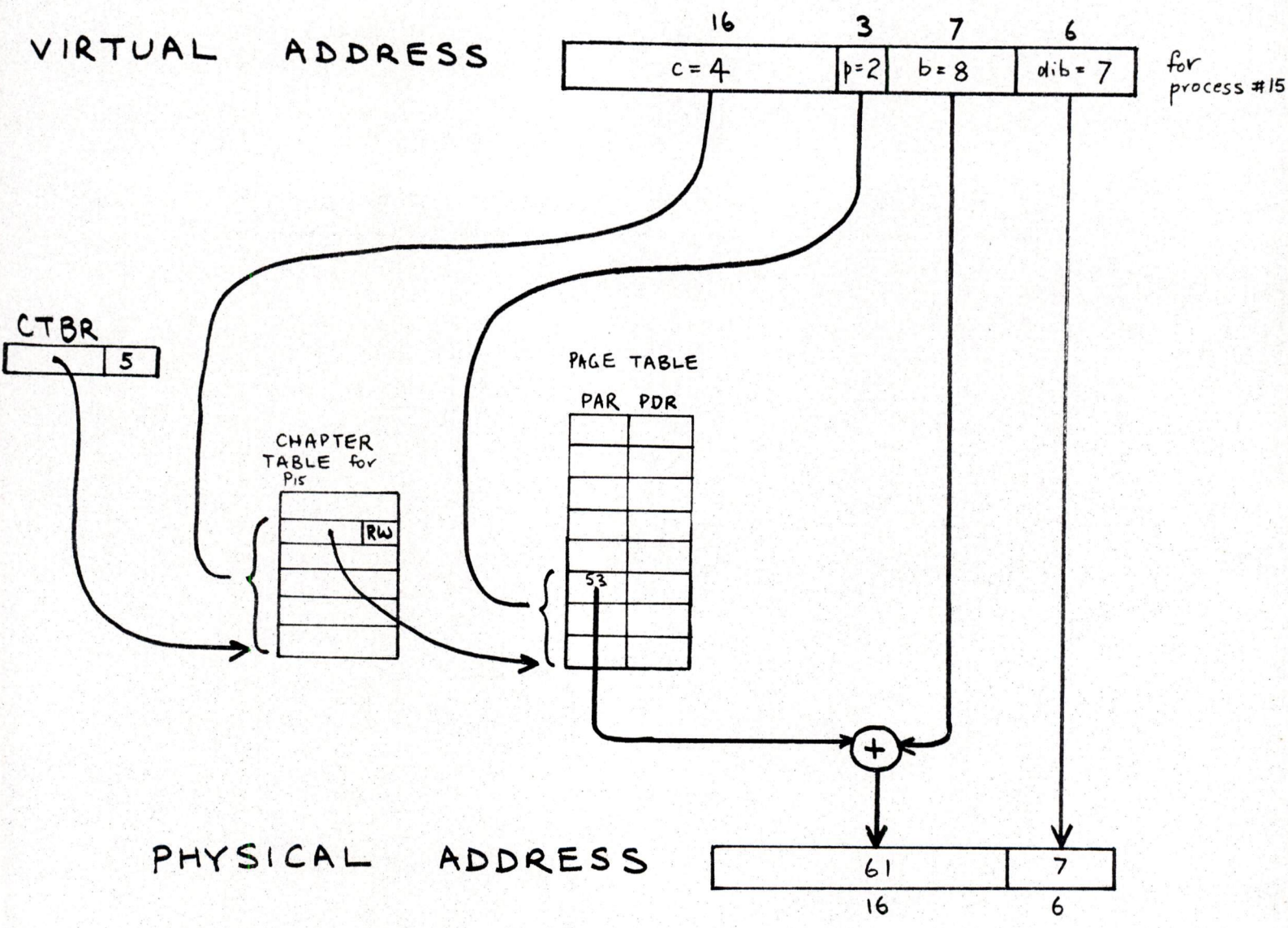
PAS	PDR
-----	-----

31-10-71
JEM

FIG 2 ADDRESS TRANSLATION EXAMPLE

0454

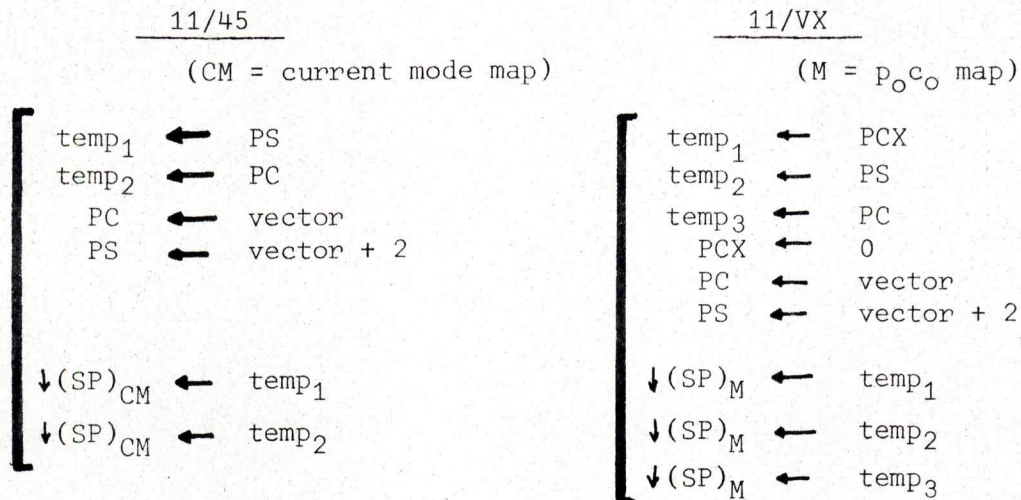
32-bit virtual to 22-bit physical



7. Interrupt Structure

All trap and interrupt vectors are in the p_0c_0 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:

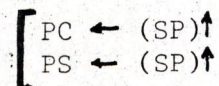


or, less rigorously:



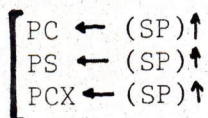
b. RTI/RTT:

→X mode (as determined by current PS)



Thus, RTI/RTT is an intra-chapter return.

X mode



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.

1. Introduction

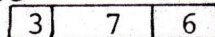
Typical problems encountered in extending the virtual address space are (1) specifying extended addresses in instructions, and (2) passing extended addresses between subroutines and between processes. It is necessary to examine where addresses are manipulated as operands - explicitly by program instructions, 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 K11 memory management scheme's structure



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.

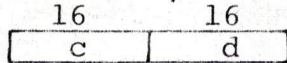
Class B: instructions whose domain is a multiple address space machine, e.g., MFPI.

Behaviour which is K, S, U-mode derived 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.

0458

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 ll 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

¹Note 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.

0459

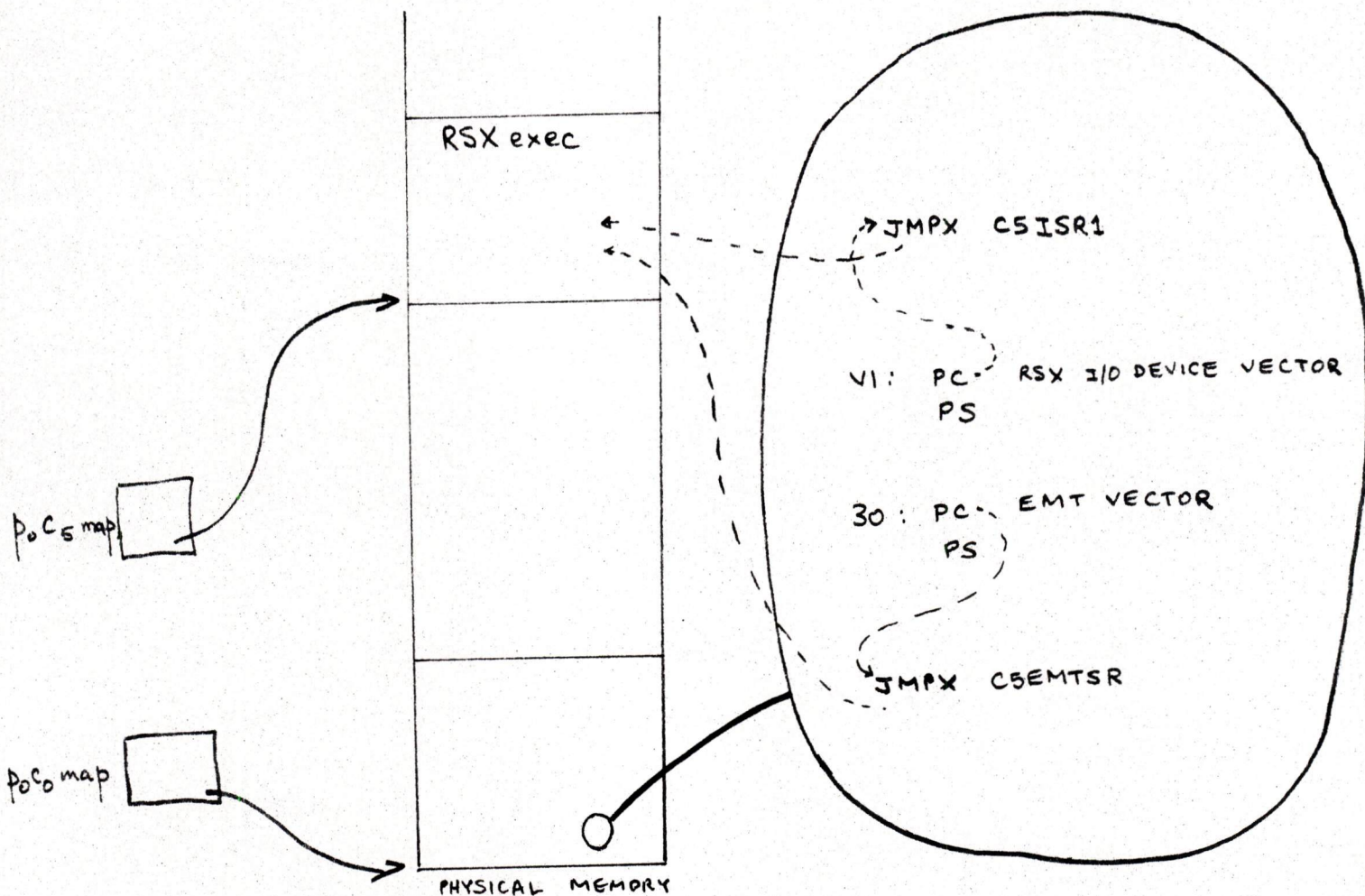
- 3. Class B instructions
 (This is a topic list only - most problems have been solved but text not yet written.)
 - a. K,S,U-mode derived address space selection is subsumed by chaptered space structure, except for Kernel space holding interrupt vectors. P₀KC₀ is explicitly recognized; page table always 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 0 connection section which loads PCX with the right chapter #.

Example 1

RSX11D exec in C₅

0460



Example 2

RSX11D exec in C₅

RSTS exec in C₇

No shared i/o devices simplification (more for resource management)

Sync traps, e.g., EMT, require the connection code in chapter 0 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

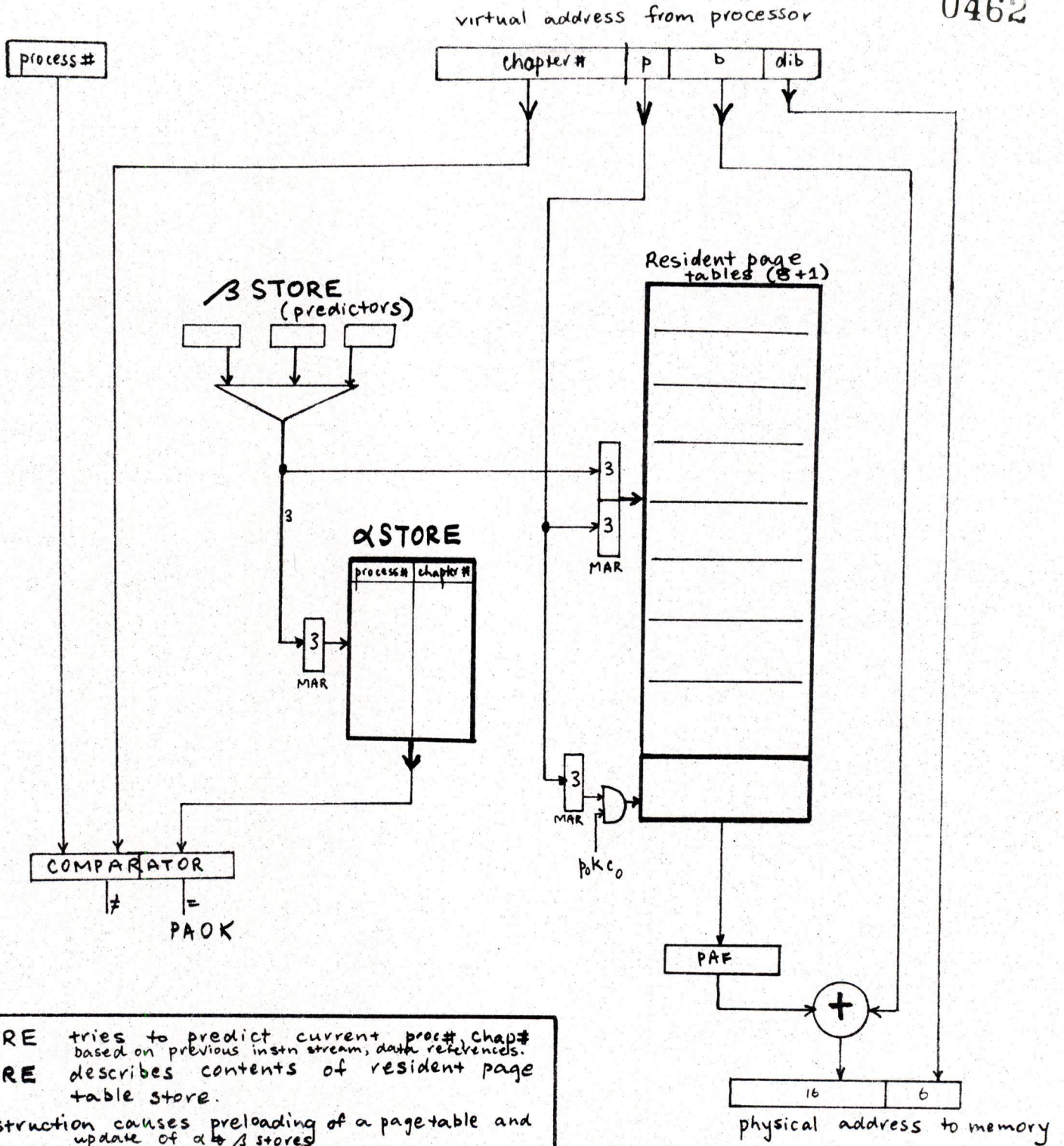
0461

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 R_iX , $i=1, \dots, 8$. 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.



① β STORE tries to predict current proc#, chap# based on previous instr 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;

0463

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 response.

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 ~~SP~~ (SP)+ and LA.

0464

VIII. Programming examples

1. FORTRAN array addressing
array > 32K words
2. passing parameters
a la old ll 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

0465

1. Atlas, Dennis, 360/67, Multics
2. (DEC) Strecker, Rodgers, Mudge, Burness

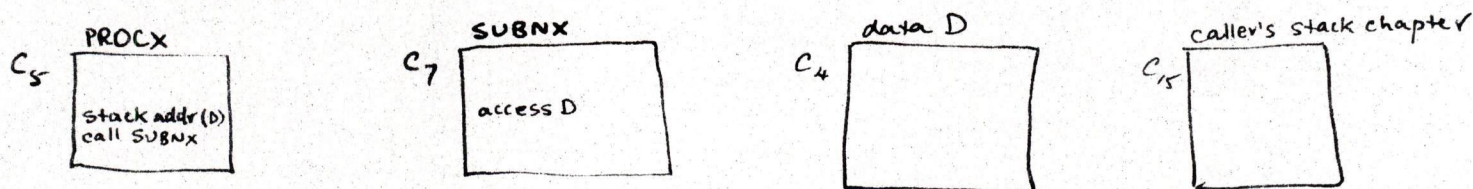
Acknowledgement: Ed Marison's contribution to 11/VX

X-mode program calling an existing non-x mode program

PROCX, the caller, ~~is~~ 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



C₇ is prefixed by an interface routine, IFR.

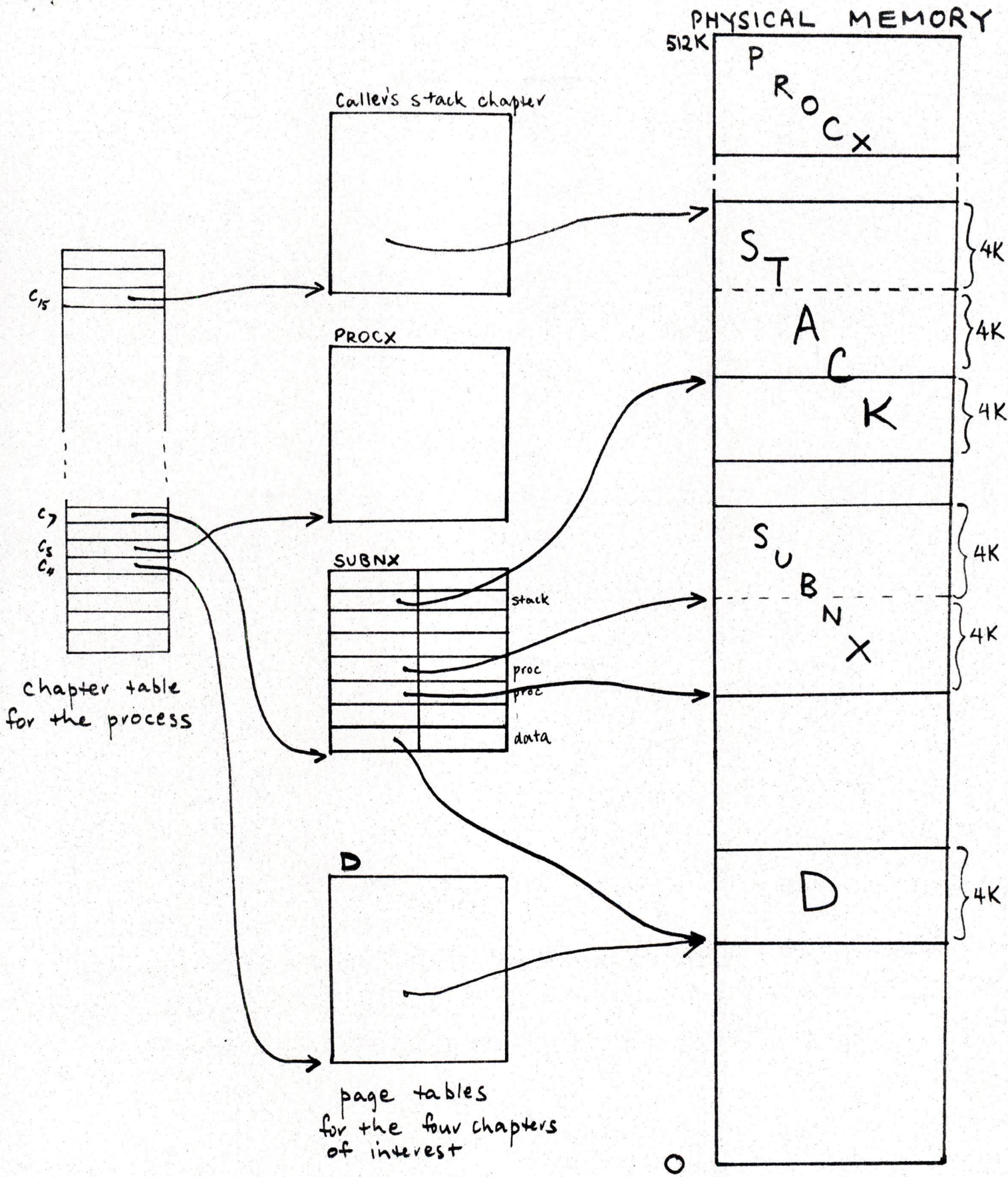
This IFR will switch the PS to non-x mode and so force all references to remain within C₇ (PCX had been set to 7 by the ~~IFR~~ interchapter jump from PROCX to SUBNX). All that remains is to ensure that data areas referenced by both PROCX and SUBNX, namely D and the stack, appear within C₇'s 16-bit VAS. This, of course, is done by mapping.

Figure 4 shows the mapping. The example is slightly more complicated than necessary ~~shown~~ — it shows SUBNX mapped to a subset of the caller's stack.

The IFR and JMPX to invoke SUBNX are as

MAPPING FOR PROCX CALLING SUBNX EXAMPLE

0467



follows:

```

C5 PROCX:
↓(SP) ← RA ; stack return address
Set up pavs
↓(SP) ← addr (SUBNX)
JMPX IFR; ; interchapter jump
RA:

```

```

C7 IFR: Mode ← TX
JSR PC, (SP)+
Mode ← X
JMPX (SP)+
SUBNX:
RTS PC

```

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. KTIIX ≡ KTIIC
 - c. IFR

digital

INTEROFFICE MEMORANDUM

Rec'd 5/23/74

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 < 0]
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 arguments 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

0470

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

$$JMPX K(R_n) \equiv LA K(R_n), PC$$

because we have only 16-bit index constants.

PS<08>

3. X-mode bit 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 unsafe programming practice i.e. indexing of the top of the system stack by a process which can be interrupted.

5. YES and no Bruce's remarks 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, i.e. see more than 32K virtual addresses 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 *E 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 \leq 28K no KT
- . MEDIUM \leq 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 Rudge

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

FROM: Tom Hastings

DEPT: -10 Software Engineering

EXT: 6512 LOC: MR 1-2/E37

MAR 21 1975

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 1080 is required. Four or five boards will have to be relayed out for the 20 series. These can be phased into subsequent 1080s, so that we can achieve the goal of the 1080 being a strict subset of the 2010 in function and hardware.

Minutes: The agenda items are indicated in parenthesis.

1. Problem: (10v) 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: (10a) 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?

(10b) 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 0, it remains there.

Solution: Make the section independent extended address of the hardware ACs be 1,,AC instead of 0,,AC. Effectively section 1 must be a code section in the extended machine with the first 20 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,,0 or (VMA 18-31 = 0 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 0,,AC will reference memory in section 0. When the PC is in section 0, MOVXA will follow the KI rule: the LH will always be 0.

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 0 to non-zero sections) and it will mean the same location.

Solution: We decided to leave the double use of 0 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 0 only. They must reference section 0 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: (10b) 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 0-5 in EFIW, or 0-17 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: (10c) 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.

5. Problem: (10d) 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 $+2^{17}$ offset.

6. Problem: (10f) Should BLT backwards be decommitted? 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 one 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: (10h) ADJSP - how can we make it check for overflow 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. (10k) What is effect of spec on 1080 ship date? Only one ECO is needed (to access 6 ACs for EXTEND).

10. (10m) 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=0, bits 18-35 of 2nd word are available to software forever (for counts, pointers, etc).

b. Bits 1-5 of Index Register

When bit 0=0, 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 4096 sections. (Since the VMA is only 23 bits long on the KL, there will probably be a flag saying whether VMA 6-12 are 0 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 0-1 = 11. MBZ(T). Micro-code will give illegal instruction trap.
- f. Bits 0-5 in EXTEND-BLT pointers. Available to software forever (like index register bits 1-5).
- g. Bits 0-5 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 0-5 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 0-5 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. (10i, 10j) EXT-BLT specification (needs a separate spec)
- b. (10g) Rules for writing section-independent subroutines
(T. Hastings will write-up)
- c. (10r) Rules for writing subroutines which will run in
section 0, extended sections, and KA and KI.
- d. (10s) PXCT
- e. (10t) 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

FREDERICK M. TRAPNELL, JR.

Computer Systems Consultant

6321 Hartley 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 40 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

(e.g. airline reservations systems). Thus, machines like the PDP-11 make practical what might be described as the dedicated, on-line business application system.

These applications are both sensitive to and demanding of system performance, which is measured by the speed with which transactions can be processed. They are sensitive to performance because it is directly visible to the system operators in turnaround time. They are demanding of performance because they are always highly sensitive to the cost per terminal. Hence, the user wants to attach as many terminals as possible to a given system. This maximum number is usually limited by system performance.

The PDP-11 has most of the attributes to perform well in this role. Its principle hardware limitation is the size of addressable memory, and particularly that which is addressable by a single user. Curiously, the need for this larger memory arises from the requirement for increased system performance.

Large applications can be implemented on systems with a small memory. This is done by dividing the programs into pieces and overlaying (or paging) them into core as required. Almost without exception, such applications are limited in performance by disk access time. An obvious solution to this limitation is to make all frequently

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

does not arise from the need to increase the number of independent tasks which can time share system resources. It comes from the need for the system to handle in main store application programs which themselves are larger, more complex and highly performance sensitive. Thus, we arrive at a key guideline for development of PDP-11 memory extensions:

- 1) The scheme chosen for extending memory must permit the individual user (task) to directly benefit from the newly available memory with little or no overhead. This means that any one user should be able to directly address the entire available memory. If this addressing cannot always be direct, then it should be nearly so. For example, user directed changing of a page register would be an acceptable but not ideal solution. (Not ideal because it is not as fast as generating the address at execution time.)

Memory Protection, Relocation and Segmentation

We have asserted that the requirement for the user to address all main memory arises from the market for dedicated application computers. This does not, however, obviate the need for memory protection. For, even in these applications the operating system should be protected from the user program to the extent that (1) the latter cannot inadvertently destroy the former and (2) user errors in communicating with the system can be gracefully detected and reported. Furthermore, the concept of multiprogrammed jobs is so deeply ingrained in users today that to prohibit it in a new product would be unwise market strategy. For these reasons, protection of one program from another must be provided.

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

to follow. It is not enough to have a clean technical solution to the problem. It must also be a good solution for a garden variety assembly language programmer. Perhaps more importantly, the compilers must be able to use it in a way which is totally transparent to the programmer! High level language programmers will not grasp complex addressing schemes.

Size of Memory Required

How big a memory should the extended addressing cover? Obviously, there is no simple answer to this; however, I think there are some bounding conditions. First, the addressing range should be at least as big as that of the Interdata 7/32, namely one megabyte or 512K words. On the other hand, it is hard to justify a maximum memory size bigger than 16 megabytes (a la IBM 360). On this basis, I feel if a small change will achieve at least a one megabyte range, then DEC should do it. If, on the other hand, a big change is required to achieve it, then a solution should be found which permits a 16 megabyte memory.

Timing of Introduction

The key factor influencing the time of product introduction of extended addressing is the recent announcement of the Interdata 7/32. From inspection of the manuals for the computers, it appears to compete in performance with the PDP-11/40. Some comparable instructions are faster on the PDP-11; for example an Add from a labled store location to a register is 2.45 microseconds vs 3.25 on the 7/32.

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-11/40 (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 Road for Extended Memory Addressing in the PDP-11"; namely:

- 1) Must be fast to change pages.
- 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 used for buffers, etc. will be coded or allocated in segments. A segment is defined as follows:

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 segldr). This program operates like the linker to resolve all internal globals to make segments 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 access it.

Segments are relocatable; ^{in physical memory} 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 ~~are~~ for each segment must be explicitly declared at assembly time. The access control code which applies to a segment must be declared at segblk 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 ~~and~~ locations; ~~pseudo~~ ~~because~~ ~~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 ~~cap~~ capabilities to access ~~all externs~~ each segment. These capabilities are 3 word quantities (probably) giving the starting physical address of the segment, its length, and the other ~~data~~ data needed for the PDR's which will reference it.

To address an extern 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 extern. 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.

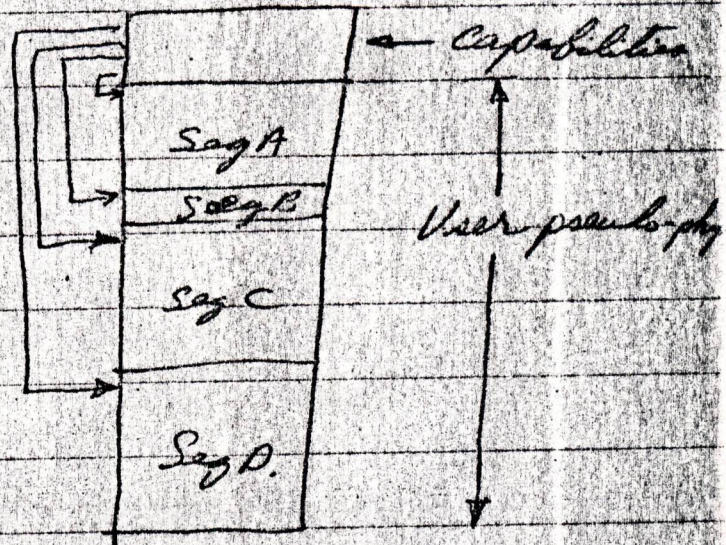
NB. Here's where this one gets klugy!

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. In limited cases 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 ^{the} 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~~ ^{user} has to ~~prove~~ 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 other info his interrupt service routine entry points, etc. It can never be overwritten (the assembler prohibits it).

Some new instructions:

	<u>New Mnemonic</u>	<u>Counterparts</u>
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)
 B) (The assembler won't let page 0 be cleared)

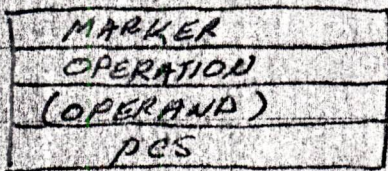
b) JSS can also cause specified page registers to be saved on the stack.

c) RSS will automatically restore APR's saved by a JSS.

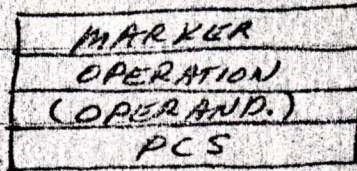
JMS and JSS are ^{two or three} ~~three~~ word instructions; RSS requires two. Their operation and operand words are identical to ~~JMS~~ their respective counterparts. They are preceded by a marker which identifies them.

Formats are:

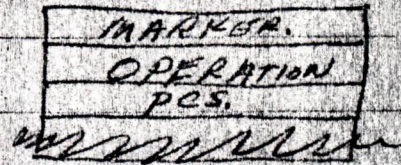
JMS



JSS



RSS



The PCS stands for Page Control Specification. Its 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} ~~prior to executing the~~ instructions. The high byte of the PCS is used to control saving and restoring in JSS and RSS.

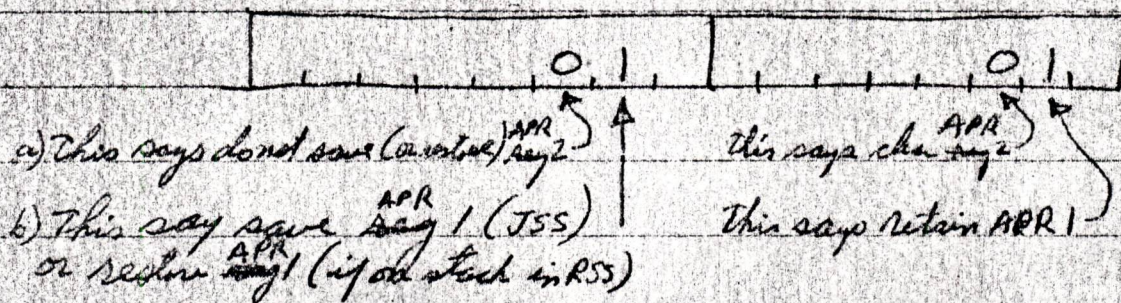
JSS uses its PCS as follows:

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~~ ^{the} JSR is executed.

RSS uses the PCS's as follows:

The APR's which have a flag set in the lowbyte of its PCS are cleared. The On RTS is executed. The ~~saved~~ APR's specified in the highbyte of the PCS on the top of the stack are restored.

PCS Format



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 an a calling segment. Protection segments are well 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 addressing 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 it 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 32 bit), called a short address and a long address respectively.
- 2) If we restrict the user to assembling (empty) modules of no greater than 64K bytes, then the assembler should ~~be able~~ not need to know the type of address. It resolves what it can; the rest are globals.
- 3) The assembler can't know ~~about~~ 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. ~~Let's~~

*) 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.

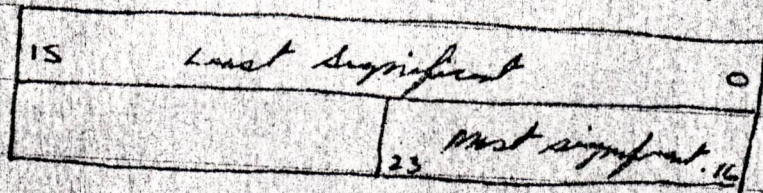
*) Let's see what happens if we let the registers hold both kinds of quantities: 16 bit short addresses and data as well as 24 bit long addresses.

Changes:

0500

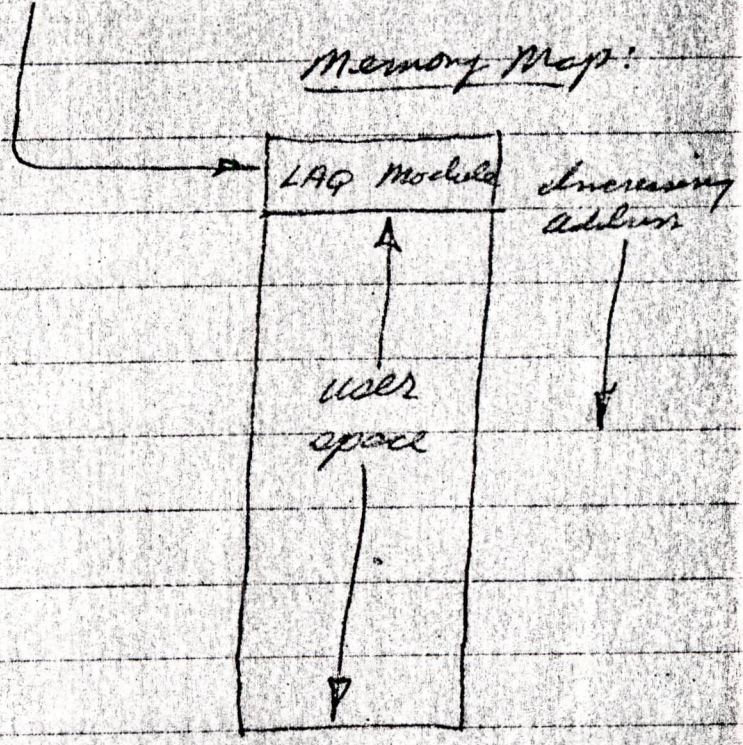
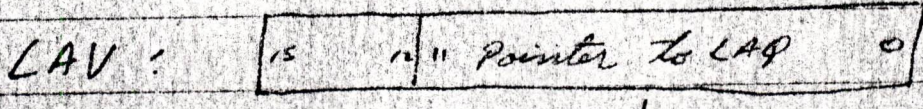
1) Define a new, two word (32 bit) quantity in memory called a Long Address Quantity or LAQ. These are made up of two adjacent words: the lowest address word holds the least significant part of the address, the other word holds the most significant part.

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 ~~globals~~ ^{references} must be ~~flat~~ 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'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 when one is specified

and when a short address has insufficient range.

Here we run into a problem. Short addresses ~~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 3200} ~~never~~ used in a short indirect address. Let it

address or an LAV. - A bit of a provision for this architecture, but on the face of it not too unclear. 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)~~

~~In case (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 add. 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 defn 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 $d = \text{reg}$ 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 so~~ are explicitly designed to ~~work with EA~~ manipulate and move EA's (or the 24 bit ~~portion~~ long form of the ~~so genl.~~ 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 EA's.

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) ~~16 bit~~ L bit to be set temporarily in the BU! (or suspend interrupts til finished).

New Instructions are:

a) Branches which operate on C' and Z'

Branch if LA higher (BAH)	Br if	$C' \vee Z' = 0$
Br if LA lower or same (BAL)		$C' \vee Z' = 1$
Br if LA higher or same (BAHS)		$C' = \phi$
Br if LA lower (BAL)		$C' = 1$
Br if LA equal (to zero) (BAEQ)		$Z' = 0$
Br if LA not equal (to zero) (BANE)		$Z' = 1$

b) Long address manipulators

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.

CSA

Compos 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} put an address or ~~is~~ data. Or if you think the above should always move 24 bit quantities try MOV Rn, (Rd) where Rd points to a table entry (of addresses or data?) Therefore, the proposition is that we only add these seven new mnemonics to the Assembler's repertoire.

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 to reference that global will ~~have~~ use the one EAQ. 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" solution: 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 the trash migrates out through the C or N 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 ka 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

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 mnemonics 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 to 23 of PSECT \$CODE1 and would look something like this:

		<u>Additional inline words</u>
MOV	{@2(R5)} , -(SP)	+1
MOV	{@2(R5)} , -(SP)	+1
MOV	6(R5) , -(SP)	+0
MOV	10(R5) , -(SP)	+2
MOV	# , -(SP)	+0
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 column 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 increases 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:

		<u>Additional I-Stream Word References</u>
L\$FAAL:	MOV [K] , R1	+1
	MPA [] , R1	+1
	ADA16 [I] , R1	+1
	MPA #4 , R1	+0
	LDA [] , R0	+2
	ADA16 [K] , R0	+1
	MPA #4 , R0	+0
	ADA28 [] , R1	+2
	ADA28 [] , R0	+2
	LDF @R1, F0	+0
	MULF @R0, F0	+0
	ADDF [S] , F0	+1
	STF F0, [S]	+1
	INC [K]	+1
	CMP [K] , [@2(R5)]	+3
	BLE L\$FAAL	+0

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 goes from 45 to 61 - a 36% increase.

/br

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
```

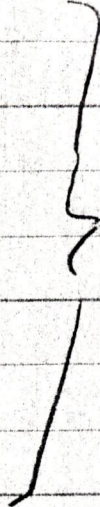
Fig. 1.

UNRELEASED SYSTEM, SUPPORTED FOR FIELD TEST PURPOSES ONLY

Static Increase

```

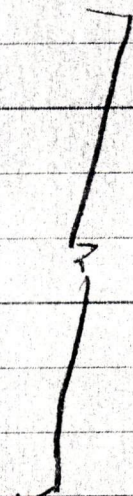
000000 .TITLE MATMUL
000000 .IDENT 29AUG
000000 .PSECT $IDATA
000000 .WORD 1,0,1,0
000010 .WORD 0
000012 .WORD 120000
000014 .WORD 0,0
000020 .WORD 31004,0,0
000026 .WORD 1,0,1,0
000036 .WORD 0
000040 .WORD 120000
000042 .WORD 0,0
000046 .WORD 31004,0,0
000054 .WORD 1,0,1,0
000064 .WORD 0
000066 .WORD 120000
000070 .WORD 0,0
000074 .WORD 31004,0,0
000080 .PSECT $CODE1
    
```



3

```

000000 MATMUL:
000000 MOV @2(R5),-(SP)
000004 MOV @2(R5),-(SP)
000010 MOV 4(R5),-(SP)
000014 MOV #SIDATA+14,-(SP)
000020 JSR PC,MKA2$
000024 MOV @2(R5),-(SP)
000030 MOV @2(R5),-(SP)
000034 MOV 6(R5),-(SP)
000040 MOV #SIDATA+42,-(SP)
000044 JSR PC,MKA2$
000050 MOV @2(R5),-(SP)
000054 MOV @2(R5),-(SP)
000060 MOV 10(R5),-(SP)
000064 MOV #SIDATA+70,-(SP)
000070 JSR PC,MKA2$
000074 MOV #1,I
    
```



21

```

000102 LSFAGG:
000102 MOV #1,J
000110 LSFABI:
000110 SETF S
000112 CLR F
000116 MOV #1,K
000124 MOV J,R1
000130 MUL $IDATA+50,R1
000134 MOV R1,STEMPS
000140 LSFAAL:
000140 MOV K,R1
000144 MUL $IDATA+22,R1
000150 ADD I,R1
000154 ASL R1
000156 ASL R1
000160 MOV STEMPS,R0
000164 ADD K,R0
    
```

T-Stream

3
3
3
3
1
1
3
3

1
1
1
1
1
1
1
1

Fig. 2

55

UNRELEASED SYSTEM, SUPPORTED FOR FIELD TEST PURPOSES ONLY

000170	ASL	R0	1	
000172	ASL	R0	1	
000174	ADD	SIDATA+16,R1	3	1
000200	ADD	SIDATA+44,R0	3	1
000204	LDF	@R1,F0	2	
000206	MULF	@R0,F0	2	
000210	ADDF	S,F0	3	1
000214	STF	F0,S	3	1
000220	INC	K	3	1
000224	CMP	K,@2(R5)	6	3
000232	BLE	LSPAAL	1	
000234	MOV	J,R1		1
000240	MUL	SIDATA+76,R1	45	1
000244	ADD	I,R1		1
000250	ASL	R1		
000252	ASL	R1		
000254	ADD	SIDATA+72,R1		1
000260	MOV	S,(R1)+		1
000264	MOV	S+2,@R1		1
000270	INC	J		1
000274	CMP	J,@2(R5)		3
000302	BLE	LSPAFI		
000304	INC	I		1
000310	CMP	I,@2(R5)		3
000316	BLE	LSPAGG		
000320	RTS	PC		
	.GLOBL	SOTSV		
	.END			

SECTION	SIZE			
SCODE1	105	+ 55	=	160
SIDATA	33	+ 3	=	36
SVARS	5	+ 0	=	5
STEMPS	1	+ 1	=	2
				<u>203</u>

,LP:/LI:3=VECMUL./-T R

203

Fig. 3

Gordon Bell
12-1