


CHAPTER 1 SEVEN VIEWS OF COMPUTER SYSTEMS

A computer is determined by many factors: its architecture¹ and other structural properties, the technological environment and human aspects of the environment in which it was designed and built. Many of the non-architectural design factors lie outside the control of the designer: the availability and price of the mostly electronic technology, the various government and industry rules and standards, the current and future market conditions, and the cumulative investment in software of the various users. The computer is the product of this total environment.

In this book we reflect on all the DEC computers: their goals, their architectures, their various implementations² (and realizations²) and occasionally on the people who designed them. We limit this presentation to the engineering of the basic computer hardware and do not describe, except as interfacial design, the engineering of peripheral equipment and of software.

We attempt to show the interrelationship of the design beginning with the architectural specifications, and observe how it was affected by technology, by the engineering organization, the sales, application, and manufacturing aspects, and finally how we perceive it is actually used. Figure 1 shows these various factors: the organizational entities and implied design activities affecting the resultant computer. The lines indicated the primary two-way flow of information for product specification. The physical flow of materials is more direct and easier to trace as it comes from basic technology suppliers either inside or outside the DEC organization, moving through the plants that build basic computer parts (memories, processors, cabinets, peripherals) and

¹ Although computer architecture has been used loosely to refer to computer hardware, we will adhere to the original definition [Amdahl, Blaauw and Brooks, 1964] to avoid further confusion. This definition is: "The term architecture is used here to describe the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behavior, as distinct from the organization of the data flow and controls, the logical design and the physical implementation."

² Using the definitions of Blaauw, 1977. 

finally to where the final system is assembled. Finally it is shipped via service personnel to the end user's site.

Computer engineering is the complete set of activities ^{the use of} (e.g., taxonomies, theories, models, heuristics) associated with the design and construction of computers. ^{It is} Like other engineering, ^{and} we especially ^{concur with} ~~adhere to~~ the definition that Hamming once gave: engineers first turn to science for answers and help, then to mathematics for models and intuition and finally to the seat of their pants.

In the few decades since computers were first conceived and built, computer engineering has come from a set of design activities that were mostly seat-of-the-pants based, ^{to a point where some parts} ~~through parts that~~ are quite well understood and based on good models and rules of thumb, ^{e.g., technology models, and some parts} ~~and finally there are parts that~~ are completely understood with useful theories, ^{e.g., circuit minimization.} ~~We believe there are a number of the set of design activities and disciplines--this chapter will concentrate on presenting them.~~

Plan for the Chapter

Although the reader must understand the design activities ~~associated~~ implied by each of the organizational entities in Fig. 1.6 in order to understand computer engineering, such a presentation would be unduly philosophical, boring, and incomplete. We are restricting this book to ~~DEC computers~~ and to design activities that are not well described in papers and theories. We also concentrate on the object of the design (the concrete) rather than segmented ^{technical} ~~activities~~ ^{aspects} covered in conventional textbooks about computer architecture, and

In this chapter we present seven views that we, as designers, have found to be useful in thinking about computers and the process which molds their form and function.

~~computer and digital system design.~~

~~The chapter is organized into six parts. The first part is a set of eight views. Each view is complete, and forms the way we can view the computer.~~

~~Four views~~ are roughly on how, as a complex system, the computer can be decomposed and characterized for designers. ^{One} ~~Two~~ views ^{is} ~~are~~ on how it is characterized as a complete system within a marketplace and ^{one} ~~two~~ views ^{is} ~~are~~ ^{on} ~~about~~ the general problem of design.

Since performance is such a major component of the objective function by which a computer is judged, we look at ~~a number of~~ the common ways for evaluating ^{ing it. This is} performance ~~in section two,~~ ^{given in the appendix.}

Although technology is clearly the dominant push for computer evolution packaging, component interconnection and power are complementary to and must support its use. Indeed, we believe how components are interconnected is equally important to the component. These complementary views are given in two sections.

Section five permits the manufacturing process, because without the ability to produce machines, the evolution wouldn't take place.

The final section gives the reader several ways to approach the study of the book.

^{copy}
View 1: Structural Levels of a Computer System

In Bell and Newell, (1971) a set of conceptual levels for describing, understanding, analyzing, designing, and using computer systems was posited. The model has survived major changes in technology, e.g., the fabrication of a complete computer on a single silicon chip, and changes in architecture, e.g., the addition of vector and array data types. The levels are shown in Figure 2, a refined version of the original. For a description of this view of a computer system we rely entirely on the following [Bell and Newell, 1971: pp 3-10].

There are at least four levels of system description, possibly five, that can be used for a computer. These are not alternative descriptions in the sense that anything said one way can be said another. On the contrary, each level arises from abstraction of the levels below it. Each does a job that the lower levels could not perform because of the unnecessary detail they would be forced to carry around.

A system (at any level) is characterized by a set of components, of which certain properties are posited, and a set of ways of combining components to product systems. When formalized appropriately, the behavior of the systems is determined by the behavior of its components and the specific modes of combination used.

Elementary circuit theory is an almost prototypic example. The components are R's, L's, C's, and voltage sources. The mode of combination is to run wires between the terminals of components, which corresponds to an identification of current and voltage at these terminals. The algebraic and differential equations of circuit theory provide the means whereby the behavior of a circuit can be computed from the properties of its components and the way the circuit is constructed.

There is a recursive feature to most system descriptions. A system, composed of components structured in a given way, may be considered a component in the construction of yet other systems. There are, of course, some primitive components whose properties are not explicable as the resultant of a system of the same type. For example, a resistor is not to be explained by a subcircuit but is taken as a primitive. Sometimes there are no absolute

primitives, it being a matter of convention what basis is taken. For example, one can build logical design systems from many different primitive sets of logical operations (AND and NOT, NAND, OR and NOT, etc.).

A system level, as we have used the term in [Fig. 2], is characterized by a distinct language for representing the system (that is, the components, modes of combination, and laws of behavior). These distinct languages reflect special properties of the types of components and of the way they combine. Otherwise, there would be no point in adopting a special representation. Nevertheless, these levels exist in the system analyst's way of describing the same physically existing system. The fact that the languages are highly distinct makes it possible to be confident about the existence of an additional intermediate level, it is because new representations have not yet congealed into distinct formal languages. As we noted, within each level there exists a whole hierarchy of systems and subsystems. However, as long as these are all described in the same language, e.g., a subroutine hierarchy, all given in machine-assembly language, they do not constitute separate system levels.

With this general view, let us work through the levels of computer systems, starting at the bottom. Each level in [Fig. 2] actually has two languages or representations associated with it: an algebraic one and a graphical one. These are isomorphic to each other, the same entities, properties, and relations being given in both.

The lowest level in [Fig. 2] is the circuit level. Here the components are R's, L's, C's, voltage sources, and nonlinear devices. The behavior of the system is measured in terms of voltage, current, and magnetic flux. These are continuously varying quantities associated with various components, and so there is continuous behavior through time. The components have a discrete number of terminals, whereby they can be connected to other components.

The circuit level is not in fact the lowest level that might be used in describing a computer system. The devices themselves require a different language, either that of electromagnetic theory or of quantum mechanics (for the solid-state devices). It is usually an exercise in a course on Maxwell's equations to show that circuit theory can be derived as a specialization under appropriately restricted boundary conditions. Actually, even at its level of abstraction, circuit theory is not quite adequate to describe computer technology since there are a number of mechanical devices which must be represented. Magnetic tapes and drums are most likely to come to mind first, but card readers, card punches, and Teletype terminals are other examples. These devices obey laws of motion and are analyzed in units of mass,

length, and time.

The next level is the logic level. It is unique to digital technology, whereas the circuit level (and below) is what digital technology shares with the rest of electrical engineering. The behavior of a system is now described by discrete variables which take on only two values, called 0 and 1 (or + and -, true and false, high and low). The components perform logical functions: AND, OR, NOT, NAND, etc. Systems are constructed in the same way as at the circuit level, by connecting the terminals of components, which thereby identify their behavioral values. The laws of boolean algebra are used to compute the behavior of a system from the behavior and properties of its components.

The previous paragraph described combinatorial circuits whose outputs are directly related to the inputs at any instant of time. If the circuit has the ability to hold values over time (store information), we get sequential circuits. The problem that the combinatorial-level analysis solves is the production of a set of outputs at time t as a function of a number of inputs at the same time t . As described in textbooks, the analysis abstracts from any transport delays between input and output; however, in engineering practice the analysis of delays is usually considered to be still part of the combinatorial level.

.....

The representation of a sequential switching circuit is basically the same as that of a combinatorial switching circuit, although one needs to add memory components, such as a delay element (which produces as output at time $t - \Delta$). Thus the equations that specify structure must be difference equations involving time. Again, there is a distinction (even in representation) between synchronous circuits and asynchronous circuits, namely, whether behavior can be represented by a sequence of values at integral time points ($t = 1, 2, 3, \dots$) or must deal in continuous time. But this is a minor variation.

.....

A glance at [Fig. 2] shows that we have described only the lower part of the logic level. There is another part, called the register-transfer level (or RT level).... The components of an RT system are registers and functional transfers between registers. A register is a device that holds a set of bits¹. The behavior of the system is given by the time course of values of these registers, i.e., their bit sets.

The system undergoes discrete operations, whereby the values of various registers are combined according to some rule and then are stored in another register (thus "transferred"). The law of

¹This assumes that the elementary state variable of the system holds a bit (i.e., one of the two values, such as 0 or 1). This need not be; sometimes the elementary variable holds a decimal digit (one of 10 values) or a character (one of, say, 48 values). For present purposes we can talk in terms of bits, without losing anything thereby.

combination may be almost anything, from the simple unmodified transfer ($A \leftarrow B$) to logical combination ($A \leftarrow B \wedge C$) to arithmetic ($A \leftarrow B + C$). Thus a specification of the behavior, equivalent to the boolean equations of sequential circuits or the differential equations of the circuit level, is a set of expressions (often called productions) which give the conditions under which such transfers will be made.

.....

We now move to the fourth and last level. In [Fig. 2] it is called the Processor-Memory-Switch level, or PMS level for short. The name is not recognized, nor is any other, since the level exists only informally. Nevertheless, its existence is hardly in doubt. It is the view one takes of a computer system when one considers only its most aggregate behavior. It then consists of central processors, core memories, tapes, disks, input/output processors, communications lines, printers, tape controllers, busses, Teletypes, scopes, etc. The system is viewed as processing a medium, information, which can be measured in bits (or digits, characters, words, etc.). Thus the components have capacities and flow rates as their operating characteristics. All details of the program are suppressed, although many gross distinctions of encoding and information type remain, depending on the analysis. Thus one may distinguish program from data, or file space from resident monitor. One may remain concerned with the fact that input data are in alphameric and must be converted into binary, or are bit-serial and must be converted to bit-parallel.

We might characterize this level as the "chemical engineering view of a digital computer," which likens it more to a continuous-process petroleum-distilling plant than to a place where complex FORTRAN programs are applied to matrices of data. Indeed, this system level is more nearly an abstraction from the logic level than from the program level, since it returns to a simultaneously operating flow system.

One might question whether there is a distinct systems level here. In the early days of computers almost all computer systems could be represented as in the diagram in M.I.T.'s Whirlwind computer programming manual in [Fig. 00]: with classic boxes of memory (storage), control, arithmetic, and input/output. Actually, this view of the computer in 1953 was considerably advanced; few texts on the logic design of computers in the 1960s have such a detailed model. This model has secondary memory (magnetic tape and drums in the Whirlwind's case). The most interesting aspect of the model, which text writers omit, is any kind of switching (the bus of [Fig. 00]). The bus provides a communication path to link the other components. Certainly the pushbuttons (actually the console) is novel for such a model.

We have dropped the program level from the original set of levels. It was placed between the RT level and the PMS level; experience showed that the inclusion of this level was awkward and inconsistent. The primary reasons ~~are~~ *is that* ~~as follows.~~ The program level is a functional attribute, whereas all the other levels are structural. If it is to be included, a better location is on top of the PMS level -- for uniprocessors it can be in either location; however, for computer systems constructed from computer modules [Chapter 00] it clearly must be located above the PMS level. *Another* ~~The final~~ reason for its awkwardness is that for microprogrammed structures, there is clearly another program level at the RT level. Moreover, the ISP notation, which was introduced to describe the original program level, has been used to describe behavior at the other levels. For example, at the PMS level the address translation mechanism for inter-computer-module sharing [Fuller, et al, 1973b] and at the RT level within DEC for simulation and microprogramming.

The state system representation, which appeared as an auxiliary representation to the side of the logic level, has also been dropped. A system is represented as capable of being in one of N abstract states at any instant of time. Its behavior is specified by a transition function that takes as arguments the current state and the current input and determines the next state and the output. This representation has been dropped because it is functional, not structural. Moreover, behavior at every level, not just the logic level, can be represented by state transition functions.

Since the PMS level (processors, memories, and transducers), interconnected by switches (often in a bus structure), was tentatively proposed as a level in

1971, it has become recognized as a formal level. Design at that level has become important as we strive to build models of system-level behavior, and will become more important as microcomputers become our standard structural building block and as the cost of interconnect dominates system cost.

Many notations are used at each of the four structural levels. The commonly used ones are given in table Notations. A complete description of the PMS and ISP notations is given in [Bell and Newell, 1971: Chapter 2]. Those aspects of PMS which are used in this book are described in Appendix 00. The ISP notation has evolved to ISPS and is described in Appendix 00.

View 4: Computer Classes: A Marketplace View

Because it is the complete marketplace process (Fig. 1) that produces the computer, this view is the most complex. A computer is characterized (in terms of marketability) as a function of price, performance, and time of introduction in what might appear to be a commodity-like environment. Later, the notion of function, or how the computer is applied must come in.

Because various computers operate at different performance rates, and at various costs, price/performance ratios affect marketability since computation can be supplied in so many ways. For example: computation can be supplied by a shared large, central efficient batch computer; each organizational entity can own and operate a shared minicomputer; an individual can operate a single desk top system; or each individual may operate a programmable calculator.

When users write programs, there is a lifetime associated with use and a possible need to have compatible products at a given time and for all time. Although the supply of basic, rapidly evolving technology permits new designs to be more cost effective, and even radical, continuity with the past must exist.

This view begins with a description of how technology provides basic improvements with each new generation (every 6 or so years). An example of alternative new designs is given to show why most new designs are evolutionary versions of the past, and usually provide increased performance at constant price. With this background, the four classes, differentiated by price and performance, that have evolved over the last four generations are presented. This simple model is then elaborated and critiqued.

Design(er) Styles Arising From Technology Advances

The influence of technology on the computers that we build and take to the marketplace is so strong that the four generations of computers have been named after the technology of their components: vacuum-tubes, transistors, integrated-circuits (multiple transistors packaged together), and LSI circuits (Large-Scale Integration). Every electronic technology has its own set of characteristics (e.g., cost, speed, heat dissipation, packing density, reliability), all of which the designer must balance. These factors combine to limit the applicability of any one technology; typically, one technology is used until a limit is reached, or another technology supersedes the old.

When an improved basic technology becomes available to a computer designer, there are four paths (~~see Fig. x) and y)~~ ^{the} designs can take to incorporate the technology:

1. use the newer technology to build a cheaper system with the same performance;
2. hold the price constant and use the technological improvement to get an increase in performance;
3. push the design to the limits of the new technology, thereby increasing both performance and price; or

*double spaced please
right through this view
(except where single spacing requested)*

4. find a drastically new structure using the computer as a basic archetype (e.g., calculators) such that the design can be considered off the evolutionary path.

Figure x shows the trajectory of the first three of the design alternatives. ~~By~~ going too far in price or performance, (i.e., building beyond the technology) is dangerous and can lead to a zero performance, high cost product. These design alternatives occur in an evolutionary fashion as in Fig. y where there is a first (base) design, and subsequent designs evolve from the base.

In the first ~~case~~ ^{design style}, the performance is held constant and the improved technology is used to build lower-cost machines -- new applications are attracted. The minicomputer (for minimal computer) has traditionally been the vehicle for entering new applications, since it is the smallest computer that can be constructed with a given technology. Each year, as the price of the minimal computer price declines, new applications become economically feasible.

The second, constant cost alternative using the improved technology to get better performance, will usually yield the best increase in total system-cost-effectiveness. This approach provides a growth in performance and quality at a constant price and is probably the best for the majority of existing users.

If the new technology is used to build the most powerful machine possible, then the designs often advance the state of the art. New designs should solve previously unsolved problems. There are usually two motivations for operating at this leading edge: preliminary research motivated by the knowledge that the technology will catch up; and national defense, where an essentially infinite amount of money is available because the benefit--avoiding annihilation--is infinite.

Using new technology for constant performance and constant cost designs

The consequence of new generation technology is best seen by observing how the technology can be applied to provide the two fundamentally different designs based either on: constant performance and decreased price; or, constant price and increased performance. The following table shows the effect of pursuing the two design strategies.

Table: Using New Technology for Constant Cost and Constant Performance Designs

Introduction time (generation)	t	t+1	t+1	t+1
Design style	base case	constant cost, increased performance	constant performance, decreased price	constant performance, decreased price
Application	base	base	base	new base
Computer price	1	1	0.5	0.5
Operating costs (range)	2-4	2-4	2-4	1-2
Total cost	3-5	3-5	2.5-4.5	1.5-3
Performance (and improvement)	1	2	1	1
Improvement (in total cost)	1	1	.83-.9	.5-.6
Performance/price (computer only and improvement)	1	2	2	2
Performance/Total-cost	.33-.2	.66-.4	.4-.22	.66-.4
Improvement in (Perf./total cost)	1	2	1.21-1.1	2

single spacing - as it is thanks

The first column gives the base case at a given generation/time, t. The price, performance and performance/price ratio of the computer are all 1. As the computer is applied to a particular environment, operational overhead is added at a cost of 2 to 4 times the original price of the computer; the total cost to operate the computer becomes 3 to 5, and the performance/total-cost ratio is reduced to between .33 and .2 (depending on the total cost).

Now assume the same operating environment, with the same fixed (overhead) costs to operate, at a new generation time, t+1 when technology has "improved" by a factor 2. Two alternative designs are carried out, one is at constant price and at constant performance (cols. 3 and 4). The application is constant in 3 cases (cols. 1-3) and a new base application is discovered (col. 4). Both the constant cost and constant performance designs give the same basic performance/cost improvement--when only the cost of the computer is considered. However, when one considers the high, fixed overhead costs

associated with a base application (cols. 1-3), there is a relatively small improvement in cost performance/cost, although there is a cost savings of 17 to 10 per cent with the minimal design. The greatest gains come in applying the computer with greater performance and getting the attendant factor of 2 gain in performance and in performance/price ratio.

To summarize, the constant-price design style gives a better gain because operating costs remain the same. Its gain can only be equalled by the constant-performance design style when operating costs are halved upon its application. This only occurs when a new application is tackled.

USING PRICE AND PERFORMANCE TO DEFINE COMPUTER CLASSES

We have shown by example how a new generation of technology allowed a complete design trade-off between price and performance, giving two equally valid approaches to new products. Figure 5 plots price and performance planes for three generation times, t , $t+1$, and $t+2$.

Computer classes are distinguished by price and are named as follows: sub-micro (to come in the next generation--say by 1980), micro, mini, midi, maxi, and super. The ~~names~~ ^{classes} midi- and maxi- are usually called by the single, non-descriptive name, mainframe, but we will keep the two separated. At each new technology time, ~~the~~ performance increases by one category (e.g., midi performance is available on a mini); and a new class of computer; the mini(mal) computer is introduced at a significantly lower price level.

If we apply the three design styles shown in Fig. x over several generations we obtain the plot given in Fig. 6.

We have developed the following thesis:

- continual application of technology via the two major design styles results in
- (a) price declines which create new classes of computers,
 - (b) new classes becoming established classes, and
 - (c) established classes becoming encroached upon.

The measure used to define a new class is price, whereas the measure defining an established class is performance.

We now present several comments on the thesis, aimed at refining it.

1. A fresh viewpoint is needed to create a new class
The continuity implied by the thesis is deceptive. It suggests that new classes come about by the continual application of the constant-performance, decreasing cost style of design. Viewing the industry as a whole this is true. However, a new class is usually not created by the same organization. It seems that a fresh viewpoint is needed to create a new class. A new company, or organization within a company, can do this.

1 A subsequent discussion on technology will examine whether the tradeoff can be made to such a high degree, but we will proceed assuming the trade-off is possible. It is also necessary to assume that performance can be specified easily as a single metric...as opposed to the n-dimensional space.

The minicomputer class came about by DEC founders taking the view that simple machines, unencumbered by the costly general purpose facilities of the mainframes (many data types, sharing mechanisms, etc.), could be built and would be viable.

While the mini(mal) design philosophy pervaded for nearly 10 years as ~~the~~ minicomputer, the semiconductor-based, processor-on-a-chip ~~(called a microprocessor)~~ computer (called microcomputers) name identifies a significantly lower priced computer. This lower cost is fundamentally based on the elimination of a level of integration (interconnection), ~~and hence is packaging based.~~ The microcomputer dispels the lethargy growing around the now constant-price minicomputer of the mid-1970's. In this way, more radical thinking permits different customers (and usage) to be identified and the price to decrease more rapidly. The sub-microcomputer is predicted to be a complete computer (with both processor and program memory) on a single chip, although it is hardly as radical as the microprocessor which moved users to apply computers based on interconnecting chips (on printed circuit boards) and interconnecting a small number of boards (as opposed to designing boards to be placed in minicomputer boxed computer configurations).

In both the mini and micro cases a fresh organization broke out. Why is a fresh viewpoint needed? Because the process produces constant-price computers. Most human organizations act to preserve the status quo. Let us examine each part of the organization in turn.

- a. The existing user, who is the potential customer for a new system has a set of fixed costs (overhead). Productivity improvements are required to stay at equilibrium with his organization and to sustain or increase funding. Operating costs (personnel, paper, power, etc.) are likely increasing due to inflation, and although a lesser priced machine could hold performance/cost constant, there would be no productivity gain. Requiring less or equal funds, would ultimately cause the loss of organizational power since this is based on people, expenses, ability to provide increased service, etc. The pressure is to get larger, more powerful machines. His software investment requires that the new system be compatible with his earlier system.
- b. The sales force respond mainly to orders which are lost because of poor performance, because its measure is so tangible. Price (and even lack of functionality) is considered only when the performance of two alternatives is equal. The amount of work a salesperson has to do is measured in sales yields and, indirectly, the number of systems sold. The reduction of the sales price of a given system by a factor of two, requires selling twice the number of systems. This implies working significantly harder (neglecting the effects of price elasticity). For example, within DEC, because of the wide price range of its products, a salesperson need sell only one large system per year, while at the low end he must sell 1500.
- c. Marketing reinforces the inputs from the sales force, while

~~A subsequent discussion on technology will examine whether the tradeoff can be made to such a high degree, but we will proceed assuming the trade-off is possible. It is also necessary to assume that performance can be specified easily as a single metric...as opposed to the n-dimensional space.~~

interjecting its own bias and noise; moreover, because of the broad range of applications, it presents conflicting product definitions. There is little information about sales that were lost due to substantially lower price or new potential products because an existing organization does not lose sales to products it doesn't have. Thus, a new product must have: the performance of the current most competitive machines; the price of the most marginal competitor (who may be losing money to enter the market); the software of all the competitors (since each differentiates itself by unique software); and all the service of the largest (and possibly highest price competitor).

- d. Engineering. The new product specification input is to improve performance as seen above. Technology pushes engineers to use higher-performance components since technology suppliers behave in an identical fashion to the computer supplier. Although there are various styles of designs (and designers), the most common tendency of the designer is to use new technology and to provide higher performance, and solve the problems inherent in subsequent designs rather than working to reduce costs again. The PDP-8 family provides an example.
- e. Manufacturing constraints and planning are similar to sales. A plant is measured in units, quality level, and unit cost. Preserving a given cash flow, when faced with radically different cost products, usually requires substantially different planning, materials handling, manufacturing, testing, etc. Since the basic time focus (weekly production schedules) is short term within manufacturing, the radical changes for a higher volume business (i.e., long term planning) is difficult. Product design change ideas back to engineering often address solving previous problems, that add to product cost.

2. The maximum evolution rate is not attained

To fully exploit each new technology improvement, a matching advance in packaging is needed. For example, higher circuit densities demand new cooling methods; lower-cost module carriers are needed to match microprocessor-based control techniques. However, engineering projects tend to minimize the number of risk dimensions. As a result, a new circuit technology is used twice: once with existing packaging and a second time with packaging that matches the technology. There are examples in both the PDP-11 and S/370 families.

3. Conflicts occur within the established classes

An established computer class, since it is defined on the basis of performance, is entered by constant cost successors from the class below it. Moreover, suppliers within a class are, by their dominant constant price evolution, operating to move out of the class.

Figure ⁸10 shows how inertia in the various producer-consumer pairings (of Fig. 1) can cause a market to move away from a set of suppliers and users. Here, we assume only two machine types ~~1 and 2 (each covering a range of~~ ^{classes.}

¹This is probably a conservative view. Since semiconductors are so commodity oriented, performance is the only way to maintain market differentiation and price.

~~prices~~). With new technology, there are no subsequent truly higher performance implementations, only lesser priced ones. In reality PC pair I goes on to build and apply higher performance machine types 3 and 4 at times $t+1$ and, $t+2$, with the attendant danger of not being able to build or utilize the attendant higher performance. Similarly, consumer set II, who may have secondary users of PC pair I, now can become fully independent at time $t+1$ with their own machines. This phenomenon of migration is clearly visible as users of large, central computation centers, (e.g., a small part of an organization) gets its own minicomputer at time, $t+1$. Then at time $t+2$ the individuals of the small organization get their own individual computers.

4. The classes have different functionality

The smoothness of the model is not meant to imply that each class implements the same instruction set and PMS-level configurations, and that they differ only in speed. Much specialization occurs in each class and many attributes appear to lesser degrees in the lower performance classes. For example, there are more data types in the ~~lower~~^{larger} machines, their address spaces (both physical and virtual) are ~~lower~~^{larger}, and the software support is generally broader. Resources devoted to increasing reliability and availability are more common in the higher priced machines. The PDP-11 family, from the VAX-11/780 down to the LSI-11, exemplify these functionality differences.

5. Different technologies contribute

The model suggests that there is one monolithic technology, steadfastly progressing, which provides incremental improvements. In fact, there are several technologies; mass storage, primary memory, I/O units, logic, packaging, and power are the principal ones. They improve at different rates: for example, memory prices have consistently halved every two years, whereas electro-mechanical technology is slowly increasing in price.

6. A definition of the minicomputer

Our model provides a named class, the mini, to describe those machines in the price range between micros and ~~mainframes~~^{midis}. However, the term mini has also been used to denote minimal computer [Bell, 1971]:

"Minicomputers (for minimal computers) are a state of mind (or the designer's minds); the current logic technology, and the characteristics found in larger computers, are combined into a package which has the smallest cost."

~~This corresponds to an earlier definition (Bell, 1971).~~ By this definition, the processor-on-a-chip (i.e., the microprocessor) microcomputers are minicomputers. ~~Thus,~~ the primary goal is cost; hardware/software tradeoffs are made to transfer any hardware costly operations to the software. A price-based definition can be exacerbated by people who collect and base analysis on time varying data; so one is comparing different machines at different times in their lifetimes.

Computing Europe, as late as November 6, 1975 was frustrated and concerned

¹This is probably a conservative view. Since semiconductors are so commodity oriented, performance is the only way to maintain market differentiation and price.

about a definition: "When is a minicomputer not a minicomputer? Surely it is time the industry faced the problem of evolving a standard definition."

Many definitions use price to segment and the consensus seems to be that a minimal configuration minicomputer should cost less than \$50,000. Such a price definition is reasonable, although one has to be careful of a price discriminator as the earliest low priced computers were short lived -- their high cost to manufacture caused the ultimate demise of their *suppliers*. ~~manufacturers.~~

Iann Baron (Computing Europe, December 1975) made a simple, useful definition: "Minicomputers are a state of mind. It is not useful to define them in terms of price or capability because of the rapid changes in technology."

Perhaps some of the difficulty is due to a lack of understanding of evolving technology. We will therefore pursue a definition in the hope that we provide some insight into technology use.

The wide variation in configurations and levels-of-integration causes the greatest complications in comparison and definition. That is, each instance (application) of a specific machine model varies greatly. The notion of balancing a computer to handle any task in a general sense just does not usually apply to minicomputer applications. This is like designing a factory to make automobiles, with only one machine type to handle the foundry, metal stamping, conveying, welding, assembly, etc. for steel, aluminum, plastic and glass parts! Minis are generally applied to specific tasks, using specific configurations, and operated with specific software rather than being configured to handle any situation. The configurations vary so that, in the case of a large data base problem, one could have a \$5,000 or \$10,000 processor and several hundred thousand dollars of disk memories. Users more often succumb to their other instincts and get the biggest machine they can for a problem (to be safe) even though it does lower the system reliability and raise the price.

A comment, Computing Europe, (December 1975) summed up the situation: "The difference is similar to tungsten or fluorescent lamps...lower efficiency...or higher overheads. Fluorescents are undoubtedly more efficient for business lighting, but have a minimal share of the Christmas tree lights market".

In Part II of the book we
~~We can further~~ try to characterize minicomputers by looking at how they have changed over four generations. ~~Part II on the early minicomputer.~~ *The part* shows the price, package, size, power, speed and logic technology evolution with time.

In July 1977 we supplied a definition to The Director of Computer Resources Development, USAF.

MINICOMPUTER: A computer originating in the early 1960's and

¹ ~~recall~~ ^{Bell} being asked by a telephone company engineer whether he should install a dozen 100 megabyte discs (each cost about \$25,000) on a large PDP-11/70 (approximately \$125,000) for a data base application ^{to do a job} that he used to do with a 370/168 (approximately \$3,500,000). ~~My~~ ^{sell} only doubt was not that it should be an 11/70, but whether he could use a lower model number like an 11/40 (approximately \$60,000), and thus get the attendant less parts and higher reliability.

predicated on being the lowest (minimum) priced computer built with current technology. From this origin, at prices ranging from 50 to 100 thousand dollars, the computer has evolved both at a price reduction rate of 20% per year and has also evolved to have increased functionality and a slightly higher price with increasing functionality and performance.

leave single spaced

Minicomputers are integrated into systems requiring direct human and process interaction on a dedicated basis (versus being configured with a structure to solve a wide set of problems on a highly general basis).

Minicomputers are produced and distributed in a variety of ways and levels-of-integration from: printed circuit boards containing the electronics; to boxes which hold the processor, primary memory, and interfaces to other equipment; to complete systems with peripherals oriented to solving a particular application(s) problem. The price range(s) for the above levels-of-integration, in 1978, are roughly: 500 to 2,000; 2,000 to 50,000; and 5,000 to 250,000.

7. Economies of Scale

A topic covering different sizes of machines would be incomplete without a discussion of economies of scale.

For nearly all man-made objects (e.g., transportation vehicles, electricity generation, buildings) there is usually some economy of scale because there are high fixed costs that do not increase as rapidly as the output of an object increases.

For computers, factors leading to economies of scale often apply over several dimensions. The same software can be used on many models. Sales and field service people can attend to a wide range of equipment. Manufacturing facilities can be adapted to produce different models.

Grosch (1953) suggested that there was an economy of scale for computers according to the performance/price relationship:

2

$$\text{Performance} = \text{constant} \times \text{Price}$$

Several studies [Bell and Newell, 1971; Knight, 1966; Phister, 1976; Sharpe, 1969; Turn, 1974] have examined whether this is true for a given set of machines; conversely it is possible to use the relationship in pricing. It is desirable that the performance increase more rapidly than price for improved operating economy.

Continue with corrections from marked up copy

We might also conjecture that there is only a linear relationship between price and performance and the square law relationship is only apparent (see Fig. Groschlin) because the curve roughly "fits" for a factor of 4 increase in price. For two alternative ways of performing the same function at different price scales, the higher priced object provides proportionately

basis).

Minicomputers are produced and distributed in a variety of ways and levels-of-integration from: printed circuit boards containing the electronics; to boxes which hold the processor, primary memory, and interfaces to other equipment; to complete systems with peripherals oriented to solving a particular application(s) problem. The price range(s) for the above levels-of-integration, in 1977, are roughly: 500 to 2,000; 2,000 to 50,000; and 5,000 to 250,000.

7. ^{ies} Economy of Scale

Grosch (1953) suggested that the
according to the performance/price relationship;

2

$$\text{Performance} = \text{constant} \times \text{Price}$$

Solomon, 1966;

Several studies [Bell and Newell, 1971; Knight, 1966; Phister, 1976; Sharpe, 1969; Turn, 1974] have examined whether this is true for a given set of machines. *On the other hand* ~~conversely~~ *price machines using this* it is possible to ~~use the~~ *clearly* relationship in pricing. It is desirable that the performance increase more rapidly than price for improved operating economy. For nearly all man-made objects (e.g., transportation vehicles, electricity generation, buildings) there is usually some economy of scale because there are high fixed costs that do not increase as rapidly as the ~~size~~ *output* of an object increases. ~~We might also conjecture that there is only~~

For computers, factors leading to economies of scale often apply over several dimensions. The same software can be used on many models. Sales and field service people can attend to a wide range of equipment. Manufacturing facilities can be adapted to produce different models.

A topic covering different sizes of machines would be incomplete without a discussion of economies of scale.

{ A topic covering different sizes of machines
would be incomplete without a discussion of
economies of scale.

basis).

Minicomputers are produced and distributed in a variety of ways and levels-of-integration from: printed circuit boards containing the electronics; to boxes which hold the processor, primary memory, and interfaces to other equipment; to complete systems with peripherals oriented to solving a particular application(s) problem. The price range(s) for the above levels-of-integration, in 1977⁸, are roughly: 500 to 2,000; 2,000 to 50,000; and 5,000 to 250,000.

7. ^{ies} Economy of Scale

Grosch (1953) suggested that there was an economy of scale for computers according to the performance/price relationship:

2

$$\text{Performance} = \text{constant} \times \text{Price}$$

Solomon, 1966;

Several studies [Bell and Newell, 1971; Knight, 1966; Phister, 1976; Sharpe, 1969; Turn, 1974] have examined whether this is true for a given set of machines. *On the other hand* conversely it is possible to *price machines using this* use the relationship in pricing. *clearly* It is desirable that the performance increase more rapidly than price for improved operating economy. For nearly all man-made objects (e.g., transportation

vehicles, electricity generation, buildings) there is usually some economy of scale because there are high fixed costs that do not increase as rapidly as the *output* ~~size~~ of an object increases. ~~We might also conjecture that there is only~~

For computers, factors leading to economies of scale often apply over several dimensions. The same software can be used on many models. Sales and field service people can attend to a wide range of equipment. Manufacturing facilities can be adapted to produce different models.

a linear relationship between price and performance and the square law relationship is only apparent (see Fig. Groschlin) because the curve roughly "fits" for a factor of 4 increase in price. ~~For two alternative ways of performing the same function at different price scales, the higher priced object provides proportionately more performance.~~ According to the IBM Telex papers, the goal of IBM's 360/370 models was to provide a factor of 3 (not 4) in performance each time the price doubled. Similarly, the higher PDP-11 models provide proportionately more performance. This aspect of design will be discussed in the following section.

Because the studies do not cover wide price ranges, there is some doubt that the square law holds. Indeed, over a narrow range (a factor of four), a linear approximation to the data would appear to fit as well as the square law does. See Fig Groschlin.

R While ~~the authors~~ ^{we} believe that there is some economy of scale (although certainly not a square law relationship), it is important to understand why such a relationship might exist. The high overhead, linear approximation to square law is one clear explanation.

A
~~The only~~ computer component that could be predicated on a square law relationship ~~has been~~ ^{is} the core memory. (~~see the PDP-8 core memory structure, page 00~~) There is an overhead cost associated with the base packaging, power and interface. The electronic selection is square law; a doubling of the selection circuitry provides access to a four times larger stack. All other costs are roughly linear, although we might expect the manufacturing cost for larger stacks to follow some economy of scale since there is a high set-up cost to threading core memories.

Replace with G

automatically holds

Economy of scale ~~does hold~~ if we use memory-size x processing-rate as the definition of performance. ~~Current memories operate with very low overhead~~

hardware, thus, there is a great dynamic range for economy of scale beginning with a single IC and going to the largest memory array--provided a processor can operate on the resultant data.

not
On the other hand, the more traditional view is that performance is simply the number of accesses/second into the memory. Therefore, there is no economy of scale in modern computers over the range of the smallest to the largest!

Using memory-size x processing-rate, note that performance increases as the price squared:

1. Processor cost is either negligible (optimistically) or it is proportional to memory size (e.g., a processor is added each time a memory chip is added). Memory size and memory speed can be traded-off equally in an application; thus, performance is their product. Note, this is akin to saying the performance of an automotive vehicle is the product of its speed times the number of passengers it carries.

$$\text{Performance} = \text{Memory-data-rate} \times \text{Memory-size}$$

2. In 1978, 4 Kbits of memory sell for about \$25 or \$.05/byte:

$$\text{Memory-size (in bits)} = 4K/25 \times \text{Price (in \$)}.$$

3. Each 4 Kbit memory chip can be accessed at a rate of 2 Mhz:

Memory-data-rate = $2M/25 \times \text{Price}$.

4. Therefore, assuming an arbitrary number of chips can be accessed in parallel by having either a 0 cost processor or proportionally priced processor, the performance is:

$$\text{Performance} = (8 \times 10^9 / 625) \times \text{Price}^2$$

$$\frac{2M}{25} \times P \times \frac{4K}{15} \times P^2 = 8.10^9 / 625 \times P^2$$

Note that using this measure says nothing about how processing is connected with memory. All these structures would appear to obey Grosch's law:

1. Each 4K chip could have an embedded processor which accessed it at a 2 Mhz rate. The processors could be totally interconnected, somehow to work on a single problem or they could be totally distributed and even completely disconnected.
2. Each time another 4 Kbit chip is added to a single system, 2 Mhz more processing is required at a price proportional to the incremental memory chip price. Only in this way, would there be a square law relationship associated with memory size additions.

For practical fixed width and fixed speed uniprocessors, performance would increase linearly with price since memory size is the only factor that can be varied after a base design is established.

¹Using Amdahl's constant when 1 instruction/second requires 1 byte of primary memory.

Remove

G

A second example is as follows. ~~If we use~~
The definition of performance,

$$\text{performance} = \text{memory size} \times \text{processing rate}$$

is akin to saying that the performance of an automotive vehicle is the product of its speed and the number of passengers it carries. If we use this definition, then performance does increase as the square of memory price, since both factors (size and rate) are each a function of memory price. To derive this we proceed as follows.

Let P = price of the memory on the system.
Assume the use of a $2^k \times 1$ memory chip and a memory system n bits wide, and further assume that the processor can use 100% of the memory data rate. To supply concrete cost and performance parameters we will use the 4 Kilobit chip; in 1978 it has a cost about \$25 and has a cycle time of (at the processor) of about 500 nanoseconds.

Then,
processing rate = memory data rate

$$= 2 \cdot 10^6 (n + m) \quad \text{where } m = \text{number of chips in the processor}$$

$$= 2 \cdot 10^6 (n + kn) \quad \text{since } m \text{ is a function of } n$$

$$= 2 \cdot 10^6 k' n$$

$$= k'' \frac{P}{25} \quad \text{since } n = \frac{P}{\text{price per chip}}$$

and,

$$\text{memory size} = \frac{P}{\text{price per bit}}$$

$$= \frac{4096}{25} p$$

$$\text{since price per bit} = \frac{25}{4096}$$

Substituting, we get

$$\text{performance} = \frac{4096}{25} p \cdot K'' \frac{p}{25}$$

$$= K''' p^2$$

8. Designing Compatible Machines ^{are designed to span a} ~~Over a~~ Price, Performance and Time Range

Nearly all modern computer designs are part of a compatible computer family which extends over price and time. Compatibility considerations are based on the economic necessity to utilize a common software base. The users' investment in software dwarfs that of the manufacturer, assuming the machine is successful. For example, if there is only one man year of software investment associated with the 50,000 PDP-11s, and each man year costs about \$40,000 and produces something on the order of 5,000 instructions, then there is a cumulative investment of 2 billion dollars and 250 million lines of program for the PDP-11. This investment is roughly the same scale as the original hardware cost.

Since technology provides such rapid changes over the generations, it is obvious that there must be backward (in time) compatibility in order to build on and preserve the user's program base. He must be able to operate programs unchanged to take advantage of improvements brought about by technology changes.

In a similar way, compatibility over a range of machines, at a given time is desirable so a user may select a machine that matches his problem set, while having the comfort that the problem can change and there will be a sufficiently large or small machine. As the problem changes with time, it is also desirable to have the appropriate, competitive, compatible machine.

Thus the goals for a complete, compatible computer product line (i.e., set of products) might be:

¹Using Amdahl's constant when 1 instruction/second requires 1 byte of primary memory.

1. the widest range of products in terms of price offering;
2. the largest economy of scale factor for performance that separates the models;
3. the smallest number of models to minimize costs of design, manufacture, selling and spares inventory;
4. the most cost-effective/competitive products (nearest to competitive machines) for each price in the space; and
5. backward compatibility with past machines.

Goals 3 and 4 are contradictory. How can the number of models be minimized, while at the same time providing competitive machines over a continuous space? In Bell and Newell (1971) we posited a multiprocessor alternative to the System/360 to cover the range. We still believe this is the best, and only way to meet both goals! Furthermore it provides increased reliability and the ability to easily upgrade performance with minor additions.

where
In Chapter 11 we assess the PDP-11 after three sets of implementations, we discuss ~~the~~ quantitatively the performance range spanned by the 11. We compare it with the span of the S/360 and compare the IBM System 360/370 computers have been the most compatible, while contrasting the different techniques used. supplying the widest equipment range. Models were separated by a factor of 2 in price and 3 in performance when the System 360 was introduced.

The initial 7 System/360 models included: 20, 30, 40, 50, 65, 75, and 91, and

models: 25, 44, 85, and (a few) 95 were later added. Since the ^{price} performance range was about 128, each model covered the intended factor of two in performance. ~~price~~

Adding 3 models caused each machine to cover a factor of 1.6 in price. The relationship between the performance and price (Bell and Newell, 1971):

$$1.36$$
$$300 = 65$$

The DEC PDP-11 family in 1977 is shown in Table PDP-11 Models.

Table PDP-11 Models

Model(s)	Price Range (in K\$)	[mid]	Performance (Scientific)
03	10 - 30	[15]	3
04	20 - 40		1
34	30 - 80		10?
45	45 - 150		40
55	60 - 200		70
70	95 - 300	[250]	65

For the price range of $300/10 = 30$, each of the six models covers a factor 1.76, and if we assume 04/34 and 45/55 are a single model then each model covers a factor of 2.34.

$$1.76^6 = 30; \quad 2.34^4 = 30$$

The relationship between performance and price, using midpoint prices over the price range of 250/15 (16.67) is: ~~1.51~~ ^{and scientific performance}

$$1.51^7 = 16.67$$

View 5: Functional Characterization of Computer Systems

In the first view we emphasized the computer structure (the system) as the particular object of design and described it in terms of inter-related design (and associated components) disciplines. The third view bound the components to nested levels physical levels of integration. In view four, the computer was considered to be a black box consisting of something that could be specified in terms of cost and performance that evolved with time.

Now we have to present a more refined view of information processing in order to more fully segment computers from one another and be able to identify those which are applicable for a given table (function).

Early computers were grouped simply into scientific and commercial denoting

9. The analysis invites further study.

We have given a qualitative explanation of the effect of technology on the computer industry. We emphasise that our view is an engineering view. We are ^{neither} ~~not~~ technology historians nor economists. The twenty years described in this book and the individual cost and performance measures surely invite analysis by professionals. The studies reported in [Sharpe, 1969] are a good departure point.

View 5: Computer Classes: ^{Functional} an Applications/~~Function~~ View

There is remarkably little shaping of computer structure to fit the function to be performed. At the root of this lies the general-purpose nature of computers, in which all the functional specialization occurs at the time of programming and not at the time of their design.

The shaping that does take place uses five primary techniques.

1. PMS-level configuration

A configuration is chosen to match

~~This is specialized~~ to the function to be performed. The user (designer) chooses the amount of primary memory, the number and types of secondary memory, the types of switches, and the number and types of transducers to suit his particular application.

2. Physical packaging

Special environmental packaging is used to specialize a computer system for certain environments, e.g., factory floor, submarine, or aerospace applications.

3. Data-type emphasis

Computers are designed with data types (and operat^{ions}~~es~~ to match) that are appropriate to their tasks. Some emphasize floating-point arithmetic, others string handling. We ^{place}~~place~~ special-purpose processors, such as Fast Fourier Transform processors, in this category also.

4. Operating-system

The generality of the computer is used to program operating systems that emphasize batch, time-sharing, real-time, or transacting processing needs.

Current Dimensions of Use

Bell and Newell [1971: Chapter 3] traced the evolution of use from the early days, when just two classifications -- scientific and commercial -- were needed, to the point where they identify seven different functional segmentations: scientific, business, control, communication, file control, terminal, and time-sharing. We refer the reader to their exposition. Very little has changed, in terms of functional characterization since then. Two points are worthy of comment. First, file control computers are still yet to materialize as mainstream separate functional entities, although there have been isolated cases, e.g., the IBM 3850 mass-storage system. The second remark applies to terminal computers. They have evolved to a much higher degree than expected. Because the control for terminals (e.g., calculators) is done with microprocessors, every terminal includes a stored program computer. Given this generality, it is a small matter to provide the terminal user with facilities to write programs. This phenomenon affects the evolution of time-sharing (when we use the term to denote close man-machine interaction as opposed to shared use of an expensive resource).

An alternative classification, which we find useful today, is the segmentation scheme shown in Table Fundisc. It is based on the intellectual disciplines and environment (e.g., home based) using and developing the computer systems.

It shows the evolving structures in each of the disciplines...hence, one can see that nearly all the environments evolve to provide some form of direct, interactive use in a multiprogrammed environment. The structures that interconnect to mechanical processes are predominately for manufacturing control. Other environments, such as transportation, are also basically real-time control. Another feature of discipline-based functional segmentation is the fact that each of the disciplines operate on different symbols.

Commercial (or financial control) based environments hold records of identifier names for entities (e.g., part number) and numbers which are values for the entity (e.g., cost, number in inventory).

Table FunDisc: Discipline/Environment Based Functional Segmentation Scheme

Commercial environment [financial control for all industry, retail/wholesale distribution=billing, inventory, payroll, accounts receivable/payable]
=records storage and processing
traditional batch transaction
processing against data base business
analysis (includes calculators)*

Scientific*, engineering and design based environment
=numbers, algorithms, symbols, text, graphs storage and processing
traditional batch computation*
data acquisition*
interactive problem solving*
real time (includes calculators and text processing)
signal and image processing*
data base [notebooks and records]

Manufacturing environment
=record storage and processing
batch*
data logging and alarm checking
continuous real time control
discrete real time control
machine based
people/parts flow

Communications and Message Based

(Message/Text transmission switching, storage and processing)

message switching

front end processing

store and forward networks

speech input/output

terminals and systems

word processing, including computer conferencing and publishing

Transportation systems

network flow control (excludes communication nets)

on-board control

Education-based (Computer Assisted Instruction)

=algorithms, symbols, text storage and processing

drill and practice

library storage

Home-based using TV set

(entertainment, record keeping, instruction)

*Implies continuous program development

The scientific, engineering, and design-based disciplines use various algorithms for deriving symbols or evaluating values. Text, graphs, and diagrams are the major ways of representing objects, and have to be processed. For these environments, we have seen the computer change from a calculator (as it was initially funded to do trajectory calculations for ballistic weapons) to a sophisticated notebook for keeping specifications, designs and scientific records. It has also evolved for direct recording and analysis of time varying signals and images. Initially the computer was only used as a transducer to collect data ^(with some preprocessing) from physical phenomena to be analyzed later on larger machines, ~~with some preprocessing (encoding)~~. Eventually, the computer was used in direct analysis and control. Now, many transducers have (or will have) computers embedded in them in order to encode information at a high level so that its output does not have to be processed by another computer. These will invariably be connected to other larger computers in a network

fashion to handle notebook graphical display and control functions. This corresponds to the intelligent terminal that is prevalent in the human interactive systems where processing is done at the lowest possible level and only the meaning of symbol is transmitted not the values of individual, time varying samples.

Manufacturing environment computers have evolved from a simple record keeping function which is quite similar to records in the commercial environment to direct on-line human control in a way identical to the other financial control-based disciplines.

Process-control computers have evolved from their initial use as assisting human operators (controllers) with data logging and alarm condition monitoring to full control of processes with either human or a second computer backup. The structure of the computer and the control task varies widely depending on whether it be a continuous process (e.g., refinery, rolling mill) or a discrete process (e.g., warehouse, automotive, appliance manufacturing).

Transportation for aircraft, trains, and eventually automotive vehicles is a form of real time control that uses both discrete and continuous control. Control is carried out in two parts: on board the vehicle and the network (e.g., airspace, highway) that carries the vehicles. The transportation control function dictates three unique characteristics for the computer structure:

1. very high reliability. Society has placed such a high value on a single

human life that all computers in this environment can not appreciably raise the likelihood of a fatality.

2. very small size for on-board computers.
3. extreme operating and storage temperature range for on-board computers - especially for automotive vehicles.

Communications and message based computers have evolved from telephone switching control, message switching, and front ends to other computers to be the dominant part of a communications system. With these evolving systems, the communications links have changed from analog-based modulation representation and transmission to sampled-data, digital transmission. By using all-digital transmission, data and voice (and video) can ultimately be used in the same system. Voice transducers enable speech communications with the computer.

Word processing (i.e., creation, editing, and reproduction) together with the long term storage and retrieval, and transmission to other sites (i.e., electronic mail) have evolved from several systems:

1. Conventional torn-tape message switching (e.g., TWX, Western Union, Telex).
2. Terminals with local storage and editing (i.e., flexowriters, teletype ASR's, magnetic card/magnetic tape automatic typewriters, and the evolving standalone word processing terminals.

3. Large, shared text preparation systems for centralized documentation preparation, newspaper publication, etc.
4. Large, systems with central filing and transmission (distribution). These will negate need for substantial hard copy. With these systems, text can be prepared either centrally with the system,, or with local intelligent word processing systems.
5. Computer conferencing. People can sit at terminals and converse with others without leaving their office.

The education-based environment implies a system which is a combination of transaction processing (for the human interaction part), scientific computation as the computer is required to simulate real world conditions (i.e., physical/natural phenomena) and information retrieval from a data base. These systems are evolving from the simple drill-and-practice systems which use a small, simple algorithm; through simulation of particular real-world phenomena; to knowledge-based systems which have a limited, but useful, natural-language-communications capability.

Home based computers are beginning to emerge. The dominant use to date is in providing entertainment in the form of games that model simple, real-world phenomena, e.g., ping-pong. Appliances are beginning to have embedded computers that have particular knowledge of their environments. For example, computer-controlled ranges can cook particular food in fairly standard ways. Alternatively, cooking can be controlled by embedded temperature sensors.

Simple calculators to record checkbooks have existed for quite some time. These will soon evolve to provide written transactions for recording and control purpose. Many domestic activities are in essence scaled-down versions of commercial, scientific, education, and message environments.

Convergence Towards Generality

As we observe the evolution of each machine class (super, mainframe, minicomputer, microcomputer) and the hand-held calculators we can see several cases of machine structures which begin as highly specialized and evolve to being quite general. We believe that this trend will become more marked.

Operating systems take on multiple functions as they evolve with time. Users specify additional needs and operating systems designers like to add function. Thus we see a COBOL run-time environment added to a simple FORTRAN-based real-time operating system. At the next stage a comprehensive file system might be added.

In a computer installation using large, highly general computers, minicomputers are installed to offload the large computers. The first application of the minicomputer is thus on a well-defined single problem. Soon more problems are added and the minicomputer system, with the help of a general-purpose operating system, is soon performing as a general computation facility. The offloading cycle begins again.

Part of this phenomenon is due to the inherent generality of a computer, and

part is a consequence of constant-cost evolution. This applies also to calculators. For example, the early scientific calculators evolved from just having logs, exponentials and transcendental functions to include statistical analysis, curve fitting, vectors and matrices.

Machines, then, evolve to carry out more and more functions. We believe that the prime discriminant is data type. Figure datatype shows an estimate of data type usage by application. We have postulated mostly high-level data types, e.g., process description. We strongly warn that this distribution is only a guess. Attempts to measure such distributions to date have not shown marked differences across applications (except for numerical vs non-numerical) because the data types have not been of a sufficiently high level.

VIEW ⁶7: The Practice of Design

Whereas in previous sections we gave different views of the object being designed, here we present a view of the process which gives rise to the object. We take the position that, by and large, computer design is like any other design. Two models of design, those of Asimow and Simon, are presented. We then conclude the view with some remarks on factors that particularly influence computer design.

Asimow's model

In Introduction^{to}~~/Design~~ [1962], Asimow gives a general perspective of engineering design and how the formal alternative generators and evaluating procedures, i.e., mathematical programming, are used, and where these formalisms break down and don't apply. Asimow defines engineering design as "a purposeful activity directed toward the goal of fulfilling human needs, particularly those which can be met by the technological factors of our culture." He distinguishes two types of design: design by evolution and design by innovation. We see examples of both in this book; however, we must warn that most computer design is evolutionary. Asimow's first diagram (Fig. Asimow) called, Philosophy of Design, shows what we believe to be basic design process. He lists the following principles [Asimow, 1962: 5-6].

1. **Need.** Design must be a response to individual or social needs which can be satisfied by the technological factors of culture.
2. **Physical realizability.** The object of a design is material good or service which must be physically realizable.

3. **Economic worthwhileness.** The good or service, described by a design, must have a utility to the consumer that equals or exceeds the sum of the proper costs of making it available to him.
4. **Financial feasibility.** The operations of designing, producing, and distributing the good must be financially supportable.
5. **Optimality.** The choice of a design concept must be optimal among the available alternatives; the selection of a manifestation of the chosen design concept must be optimal among all permissible manifestations.
6. **Design criterion.** Optimality must be established relative to a design criterion which represents the designer's compromise among possibly conflicting value judgments that include those of the consumer, the producer, the distributor, and his own.
7. **Morphology.** Design is a progression from the abstract to the concrete. (This gives a vertical structure to a design project.)
8. **Design process.** Design is an iterative problem-solving process. (This gives a horizontal structure to each design step.)
9. **Subproblems.** In attending to the solution of a design problem, there is uncovered a substratum of subproblems; the solution of the original problem is dependent on the solution of the subproblem.
10. **Reduction of uncertainty.** Design is a processing of information that results in a transition from uncertainty about the success or failure of a design toward certainty.
11. **Economic worth of evidence.** Information and its processing has a cost which must be balanced by the worth of the evidence bearing on the success or failure of the design.
12. **Bases for decision.** A design project (or subproject) is terminated whenever confidence in its failure is sufficient to warrant its abandonment, or is continued when confidence in an available design solution is high enough to warrant the commitment of resources necessary for the next phase.
13. **Minimum commitment.** In the solution of a design problem at any stage of the process, commitments which will fix future design decisions must not be made beyond what is necessary to execute the immediate solution. This will allow the maximum freedom in finding solutions to subproblems at the lower levels of design.

14. **Communication.** A design is a description of an object and a prescription for its production; therefore, it will have existence to the extent that it is expressed in the available modes of communication.

Asimov goes on to define the phases of a complete project.

1. **Feasibility study.** The purpose is to determine some useful solutions to the design problem. It also allows the problem to be fully defined and tests whether the original need which initiated the process can be realized. Here the general design principles are formulated and tested.
2. **Preliminary design.** This is the sifting, from all possible alternatives, *to find* a useful alternative on which the detailed design is based.
3. **Detailed design.** This furnishes the engineering description of a tested and producible design.

While ~~these~~ *the above* are the primary design phases, there are four ^{succeeding} phases resulting from the need for production and consumption by the outside world.

4. **Planning the production process.** This is really another design process which is simply a special case of design. The goal is to design and build the system that will produce the ~~computer system~~ *object*.
5. **Planning for distribution.** This activity includes all aspects related to sales, shipping, warehousing, promotion, and display of the product.

6. **Planning for consumption.** This includes maintenance, reliability, safety, use, aesthetics, operational economy, and the base for enhancements to extend the product life.

7. **Retirement of the product.**

Obviously all these activities overlap each other in time, and interact as the basic design is carried out. For example, any design that doesn't consider the distribution and consumption phases will fail. Historical information appears to constrain the first phases of the design. In reality, the later phases occur concurrently with the design, but with less intensity as the product moves from phase to phase. Phister (1974) posits a model of this process (Figs ~~4.21.3~~ ^{Phister 1} and ^{Phister 2} 4.21.4) and gives the amount of time spend in each activity (Fig. ~~4.21.7~~ ^{Phister 3}) for a hardware product.

The Generate and Test Model

A more abstract model of design is the one that Simon uses for human problem solving; it is called generate (posit a structure--solutions) and test (the alternatives). In The Sciences of the Artificial, Simon [1969] discusses the science of design and breaks the problem into: representing the design problem alternatives, the search (i.e., generating alternatives), and computing the optimum. When it is too expensive to search for the optimum, as is often the case, one resorts to selecting satisfactory (he calls them satisfying) alternatives. For most parts of computer design, the design variables are selected on the basis of satisfactory (satisfying) rather than optimum choice. Simon also discusses the tools of design, including

simulation as an alternative to building the complete system, and as a way to evaluate the behavior of various alternatives.

In his discussion of the importance of the design hierarchy, Simon introduces the notion of "architecture of complexity". The first ~~four~~^{three} views of this chapter have been influenced by this hierarchies view. In practice, designers and managers spend a great deal of effort in structuring complexity by various levels and design activities.

An associated problem is that of design representation. The more representations one has, the larger is the number of design problems that can be tackled, and hence the closer one can get to a global optimum. Most disciplines have ~~both~~^{at least two:} schematic and visual representations. In chemical engineering, heat balance is obtained by x, not from a plant piping diagram. In the design of power supplies, transformer design is accomplished using equivalent circuits, not ~~the~~^a physical representation. In the design of computer buses, most designers work with timing diagrams; however, alternative representations are state diagrams and Petri nets.

In general, the importance of alternative representations in computer engineering is not well understood. The large number of representations that do exist at the programming level is deceptive. There are many different algorithmic languages, but they differ mostly in syntax, not in semantics.

Classical Optimization

It is too simplistic to think that computer design should be a well-defined activity in which we can employ mathematical programming to obtain optimum *solutions*.

There are five major problems.

1. the cost function is multi-variable
2. the primary measure, performance, is not well understood
3. the objective function that relates cost and performance is not understood
4. objectives are not as objective as they look
5. there is a dynamic aspect (because the technology changes rapidly) which we find hard to quantify.

We use the following extract from a discussion of design given in [Bell, Grason, and Newell, 1972]^{: 23-24} to cover the problems of objectives and evaluations.

Objectives can often be stated as maximizing or minimizing some measure on a system. A system should be as reliable as possible, as cheap as possible, as small as possible, as fast as possible, as general as possible, as simple as possible, as easy to construct and debug as possible, as easy to maintain as possible -- and so on, if there are any system virtues that we have left out.

There are two deficiencies with such an enumeration. First, one cannot, in general, maximize all these aspects at once. The fastest system is not the cheapest system. Neither is the most reliable. The most general system is not the simplest. The easiest to construct is not the smallest, and so on. Thus, the objectives for a system must be traded off against each other. More of one is less of another and one must decide which of all these desirables one *wants* ~~wasn't~~ most and to what degree¹.

The second deficiency is that each of these objectives is not so objective as it looks. Each must be measured, and for complex systems there is no single satisfactory measurement. Even for something as standardized as costs there are difficulties. Is it the cost of the materials -- the components? Do we use a listed

¹Belief in the perversity of nature that forces all good things to trade off against each other should not be carried to extremes. The field of digital systems provides a counter-example. Digital systems now, compared to digital systems even a few years ago, are: faster, cheaper, smaller, more reliable, simpler, and easier to maintain. In short, they are better in every way, and nothing had to be traded off in the final performing system to obtain them. Within this, there always exist small trade offs, e.g., between speed and

retail cost or a negotiated cost based on volume order? What about the cost of assembly? And should this be measured for the first item to be built, or for subsequent items if there are to be several? What about the costs of design? That is particularly tricky, since the act of designing to minimize costs itself costs money. What about cost measured in the time to produce the equipment? What about the cost of revising the design if it isn't right; this is a cost that may or may not occur. How do we assign overhead or indirect costs? And so on. In a completely particular situation one can imagine an omniscient designer knowing exactly which of these costs count and being able to put dollar figures on each to reduce them all to a common denominator. In fact, none of us knows that much about the world we live in and what we care about.

The dilemma is real: there is no reducing the evaluation of performance in the world to a few simple numbers. The solution is to understand what systems objectives are: they are guides to understanding and assessing system behavior in various partial aspects. Various measures for each type of objective are developed, and each shows something useful. Since all measures are partial and approximate (even conceptually), rough and ready measures that are easy to make, display and understand are often to be preferred to more exact and complex measures. Standard measures are to be developed and used, even if not perfect. Experience with how a measure behaves on many systems is often to be preferred to a better, but unique, measure with which no experience exists.

We do not treat systematically all the different system measures. Many of them are illustrated throughout the book. However, Table BGN does provide a guideline, listing in one place the components that contribute to overall cost and performance.

Table BGN: Cost and performance components for a system.

Cost components

- designing
 - specifying
 - designing (drawing, checking, verifying)
 - prototyping
 - packaging
 - describing (documenting)
 - designing the production system
 - standardizing
- producing
 - buying (parts)
 - assembling
 - inspecting

testing
selling and distributing
using
 understanding
 configuring (i.e., user designing)
 purchasing
 applying
 operating in the environment (heat, humidity, vibration, color, power, space)
 repairing
 remodeling

Performance

 designing, producing, selling
 using
 environment
 for a single task (operation times, operation rate, memory size, utilization)
 for a set of tasks
 reliability, availability, maintainability, and error rate
 mean time between failures (mtbf)
 availability (percent)
 mean time to repair (mtrr)
 error rate (detected, undetected)

Conflicts (Tradeoffs) Among the Various Factors

It is necessary to point out some conflicts among the various activities. The list of conflicts is very real and taken from experience.

~~[rewrite following paragraph]~~

each of the packaging
System Cost Versus Component Cost. ~~Because~~ DEC sells products at ~~all~~ levels of integration (chips...~~complete~~ turnkey application systems), *Because each product is constructed from lower packaged levels, i.e., the levels model strictly applies,* ~~one has to pick the level at which the product is to be most cost-effective. The lowest level it is very difficult to have designs that are optimally competitive at usually is the most competitive. To the extent price (and generality) is every level. For example, if DEC sold just hardware systems (cabinet traded off, a product becomes more expensive to apply at a particular higher level) it would not need a boxed version of its CPU's. It would then delete the box level and the price of the systems product would be proportionately lower. When primitives are to be used as building blocks, there is a cost associated with providing generality, hence modularity. Some boxes are overpowered, for example, for most of their final applications. Generality is sometimes traded to obtain low-cost, non-extendable systems such as the RXT-11.~~

Initial Sales Price Versus User Life Cycle Cost. There is a cost associated with parts that break and have to be repaired and maintained. Nearly every part of the computer can be improved over a range of a maximum of a factor of 10 to provide increased reliability (extended MTBF) for a price. To the extent these costs are added, means the product will be less competitive in terms of a higher purchase price. However, if the total life cycle costs are considered, the product may still be ~~lower~~ ^{better} even at the higher initial cost.

Reliability, Availability, Maintainability (and Producibility) Versus Performance. By designing to take advantage of the fastest components and operating them at the limit of their capability, one is able to have increased performance. In doing so, the tradeoff is clear, producibility, reliability (error rate), and maintainability (ease of fixing) all generally suffer.

Performance Versus Cost. This is the most traditional design tradeoff. In addition to the conventional product selection, the planning of a computer family further increases selection/tradeoff process.

Early Shipment Versus Product Life and Quality. By delivering products before they are fully engineered for manufacture, one is able to improve the ship date but at a significant risk. If faults are found that have to be factory ~~retrofit~~ ^{retrofitted in the} or field ~~retrofit~~, the cost far outweighs any early product availability.

Another Longer Design Versus Product Life. By taking longer to design, a product can be designed in such a way that it is easier to enhance.

Operating Environment Versus Cost. Here there are numerous tradeoffs even within a conventional environment. In each of the packaging dimensions: heat, humidity, altitude, dust, EMI, etc. there are similar tradeoffs that may appeal to unique markets or may simply translate to increased reliability in a given setting. The Norden 11/34M is an example of packaging to provide a PDP-11 for the aerospace environment.

Responsible Disciplines

Computer science and electrical engineering are the responsible disciplines. From computer science we get many of the technical aspects, e.g., instruction-set architecture; the theory, e.g., algorithms and computational complexity; and software design, e.g., operating systems, language translators, that are applied in the practice of computer engineering. However, in their construction, computers are electrical. Thus, the discipline that has fundamental responsibility is electrical engineering. To illustrate this point we include as Table Maxims a set of maxims, due to Don Vonada, an experienced DEC engineer.

^{caps}
View 2: Levels of Interpreters

Our previous view was structural. This present view, that a computer system consists of levels of interpreters, is functional. It is the most uniform view. Whenever one uses the onion skin model, for example the one in Figure 15, one is taking this view. There are indeed many layers, but as we analyze the view we will discover that examples of layered pure interpreters are rare in practice.

An interpreter is a processing system which is driven by instructions and operates upon state information. The basic interpretive loop, shown in Fig. Iloop, is most familiar to us at the machine language level. It also exists at several other levels. For example in a BASIC interpreter, the instruction counter corresponds to the source statement number; at the microcode level a microprogram counter sequences through the machine states which interpret a machine instruction.

To formalize the notion of levels of interpretation we represent a processing system by the diagram in Fig. formalproc.

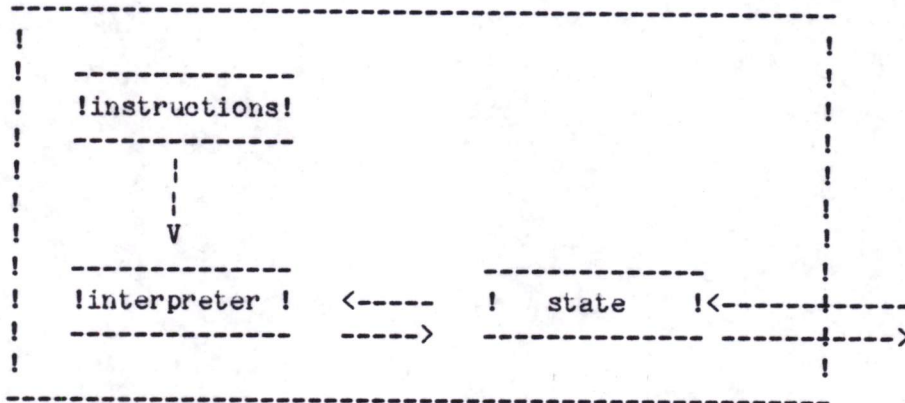


Figure Formalproc: A processing system

The state of an interpreter is either internal (components of the interpreting mechanism) or external (state variables of the level being interpreted).

Consider the four processing systems listed in Table four.

Table Four

Level 4	Instruction:	seat allocation request message
	Interpreters:	airline reservation system
	Internal State:	number of requests pending at this moment location of passenger list on a disk file number of lines connected to system rotational position of disk
	External State:	number of reserved seats on a given flight airline name for a given flight
Level 3	Instructions:	Fortran statement codes
	Interpreters:	Fortran Execution System
	Internal State:	memory management parameters

		user name main storage size location of disk files interrupt enable bits expression evaluation stack dimensions of arrays
	External State:	subroutine names values of data in arrays statement number program size value of an expression <u>do</u> -loop variable value printed characters on line printer
Level 2	Instructions:	machine language instructions
	Interpreters:	processor (CPU)
	Internal State:	machine architecture <i>program</i> defined registers condition codes program counters
	External State:	data in main memory disk-controller registers
Level 1	Instructions:	microcode
	Interpreters:	micro machine microprocessor
	Internal State:	instruction internal register flip-flops <i>holding error status</i> condition codes micro address <i>stack of microprogram subroutine links</i>
	External State:	machine architecture defined <i>program</i> registers <i>condition</i> card From Codes program counter

Each level implements the one above it. The level 1 system is a microprogrammed processor, implemented in real hardware, the machine seen by the logic designer. The level 2 system is the CPU. The level 3 system shown here is a FORTRAN-language processing system. The level 4 system is an airline-reservation system. This interpreter operates on messages received

from outside of the system, tests and modifies the state, and generates messages to send back. These four systems form the hierarchy shown in Fig. Hier. Each interpreter is sequencing through multiple steps in order to perform a single operation of the next-higher-level interpreter.

In practice, few systems are levels of pure interpreters, although the layers are present. There are two primary departures from the pure interpreter model: (a) high-level-languages are usually compilatively executed, and (b) some layers are bypassed when ~~[CM: include discussion on - levels is done with altogether or - mixture of comp + interp. or both]~~

Level 3 is done away with for most high-level language systems.

→ Compilative execution is a two-step process. Programs at level 4 are translated (compiled) into statements understood by the level 2 interpreter. These statements are then executed.

→ shows five levels of pure interpreters in practice. For most of the FORTRAN functions (type A) they are directly executed on the ISP interpreter. We call this bypass of levels by the name pipe. Figure pipes represents how the levels appear in practice.

Other pipes exist for type B functions in the OTS.
P Having presented the pure interpreter model we can ^{now return to} see the onion skin [CM: Return now to the onion skin layers] Figure 15 layered model and better understand how the different layers relate.

P Using the macromachine hardware (architecture) as a base level interpreter, it is most often extended with an operating system. There may be several operating system levels so that the machine can be built up in an orderly fashion. A kernel machine might manage and diagnose the hardware components (e.g., disks, terminals) and provide synchronizing operations so that the multiple processes controlling the physical hardware can operate quasi-concurrently. Next, more complex operations like the file system and

basic utilities are added, followed by policy elements such as facilities resource management and accounting. As viewed through the operating system, one sees a much different machine than that provided by the basic instruction-set architecture. In fact, the resultant machine is hardly recognizable as the architecture most usually given by a symbolic assembler. It includes the basic machine but has much more capable i/o and often the ability to be shared by many programs (or tasks).

Operating systems designers believe all these facilities are necessary in order to implement the next higher level interpreter--the standard language. The language level may include interpreters or compilers to translate back to the machine architecture for ALGOL, BASIC, COBOL, FORTRAN, etc. or any of the other hundred standard languages and their dialects. At the language level, a user again observes a common language machine.

Often an additional special language is used because an application can't be expressed easily using the standard language per se and it is necessary to have operations that are within the domain of the problem. Finally, using a special language, various application subprograms (a program library) are written to enable specific applications to be written. One then may build the application and finally the real user can get a problem solved in a cost-effective fashion provided there has been the right set of operations (languages) at each of the levels.

Sometimes the stratification is done to help manage complexity or to allow specialization of design activities. ~~and to go~~ Sometimes ~~from~~ the underlying layers are completely hidden from the user to aid ease of use [Mudge, 1973].

1. stratification intentionally done to manage complexity, allow specialize - a design level, ease of use, e.g., DIAL.
2. it doesn't really matter performance forces more compile or to do away with a level altogether. manage comp. add levels e.g., nanocode]

Finally, note that using fundamentally different outer layers for a common inner set of layers creates quite different machines, hence a set of onions. Therefore it is important to realize that when dealing with common core hardware, multiple operating systems, languages and applications a network of machines is formed, not just a single, layered machine.

In the final analysis, the number of levels is just another tradeoff.

→ Performance considerations lead us to do away with levels, complexity leads us to add levels.

^{Comp}
View 3: Packaging Levels of Integration

This is a structural view that packages the various components (hardware and software) into levels. The current levels for DEC ~~minicomputers and~~ ~~large-scale systems~~ are as follows.

- 9 applications
- 8 applications components
- 7 special languages
- 6 standard languages
- 5 operating systems
- 4 cabinets
- 3 boxes
- 2 boards
- 1 integrated circuits

This view is the most important in the book, because it shows how computer systems are actually structured and hence how their costs are structured.

It is the least general of the views and the most timely, i.e., easily dated.

It is a structural view of the object being sold, and as such, is completely a function of the technology, the organization building the system, and the marketplace. As we go through the view we will see how rapidly the constituent parts at each level have changed and how the computer industry allocation across the levels has changed.

Indeed, the view could be titled "Dynamic Levels of Integration". We will elaborate on the three major changes.

1. Changes in the hardware levels

The shrinking in physical size of functions has three effects.

- a. lower levels subsume higher levels
 - b. The semiconductor component supplier is forced to assume higher and higher level design responsibilities
 - c. Levels disappear
2. Changes in the software levels
- a. each level grows in size as more functionality is added over time
 - b. more levels are added as minicomputers are applied to a broader range of applications
 - c. functions migrate downward from level to level
3. Changes in the hardware/software interface
- a. software functions migrate into hardware for higher performance

The software levels were also discussed in the previous view. They are the operating system; the languages; application library; and applications levels.

Within each level of the software there can, of course, be an arbitrary nesting of levels using lower order primitives of the level in order to provide the top-most, integrated resulting design. There will be little or no discussion herein about these software levels except as they act as consumers (users) of lower level hardware.

Hardware levels

For the hardware levels, we might consider time to be the packaging view and base it on the physical levels of interconnection. In fact, it is the interconnection and packaging per se that determines the machine more than any other factors excluding the basic lowest component (now semiconductors) level technology. This base acts to constrain and limit all higher levels.

This constraining and limiting by the interconnection and packaging takes place because most manufacturing costs are associated with the physical structure--not the basic lowest level components that are interconnected. As interconnection levels must be introduced to build complex structures, many usually unwanted side-effects occur. The interconnection requires space and interfaces with cooling which in turn creates noise. ~~Long~~ ^{Long} interconnections increase signal transmission delays, and these reduce performance. ^{Signal} ~~Signal~~ transmission makes the computer susceptible to electromechanical interference and it can radiate ~~EM~~ ^{electromagnetic} waves that must be controlled. Finally, there is the domain of industrial design - aesthetics - and everyone is a self-based authority on appearance.

Figure 11 shows how the levels-of-integration prices versus time for small computers. The price depends partly on implementation and architecture word-length; so, as the word-length is made shorter, there is some economy, particularly for minimal computers. Note there is not a 4-bit computer shown at the board level, but only as chips. In fact, most hand held calculators are implemented using 4-bit, stored program computers with fixed programs that occupy a single IC. ~~and~~ ^{They} have no associated modules, backplanes, boxes and cabinets--only a single package that fits one's hand. We see a movement in

price versus time that is parallel; for various different scaled machines--which means that all constituent components have the same fractional use. An astute marketing-oriented person might ask, "How, with all the technology can we do something unique to get off the curve?" One answer: "Reduce prices by not providing a power supply and mounting hardware. Let the user provide all added-on parts and mount the computer as needed. In this way, the price, though not necessarily the total cost to the user is reduced. We'll sell at the board level." Computer Automation introduced the "Naked Mini" TM in 1972 at a lower component price so that users could supply more added value--packaging and power technology.

Chalmers

A similar effect can be seen in the 11 series, since the 11/20's introduction in 1970. Most models are sold at the box and cabinet level with module level options. In 1976 a module with 4,096 words of memory and processor (the LSI-11) was provided for \$600 as described in chapter 00. The boxed version costs, reflecting negligible improvement in packaging, since the 11/20 boxed version sold for x' .

Note from Fig. x , ^{that} semiconductors, the lowest level of technology, have the greatest price decline. This should continue as multiple dice are mounted on one substrate.

multi-die chips

Modules have a lesser price decline because they are a mix of: ICs, printed circuit boards, labor (and capital equipment to insert the components), and testing labor (and capital equipment). At the module level there have been negligible gains in fabrication and printed circuit board technology; and thus increased labor costs yield an over-all cost increase.

At the box level-of-integration, power supplies and metal or plastic boxes increase due to their labor intensive nature. The gains are ^{derivative: they are} by-products of using less power and less space. Finally as boxes are integrated (by people), and applied (by custom, people intensive programmers) the prices are approximately constant.

The changing domain of the semiconductor suppliers

In the early 70s Intel, North American Rockwell and other semiconductor companies began to use the higher semiconductor densities to reduce the number of levels of integration by packaging a complete processor-on-a-chip. These organizations had assimilated logic design, and were frustrated because their customers could really not identify higher functionality units (beyond memory) requiring on the order of 1,000 gates on a chip. Also, the speed of these high density units was quite low.

They discovered that the best finite state machine that one can make is just a simple computer because it provides the finite state machine plus the useful functions that are not covered by switching circuit theory.¹

Processors-on-a-chip have followed a very steep price decline of up to 50% price reduction/year. A processor costs anywhere between \$5-10.

Robert Noyce, of Intel, presented Fig. 12 in October 1975. It illustrates what is happening in the semiconductor industry; and I have added to it slightly showing the technology that DEC has assimilated with time. It shows

¹It is simply a small matter of programming to do something useful but we all know that programs cost \$1-\$100 per instruction to write.

the breadth semiconductor manufacturers have in technology, starting from semiconductor device level, through the view Noyce has of the various levels-of-integration continuing into end user applications.

The complete model

Figure 13 assigns ordinal numbers to the levels-of-integration we ~~want to~~ use throughout the book. The assignment ~~to~~^{of} a number to what might appear to be quite fuzzy, is our attempt to be convinced that we can structure this knowledge in precisely this fashion for quite some time--say 10 years. The numbers correspond to the structure of mini (and larger) computers. Figure 13 also shows how a group of the levels are compressed for smaller systems, and more spread out for larger systems. For very tiny systems such as a hand-held calculator, which ~~only~~^{only} consists of one chip, printed circuit board, keyboard and lights, power (perhaps a battery), package, operating system, and machine language application program, the entire structure has only two levels of integration--within the chip and within the calculator via the printed circuit board.

Figure 13 associates the physical levels-of-integration with: the corresponding levels of the abstract machine (and its language); and with the conceptual levels design disciplines. One can start from the top-most level and go down below the chips and on to the atomic level. One could argue whether the world is analog or digital at the atomic level, which is roughly level 0. A physicist could probably decompose a semiconductor system many more levels, but they do not correspond to machines.

The levels-of-integration model is that components at one level, are combined into a system at the next highest level in a hierarchy, ~~although the structure~~ is actually a lattice or network, and not a tree. Each level can, of course, be nested itself. A level also denotes that there is a single conceptual design discipline or set of interacting disciplines which determine the function, structure, performance and cost of the constituent level.

*However, as
Fig 14 shows,
the structure*

double-spaced please

View 8: ^{caps} ~~A~~ ^{The} Blaauw Characterization of Computer Design

Another framework (view) we use is based on the powerful, clear, ^{stated} distinctions ^{of} Blaauw ^[1971] ~~defines~~ ^(Blaauw, [1971]). He distinguishes architecture, implementation, and realization as three separable levels in the construction of anything, including computer structures.

The architecture of a computer system defines its functionality (behavior) as it appears to the machine-level programmer, namely the instruction set we call the ISP. The implementation is the actual hardware structure - the register-transfer level behavior and data flow organization of a computer. This also includes various algorithms for controlling a machine as it interprets an architecture. Realization encompasses the actual technologies used and includes the kind of logic, how it is packaged and interconnected. Realization includes all the details associated with the physical.

^{Modern} ~~An~~ architectures usually ^{have} ~~has~~ multiple implementations. For example, the LSI-11, 11/40, and 11/60 are different implementations of the same basic PDP-11 instruction set. ^{Some times, although rarely, a particular implementation has more than one realization.} For example, the IBM 7090 has ~~approximately~~ the same architecture and implementation (i.e., the RT structure) as the IBM 709. The difference lies in realization: the 709 used vacuum tubes, the 7090 used transistors. For a more recent example, two models of the PDP-11 architecture that share the same implementation are the 11/34 and Norden's ruggedized realization, the 11/34M. The latter uses militarized semiconductor components and placement as well as a different packaging and cooling system.

The following table attempts to clarify the distinguishing characteristics of

architecture, implementation and realization.

Characteristics of Design Areas (Blaauw and Brooks, Computer Architecture, 1978)

	Architecture	Implementation	Realization
Purpose	function	cost and performance	buildable and maintainable
Product	principles of operation	logical design	release to manufacturing
Language	written algorithms	block diagram, expressions	lists & diagrams
Quality measure	consistency	broad scope	reliability
Meanings (used herein)	ISP	RT-level machine; Microprogrammed machine ISP	physical realization; physical implementation sequential machine (at logic level)

In this book, we concentrate on the realization and implementation; ~~architecture~~ *instruction-set architecture* is discussed only insofar as it interacts with the other two characteristics.

Reconciliation of Design Views (Blaauw versus ^{Structural Levels} ~~Conceptual Disciplines~~)

It is important to try to reconcile the differences between the views of Blaauw and Brooks and ours because all of us engaged in computer engineering use the words, architecture, implementation and realization--quite often to mean different things. For architecture, although we have good congruence with the definition, we prefer to not limit the definition of architecture to just a machine as seen by a machine language programmer. Instead, we will use architecture to mean the ISP associated with any of the machine levels described in the interpretation levels view (page 00). Therefore, architecture

standing alone will mean the machine language the ISP. We will also use:
architecture of the microprogrammed machine as seen by a microprogrammed
machine's microprogrammer; architecture of the operating system as the combined
machine of operating system and machine language; and architecture of a
language for each language machine. For example, ALGOL, APL, BASIC, COBOL and
FORTRAN all have as separate and distinct architectures as a PDP-10 and a

PDP-11. *do. This use of architecture, since it describes behaviour, is quite consistent with Blaauw's. Moreover, when applied to software structures, Blaauw's framework fits well. There*

are two implementations, FORTRAN-IV-PLUS (an optimizing compiler) and FORTRAN IV (a threaded code comp system), of the one ANSI FORTRAN architecture. Moreover, ~~different~~ ^{different} implementations could use different realization techniques: some use BLISS, other ^{use} assembler language.

Although Blaauw and Brooks define implementation and realization clearly, these definitions aren't universal. *widely used. The main problem is that they are both sensitive to* Also note the two definitions interact. As engineers we have overused and intermixed them so that the two words are used interchangeably in the rest of the book to have roughly the same meaning (e.g., "The KI-10 was implemented using TTL H-series logic.") In the table, we have given our definitions for the two words to further relate descriptions back to these definitions if the reader chooses. Implementation is the Register Transfer level machine roughly the microprogrammed machine and realization is the physical realization to us the physical implementation in terms of packaging and technology.

idea

technology changes and hence in practice to slowly.

We have found that ~~the summary of~~ the most useful distinction is between architecture on the one hand and implementation ~~part~~ (subsuming realization) on the other. Seeing the distinction crisply enables one to preserve architectural compatibility between machine models, and this is crucial if users' and manufacturers' software investments are to be preserved. Implementation can then be as dynamic as we like, being continually changed by technology. Architecture must remain static for long periods (ten years is a common goal).

Kennice Wilkes, ^{in 1949,} only one month after his EDSAC computer was operational, and before any stored program computers ~~was~~ in the United States were operating, had already perceived the value in having a series, or set, of computers share the same instruction set. He said the following.

"When a machine was finished, and a number of subroutines were in use, the order code could not be altered without causing a good deal of trouble. There would be almost as much capital sunk in the library of subroutines as the machine itself, and builders of new machines in the future might wish to make use of the same order code as an existing machine in order that the subroutines could be taken over without modification."

indent
and
single
space

new page please

caps 8 Summary: ~~are~~
View 9: Computers ~~are~~ Surfaces in Computer System Space

is a summary.

In this last view we characterize computers by a number of distinguishing attributes so that they can be positioned as surfaces in a computer space. The function, structure, ^{and} technology dimensions of the space are grouped according to architecture and performance (see Table 6.Space). The values of the attribute are given in order of increasing time as the computer has evolved. When appropriate two or more of the attributes that are closely correlated with

one another are grouped together. (For example the processor state and addresses per instruction are closely correlated.) The original computer space published in Bell and Newell (1971) has been updated to account for omissions and advances, ^{These} which have caused extensions to the space both ^{within} in a dimensions and in added dimensions.

Since we are discussing only the DEC computers we make no claim that they traverse the complete space, nor even that they are the best example for a point in the space. We do annotate the table so that the reader can see how the DEC computer design examples in the rest of the book do fit the space, and have evolved with time.

Because we need to show the common design alternatives in a coherent way, we present the space,

Even though this book is ~~about examples of computer architecture and is not an attempt to be either definitive or introductory, we do need to show the common alternatives in a coherent way.~~ We also claim that the space is fairly

complete, right now and until the mid 80's ^{Since} ~~it is~~, all the existing computers can easily be positioned within ^{the space, we} ~~it~~. We assume the readers will nod with *We expect that it will be satisfactory until the early 80's.*

appropriate understanding when viewing the table and all the terms should be familiar to computer engineers.

~~Performance~~ **Appendix:** Performance

In this section we will discuss computer hardware performance, neglecting the operating system and language use because the latter so drastically affect performance as to make the discussion meaningless. Performance parameters are a combination of architecture (the ISP), hardware implementation and resources (the PMS structure) being acted on by ~~some program~~^(the use). For example, although a certain machine (hardware) can deliver 500,000 instructions/per second to a program, we must also define what the instructions are (the ISP) and how a particular program makes use of the instructions.

Most of the time we use overly simplistic hardware measures (e.g., instruction time) ^{can be used} to characterize machine performance. ^{for many cases.} The ultimate performance parameters have to be based on actual use (i.e., workload) parameters, otherwise there is no way to correlate the primitive ^{hardware discussed here} measures to real performance. Benchmarks of (including standardized benchmarks such as Whetstones for the algorithmic scientific languages) ^{and COBOL benchmarks for commercial applications} and synthetic or real workload provide the only real test by which performance can be compared. The purpose of this section is to discuss hardware performance generally, and then to sort out its various components.

When we measure performance, there is a tacit assumption that sufficient software exists to exploit a hardware structure, and that the transformation from this basic, lowest level hardware machine (i.e., the macro machine) to the user machine (as provided by a language such as COBOL or FORTRAN) is relatively constant across various architectures. As each level is crossed, a

transformation takes place requiring computational work. The form of the work with compiled languages is direct execution via the processor and run time support program and with interpreted languages the processor executes an interpretation program which indirectly interprets the final program) ^{data (i.e.,}

At the lowest level, the internal micro machine provides the architectural facade, the ISP, ~~for the macro machine and~~ ^{ing} operates at roughly 10 times the ~~later's~~ ^{of the macro machine} speed. Thus a machine executing 1 million instructions per second may have an effective microcycle time of 100 nanoseconds, ~~permitting~~ ^{for for executing} 10 million micro instructions/sec. At the next level, ~~a~~ ^{the a} conventional macro machine (ISP) executing 1 million instructions per second, is capable of perhaps 0.1 to 0.25 million higher level FORTRAN language statements (instructions) per second depending on the mix of built-in functions and external functions called.

It is difficult to use the simplistic constant ratio measures across each level of interpretation when comparing machines of differing classes (e.g., micro to super) because there is not a consistency of data-types (e.g., micros currently have no built-in real arithmetic, whereas minis do). However, for machines within a class (e.g., mini) where the data-types are implied by the class name, ^{so simplistic comparison} it is probably all right, since the two machines most likely have about the same data-types. Hence a count of the number of data-types reflecting the built-in operations is one of the more significant architectural performance indicators, whether it be for a micro machine, macro machine or a language machine. ↗

PMS (I.E., Resources) Performance Parameters

The PMS structure, with the corresponding attributes determining performance (e.g., memory cycle time, processor execution rate) provides the basis for understanding machines and comparing them with each other. Figure BPMS gives a PMS diagram of a basic computer, with the parameters that ^{to a first approximation,} ~~will be used to~~ characterize performance. ~~The diagram shows the structure and key performance parameters of each component as they provide a total computing environment.~~

Alternatively, one might use a more descriptive, or tabular form; but the goal ~~here~~ is to provide a structural/performance basis for parameterization, comparison, and specifying the finite resources of the computer ^{so that performance can be determined against actual workload}

It ^{is} imperative to consider the resource constraints, and the effect of their interaction as each next layer of a machine is designed. For example, a certain line printer requires buffer space (Mp. size) and processing time (Pc. speed) which is then unavailable at the next machine level (e.g., FORTRAN). Although a clear layering is needed to characterize each level, we will not carry on this layering beyond the hardware.

Bell and Newell (1971) argued that a machine (at any level) can be described with any number of parameters, and carried out the exercise for up to 5 parameters:

Number of parameters

allowed: 1 2 3 4 5

1 Pc(i.rate) - - Pc(op.rate)

single spaced

2	Mp(size)	-	-
3		Ms(size)	-
4			Pc(i.width)
5			No. of terminals

single spaced

Amdahl states a correlation between P_c and memory size (see Fig. Page 0)

Information rate between the processor and memory, is used as the processor speed indicator instead of the more conventional instructions per second. Compound indicators such as the product of processor speed times memory size to indicate basic computational performance ~~used in the earlier view~~ were not allowed.

also note that by having only a single parameter, say P_c (rate), and ~~somehow we must express Am~~

The following example shows 3 different architectures with 2 implementations of a stack architecture (~~one~~ one has the stack in the primary memory, Mp, and the other assumes the stack is in the processor, Pc, using fast registers). The hardware implementations are held roughly constant (the Pc-Mp data rate) and the architecture is varied in order to compare the effect on performance. Note the difference in the various measures in what should fundamentally be about the same performance for a given problem.

The benchmark program is the simple expression, $A := B + C$

Stack

Stack

1-address or

	<u>(top in Mp)</u>	<u>(top in Pc)</u>	<u>general reg.</u>	<u>3-address</u>
Program	push B push C add pop A	push B push C add pop A	load B add C store A	add B,C,A
No. of Instns.	4	4	3	1
Accesses	$4op'+3a+6d$	$4op'+3a+3d$	$3op+3a+3d$	$1op''+3a+3d$
Program size (bits*)	64	64	72	60
Bits accessed	$16+48+192$ =266	$16+48+96$ =160	$24+48+96$ =168	$12+48+96$ =156
Time to execute** (microseconds)	$0.5+1.5+6$ =8	$0.5+1.5+3$ =5	$.75+1.5+3$ =5.25	$.37+1.5+3$ =4.87
Statement exec. rate(actual performance)	$1/8 = .125m$	$1/5 = .2m$	$1/5.25 = .19m$	$1/4.87 = .21m$
Oper.rate	$2/8 = .25m$	$2/5 = .4m$	$2/5.25 = .38m$	$2/4.87 = .42m$

*single
spaced
please*

Inst. rate	4/8 = .5m	4/5 = .8m	3/5.25 = .57m	1/4.87 = .21m
Pc(i.rate)/ word length	32m = 1m	32m = 1m	32m = 1m	32m = 1m

*assumes: address/a = 16b; data/d = 32b; op = 8b; op' = 4b; op'' = 12b

**assumes a memory limited processor which can access 32b/microsecond

The statement execution rate (the actual performance) is the highest for the 3-address machine, reflecting the highest performance whereas the conventional measure (instns/sec) shows it to have the lowest performance (by a factor of 4) over the fastest machine. A more subtle measure, operation-rate, is correlated with the true benchmark statement execution rate. It should be noted, (ignoring the first machine, a stack machine with stack in Mp) that the information-rate is a good performance indicator - versus the conventional, but poor, instruction-rate measure. For more unconventional machines, instructions/sec. tends to become a significantly poorer measure. The various vector/array machines (e.g., ILLIAC IV, CDC STAR, CRAY-1) have single instructions to operate on at least 64 operands per instruction, hence instructions/second would be a poor measure.

Similarly hand held calculators have single instructions such as Sin, Polar to Cartesian co-ordinate conversion; ~~and~~ using anything but a final benchmark problem would be unfair. Accesses/sec. will be used as a Pc performance measure.

The multiprocessor case

For multi-processors the number of processors x the memory accesses/sec. roughly gives the total Pc.rate. Pc.rate can be computed more precisely by using the number of primary memory modules, M_p , and their data-rate as can be seen in Chapter 00 on the C.mmp computer. For a system where the memory access time, and the memory rewrite time equal the time for a Pc to operate on a word, the performance [Strecker, 1970] is roughly:

$$Pc.speed \text{ (in accesses/sec.)} = (m/t.access) \times (1 - (1 - 1/m)^p)$$

where m = # of M_p modules, and p = # of Pc's

Note that when $p = m = \text{large}$, the performance reaches an ~~asymptote~~ asymptote:

$$= m/tc \times (1/e)$$

double spacing

asymptote

In the case of multiprogramming systems (e.g., real time, transaction, and timesharing), the time to switch from job to job is ~~critical~~ *important if there is a high*. ~~This measure, process context switching rate, is one of the main attributes since most computer systems operate with some form of multiprogramming.~~

~~process context switching rate~~

The memory sizes (in bytes) for both primary and secondary memory gives memory capability. The memory transfer rates are needed as secondary measures,

especially to compute memory interference when multiple processors are used. This measure also permits system performance to be computed by subtracting the secondary memory transfers and external interface transfers. For file systems which require multiple accesses to secondary memory for single items, the file access rate capability is needed in order to compute performance. Similarly, for multiprogrammed systems which use secondary memory to hold programs, the access rate is needed. ~~note the correla~~

Communications capability with humans, other computers, and other electronic encoded processes are equally important structure and performance attributes. Each channel (e.g., a typewriter) has a certain data rate and direction (full duplex for simultaneous two way communication). Collectively, the data rates and the number of channels connected to each of the 3 different environments (people, computers, other electronically encoded processes) signify quite different styles of computing capability, structure and ultimately use. For example, having no communications connection to other computers implies a standalone system. Having only interconnection to mechanical processes via electronically encoded links implies a real time structure. Similarly only human intercommunication with multiple terminals denotes a timesharing or transaction-processing orientation.

Figure PLOT uses a Kiviat graph to display the above six main resource dimensions of processing, primary and secondary memory capacity, and the three communication channels in a single 6-d graph, with 3 additional dimensions. Each dimension is logarithmic over a factor of 1,000,000 with the value 0, denoting the absence of an attribute (e.g., there is not communication with

remove
?

external systems beyond human and standard communication). Secondary measures and unit quantities are denoted by separate numbers by each dimension. Unit quantities can either be multipliers, x (denoting the measure is for 1 unit such as a disk) or divisors, /, (denoting the measure is for all the combined units of the system). A number-sign, #, is equivalent to a multiplier, x, denoting the graph value is for just one unit. Additional attribute: values are plotted parallel to or as marks on a given dimensional scale. Occasionally dimensions are further specified (e.g., audio, video). Arrows denote directionality of information flow. Note that if the Pc speed is "balanced" with Mp size according to Amdahl's constant, then the value of the two should be about the same. (Here Pc is accessing 300,000 bytes/sec. corresponding to say 100,000 instructions/sec., with Mp of 100,000 bytes). The graph conventions include subtleties of showing fixed points (i.e., ROM or hardwired), and averages, range and overhead due to other resources.

The arrangement of the six dimensions allows easy recognition of a structure in terms of the relative mix of the resource/performance attributes. Figure BPMSR gives a diagram of a computer system in the same order as the graph's dimensions.

Figure EXAMPLES shows how the 6-d plot can be used to represent and differentiate various computing structures in which we're interested. The first two structures are keyboard i/o, i.e., a single information transducer we know as the typewriter which has half-duplex i/o at 10 characters (or bytes) per second. A 10 char./sec. teletype is formed by adding a line interface.

The simple, early fixed function hand-held calculator, e.g., the HP35, which has a fixed processing/memory structure with about 4 x 10 digits (or 20 bytes to be more precise, of primary memory and store, limited keyboard input and 10 light LED output at about 10 char/sec. The internal fixed program is stored in about 2,000 ROM bytes--hence there is a single, fixed point; and the operation-rate of the unit is fixed at about 100 accesses/sec. of the HP35's powerful data-types. The HP65 programmable calculator is shown next with various fixed functions being replaced by programs, and Mp and Ms are each 500 bytes. The functions in ROM, though still present are not apparent to the user, hence are removed.

The second line gives graphs of various terminal structures beginning with a fixed function operating at 10,000 accesses/sec. (or 100 usec) with about 1,000 bytes of local memory and 2400 bits/sec. or 300 bytes/sec. access to a computer. The unit can be made programmable at 20,000 accesses/sec. by providing processing on a 4,000 byte primary memory. Mass storage, here a floppy disk, is also added in the second case--which also serves as a communication link. Communication to the external world is at 2,400 baud, or 300 bytes/sec. Output to the screen is at 2,400 bytes/sec. or 19,200 bits/sec. with input at 10 char/sec.

The next two systems are remote job entry stations, the first is fixed function and the second programmable. There are two i/o channels, one of 2,400 baud (i.e., 300 8-bit bytes/sec.) for the card reader and 4,800 baud (or 300 lines/min. = 5 lines/sec. at 120 bytes/line = 600 bytes/sec.) for the line printer connected via a 4,800 baud full duplex link. The second RJE terminal

also includes a Pc at 50,000 accesses/sec. and an Mp of 16Kbytes. A tape unit of 50Kbytes/sec. which holds 300 Mbytes.

The next system is a programmable, store and forward system with 16 Kbytes, with a Pc which has an access rate of 100,000, with a context switching time of 1 millisecond. There are 32 lines of 10 to 150 bytes/sec. The four communication links to other computers operate at 600 or 1,200 bytes/sec. (or 4,800 or 9,600 baud). The next system is a fixed function, remote full duplex analog multiplexor with 16 channels operating at 16 x 100 bytes/sec. and multiplexed into a 1,200 byte/sec. (9,600 baud) line--hence the line limits the maximum sampling rate.

The next system is a programmable, remote, standalone process control system. Note the absence of any lines to communicate with other machines. A secondary memory system of 10 million bytes is used for communication with other computers. Both gross and net Pc (2,000 accesses/sec.) (2,000 bytes) resources are given. Net capabilities are after the other resources are managed. One-hundred transducers are sampled each 10 milliseconds with 3 transducers connected to humans at a data-rate of 30 bytes/sec.

The last series of systems are, general purpose, multiprogrammed computers. The first is a batch system with card and line printer. The next is an 11/70 with 100 real time inputs, 60 terminals, and 2 connections to other computers. The KL10 is a large, multi-user (100) timesharing system. Finally the largest computer, the CRAY-1 is given, showing the dependence on external computers for Ms, and terminals.

ISP (Architecture) Parameters

Whereas the hardware structure and operation rates mainly determines performance, the architecture does ^{contribute} ~~have a minor effect~~ as seen in the previous example. Within a given machine class (say minis), we believe architecture has ^{less of an} a minor effect on performance provided the data-types are embedded. ^{rather less effect} A simple, ^{Analys} yet effective single metric is the address size. The values for the data-types dimension is given in order of increasing complexity in Table computer-space. However, it is difficult to order the dimensions, except by complexity, because the ~~issue relevant~~ to performance is ^{determined by} whether a given problem requires the embedded data-type.

One can compare architecture relatively precisely (eliminating the effect of particular hardware implementations) by comparing a count of the number of bits which are statically required to encode the algorithm (s-measure) and the number of bits that are dynamically flow between the Pc and ^{(M)measure} Mp. A third measure gave the activity of the internal register processor (R-measure).

U.S. Defense Department's Computer Family Architecture

In the (CFA) study [~~Barbacci et al, 1977; Wald and Salisbury, 1977~~] which lead to the selection of the PDP-11 ^{as the standard} architecture, benchmarking was used to compare several architectures.

Barbacci et al, 1977; Bun et al, 1977; Fuller et al 1977a; Fuller et al 1977b

The benchmarks (see Table 3; from ^{Fuller} [~~Barbacci~~ et al, 1977]), oriented to real time use were each programmed by programmers using assembly language. The resultant programs were run on a simulator ^(instrumented to provide the S, M, and R measures) that interpreted the formal ISPS descriptions of the machines. The ISPS interpreter was instrumented to give

the above measures.

The CFA project also developed a single architectural measure based on a weighted average of various ISP parameters. The weightings were determined by the CFA user community and each parameter was evaluated in comparison with several competitive architectures. The parameters and their weights are given in Table 1 [from ^{Fuller}~~Barbaacci~~ et al, 1977].

Table 1 - Absolute Criteria for CFA Evaluation

1. Virtual Memory Support. - The architecture must support a virtual to physical translation mechanism.
2. Protection. - The architecture must have the capability to add new, experimental (i.e., not fully debugged) programs that may include I/O without endangering reliable operation of existing programs.
3. Floating Point Support. - The architecture must explicitly support one or more floating point data types with at least one of the formats yielding more than 10 decimal digits of significance in the mantissa.
4. Interrupts and Traps. - It must be possible to write a trap handler that is capable of executing a procedure to respond to any trap condition and then resume operation of the program. The architecture must be defined such that it is capable of resuming execution following any interrupt.

Single spaced

5. Subsetability. - At least the following components of an architecture must be able to be factored out of the full architecture:

Virtual-to-Physical Address Translation Mechanism

Floating Point Instructions and Registers (if separate from general purpose registers)

Decimal Instructions Set (if present in full architecture)

Protection Mechanism

6. Multiprocessor Support. - The architecture must allow for multiprocessor configurations. Specifically, it must support some form of "test-and-set" instruction to allow the implementation of synchronization functions such as P and V.
7. Controllability of I/O. - A processor must be able to exercise control over any I/O Processor and/or I/O Controller.
8. Extendability. - The architecture must have some method for adding instructions to the architecture consistent with existing formats. There must be at least one undefined code point in the existing opcode space of the instruction formats.
9. Read Only Code. - The architecture must allow programs to be kept in a

single spaced

read-only section of primary memory.

single space

Quantitative Criteria for CFA Evaluation

Weight (%)

1. Virtual Address Space

- (a) V_1 : The size of the virtual address space in bits. 4.3
- (b) V_2 : Number of addressable units in the virtual address space. 5.3

2. Physical Address Space

- (a) P_1 : The size of physical address space in bits. 6.1
- (b) P_2 : The number of addressable units in the physical address space. 5.1

3. Fraction of Instruction Space Unassigned 6.0

4. Size of Central Processor State

- (a) C_s : The number of bits in the processor state of the full *architecture* 4.9
- (b) C_s : The number of bits in the processor state of the minimum subset of the architecture (i.e., without Floating Point, Decimal, Protection, or Address Translation Registers). 3.7

(c) $C_m 1$: The number of bits that must be transferred between the processor and primary memory to first save the processor state of the full architecture upon interruption and then restore the processor state prior to resumption. 6.0

(d) $C_m 2$: The measure analogous to $C_m 1$ for the minimum subset of the architecture. 4.5

5. Virtualizability

K: is unity if the architecture is virtualizable as defined in [Popek and Goldberg, 1974] otherwise K is zero. 5.6

6. Usage Base

(a) B_1 : Number of computers delivered as of the latest date for which data exists prior to 1 June 1976. 3.1

(b) B_2 : Total dollar value of the installed computer base as of the latest date for which data exists prior to 1 June 1976. 2.5

7. I/O Initiation

I: The minimum number of bits which must be transferred between main memory and any processor (central, or I/O) in order to output one 8-bit to a standard peripheral device. 12.4

8. Direct Instruction Addressability

D: The maximum number of bits of primary memory which one instruction can directly address given a single base register which may be used but not modified. 10.2

single spaced.

9. Maximum Interrupt Latency

Let L be the maximum number of bits which may need to be transferred between memory and any processor (CP, IOC, etc.) between the time an interrupt is requested and the time that the computer starts processing that interrupt (given that interrupts are enabled). 9.2

10. Subroutine Linkage

J_1 : The number of bits which must be transferred between the processor and memory to save the user state, transfer to the called routine, restore the user state, and return to the calling routine, for the full architecture. No parameters are passed. 6.3

J_2 : The analogous measure to S_1 above for the minimum architecture (e.g., without Floating Point registers). 4.5

Insert (K)

Actual (i.e., Compound PMS/ISP) Performance Measure

(K)

The measures are defined so that computer architectures maximize some and minimize others. The measures that an architecture should maximize are $V_1, V_2, P_1, P_2, U, K, B_1, B_2$, and D , while the measures that should be kept to a minimum are $CS_1, CS_2, CM_1, CM_2, I, L, J_1$, and J_2 .

DEC

Quantitative Criteria Data for ~~Candidate~~ Architectures:

	PDP-8	PDP-11	PDP-10	VAX-11
V1	16.580	20.000	23.170	35.000
V2	15.580	19.000	23.170	35.000
P1	18.580	25.000	27.170	33.000
P2	18.580	24.000	27.170	33.000
U	0.090	0.043	0.030	0.800
CS1	78.000	1168.000	756.000	786.000
CS2	38.000	144.000	684.000	632.000
CM1	564.000	1136.000	144.000	1464.000
CM2	264.000	496.000	144.000	1336.000
K	1.000	1.000	0.000	0.000
B1	0.000	14700.000	0.000	0.000
B2	0.000	311.000	0.000	0.000
I	12.000	16.000	36.000	64.000
D	11.580	19.000	23.170	35.000
L	108.000	112.000	216.000	256.000
J1	1272.000	1424.000	1476.000	1152.000
J2	864.000	400.000	1476.000	1152.000

composite measure, a maximal measure, the inverses of those measures to be minimized were used.

Lloyd Dickman, of DEC, has computer enumerated the following for the four DEC computers.

Doing four pairwise comparisons he obtained the following.

Comparing the PDP-8.

~~The~~ PDP-11 1.16
PDP-8 .85

PDP-10 1.01
PDP-8 1.00

PDP-8 1.04
VAX-11 .96

Comparing the PDP-11.

PDP-11 1.16
PDP-8 .85

PDP-10 1.05
PDP-11 .95

PDP-11 1.01
VAX-11 1.00

Comparing the PDP-10.

PDP-10 1.01
PDP-8 1.00

PDP-10 1.05
PDP-11 .95

VAX-11 1.17
PDP-10 .84

Comparing VAX-11.

PDP-8 1.04
VAX-11 .96

VAX-11 1.00
PDP-11 1.01

VAX-11 1.17
PDP-10 .84

Taking the four machines as one set he obtained the following.

VAX-11 1.23
PDP-8 1.09
PDP-11 1.03
PDP-10 .66

In order to measure the performance of a specific computer (e.g., an 11/55), it is necessary to know the ISP, the hardware performance and the frequency of use for the various instructions. That is the execution time, T , is the dot product of the fractional utilization of each instruction U_i times the T_i time to execute each instruction, T_i .

There are three ways to estimate the instruction utilization, U and hence obtain T ; each providing increasingly better answers. The first, simply defines either a typical or average instruction. The second uses "standard" benchmarks to characterize a machine's performance precisely. In this way machines can be compared and there is an absolute measure. Finally, since the actual use has not been characterized in terms of the standard benchmark (and may even not be easily characterized in terms of it) a specific test (~~i.e.,~~ unique benchmark) may be necessary. This later characterization is quite possibly needed for real time and transaction processing where computer selection and installation is predicated on exactly doing the job.

Typical instructions

The simplest, single parameter of performance is the instruction time for some simple operation (e.g., add). These were used in the first few ^{two computer} generations especially since high level languages were less used. Such a metric is an approximation to the average instruction time and assumes all machines have about the same ISP and hence there is little difference among instructions, or that a specific data-type will be used more heavily than another, or that a typical add time will be given (e.g., the operand is in a random location in

primary memory call versus being cached or in a fast register).

Although it is possible to take the average instruction time by executing one of every possible instructions, since the instruction use depends so much on the data they interpret, this average is relatively meaningless. A better measure is to keep statistics about the use of all programs and to give the average instruction time based on use on all programs. Again, such a measure while useful for comparing two machines implementations of models of the same architecture, is also relatively useless, when it comes to particular specific useage.

Many years ago, there were attempts to make better characterizations by weighting the instructions use (i.e., forming a typical U) as to what they did, (e.g., floating point versus indexing and character handling) to give a better performance measure. We found for such instructions mixes that began to better approximate performance. These mixes, from Bell and Newell (1971) are given in Table Mix.

Table Mix Instruction-mix weights for evaluating computer power

	Arbuckle[1966]	Givson ¹	Knight(scientific)	Knight(commercial)
Fixed +/-	...	6	10(25) ²	25(45) ²
X	...	3	6	1
Divide	...	1	2	
Floating +/-	9.5		10	

Floating X	5.6			
Floating divide	2.0			
Load/store	28.5	25(move)		
Indexing	22.5			
Conditional branch	13.2	20		
Compare	...	24		
Branch on character	...	10		
Edit	...	4		
I/O initiate	...	7		
Other	18.7	...	72	74

¹Published reference unknown.

²Extra weight for either indirect addressing or index registers.

The best known, Gibson mix, is still used even today. It has a decidedly commercial flavor, and quite possibly reflects the proportion of machines executing commercial mixes with character operations as opposed to scientific, switching and control where proportional more integer and floating-point data types are used. Such mixes are still better approximations than a single instruction average, because use enters in. ^{note} We ~~must warn~~ that if the data-type operation is not present in the machine, the programmed subroutine time must be given -- typically a factor of 10-20 greater than for built-in operations.

Standard Benchmarks

The best estimate of real use comes from carefully designed "standard" benchmarks. *because it is understood and because many other machines use it.* Several organizations, particularly those who purchase or use many machines extensively have one or more programs that they believe characterize their own work load. Whether a standard benchmark can be of value in characterizing performance depends on the degree it is typical of the actual computers use. A further advantage of benchmarks is that they are the language that the computer is to be used, and hence, reflect the application and also characterize the language machine architecture. To illustrate the variability in the scientific FORTRAN benchmark metrics, performance of a number of machines, VAX-11/780 with floating point accelerator option, is compared with the 11/70 and with the 2050 Model B for 17 benchmarks. Two scientific benchmarks of the National Physical Laboratory in the UK [~~Witchman~~^{197X}??] are singled out as being the most useful because of the extensive effort (e.g., frequencies of the trigonometric functions, subroutine calls, and I/O were considered) and considerations into designing them as typical. Although these characterize scientific mix with FORTRAN, they can be used to compare various languages.

There are similar benchmarks for commercial processing which generally use the COBOL language.

Exact use characterization

In the event a machine has to be fully characterized before installation, there

is no alternative to running the exact problem which will be run on the final system. This is the most expensive alternative to characterize performance and should be avoided because of the dynamic nature of use. Showing that an application will yield a given performance on a particular machine is a weak guarantee about performance if any part of the problem changes.

Popek, G. J. and Goldberg R. P., Formal Requirements for Virtualizable Third Generation Architectures, Communications of the ACM, vol. 17, no. 7, July 1974, 412-421.

Wald, B. and Salisbury, A., Editors, "The Computer Family Architecture Project: Service Perspectives and Overview", Special Issue of Computer, vol. 10, no. 10, Oct. 1977, 9-43.

Barbacci, M. R., Burr, W. E., Fuller, S. H. and Siewiorek, W. E., Editors Evaluation of Alternative Computer Architectures, Dept. of Computer Science, Carnegie-mellon University, Pittsburgh, Pa. Feb. 1977.

Knight, K. E. Changes in Computer Performance Datamation, vol. 12, no. 9, pp. 40-54, Sept. 1966.

Arbuckle, R. A., Computer Analysis and Throughput Evaluation Computers and Automation p. 13, Jan 1966.

Wichman, ?

Turn, Rein, Computers in the 1980s, Columbia University Press, N.Y., 1974.

Sharpe, W. F., The Economics of Computers, Columbia University press, N.Y., 1969.

Phister, M. ~~1976~~ *Data Processing Technology and Economics*
Santa Monica Publishing Co., Santa Monica, 1976.

NCC

Asimow, Morris, Introduction to Design, Prentice Hall, 1964.²

Simon, Herbert A., The Sciences of the Artificial M.I.T. Press, 1969.

~~Strecker, W.D.~~

~~1970~~

Fuller, S.H., Shaman, P., and Lamb, D. Evaluation of Computer Architectures via Test Programs, Proc AFIPS NCC 1977, pp. 147-160.

Burr, W.E., Coleman, A.H., and Smith, W.R. Overview of the Military Computer Family Architecture Selection, Proc AFIPS NCC 1977, pp. 131-138

Fuller, S.H., Stone, H.S., and Burr, W.E. Initial Selection and Screening of the CFA candidate computer architectures, Proc AFIPS NCC 1977, pp. 139-146

Barbacci, M.R., Siewiorek, D.P., Gordon, R., Howbrigg, R., and ~~Zuckerman~~ Zuckerman, S. An architectural Research Facility - ISP descriptions, simulation, data collection. Proc AFIPS NCC 1977, pp 161-174.

Mudge, J.C. Human factors in the Design of a Computer-assisted instruction system. Ph.D. Dissertation. Univ. of North Carolina at Chapel Hill, 1973.

Chapter 1 figures

Figure

Fig 1 ~~organization~~ Computer Production - Consumption Process

2 structural levels from Bell and Newell

15 Onion skin

Iloop the basic interpretive loop

formalproc a processing system

hier hierarchy of interpreters

pipes pipes through interpreters

11 levels vs time

12 Noyce's levels

13 Levels of integration

14 Network, not tree

x 3 design styles

y evolutions from base design B

5 Machine price & perf vs time (machine planes)

6 Price vs time for each machine class

~~8~~ 8 Migration of constant performance machine use among P-C pairs

9. Groschein

datatype * histogram of datatype by application

Asimov Philosophy of design

Platten 1 product development schedule I Fig 4.21.3 (p.207)

Platten 2 " II Fig 4.21.4 (p.207)

Platten 3 engineers assigned vs elapsed time Fig 4.21.7 (p.209)

Maxims Vonada's engineering maxims

~~Table 3 12 test~~

Table 1 the computer space

Appendix figures

• BPMS PMS diagram of a basic computer

Table 3 12 test programs used in CFA.

bottom of p. 58

Performance

In this section we will discuss computer hardware performance, neglecting the operating system and language use because the later so drastically affect performance as to make the discussion meaningless. Performance parameters are a combination of architecture (the ISP), hardware implementation and resources

(the PMS structure) being acted on by some program. For example, although a certain machine (hardware) can deliver 500,000 instructions/per second to a program, we must also define what the instructions are (the ISP) and how a particular program makes use of the instructions.

Most of the time we use overly simplistic hardware measures (e.g., instruction time) to characterize machine performance. The ultimate performance parameters have to be based on actual use (i.e., workload) parameters, otherwise there is no way to correlate the primitive measures to real performance. Benchmarks (including standardized benchmarks such as Whetstones for the algorithmic scientific languages) and synthetic or real workload provide the only real test by which performance can be compared. The purpose of this section is to discuss hardware performance generally, and then to sort out its various components.

When we measure performance, there is a tacit assumption that sufficient software exists to exploit a hardware structure, and that the transformation from this basic, lowest level hardware machine (i.e., the macro machine) to the user machine (as provided by a language such as COBOL or FORTRAN) is relatively constant across various architectures. As each level is crossed, a transformation takes place requiring computational work. The form of the work with compiled languages is direct execution via the processor and run time support program and with interpreted languages the processor executes an interpretation program which indirectly interprets the final program.

At the lowest level, the internal micro machine provides the architectural

facade, the ISP, for the macro machine and operates at roughly 10 times the later's speed. Thus a machine executing 1 million instructions per second may have an effective microcycle time of 100 nanoseconds, permitting 10 million micro instructions/sec. At the next level, a conventional macro machine (ISP) executing 1 million instructions per second, is capable of perhaps 0.1 to 0.25 million higher level FORTRAN language statements (instructions) per second depending on the mix of built-in functions and external functions called.

It is difficult to use the simplistic constant ratio measures across each level of interpretation when comparing machines of differing classes (e.g., micro to super) because there is not a consistency of data-types (e.g., micros currently have no built-in real arithmetic, whereas minis do). However, for machines within a class (e.g., mini) where the data-types are implied by the class name, it is probably all right, since the two machines most likely have about the same data-types. Hence a count of the number of data-types reflecting the built-in operations is one of the more significant architectural performance indicators, whether it be for a micro machine, macro machine or a language machine.

PMS (I.E.. Resources) Performance Parameters

The PMS structure, with the corresponding attributes determining performance (e.g., memory cycle time, processor execution rate) provides the basis for understanding machines and comparing them with each other. Figure BPMS gives a PMS diagram of a basic computer, with the parameters that will be used to characterize performance. The diagram shows the structure and key performance

parameters of each component as they provide a total computing environment. Alternatively, one might use a more descriptive, or tabular form, but the goal here is to provide a structural/performance basis for parameterization, comparison, and specifying the finite resources of the computer.

It's imperative to consider the resource constraints, and the effect of their interaction as each next layer of a machine is designed. For example, a certain line printer requires buffer space (Mp. size) and processing time (Pc. speed) which is then unavailable at the next machine level (e.g., FORTRAN). Although a clear layering is needed to characterize each level, we will not carry on this layering beyond the hardware.

Bell and Newell (1971) argued that a machine (at any level) can be described with any number of parameters, and carried out the exercise for up to 5 parameters:

Number of parameters

allowed:	1	2	3	4	5
1	Pc(i.rate)	-	-	Pc(op.rate)	
2		Mp(size)	-		-
3			Ms(size)	-	-
4				Pc(i.width)	-
5					No. of terminals

Information.rate between the processor and memory, is used as the processor speed indicator instead of the more conventional instructions per second. Compound indicators such as the product of processor speed times memory size to indicate basic computational performance used in the earlier view were not allowed.

The following example shows 3 different architectures with 2 implementations of a stack architecture -- one has the stack in the primary memory, Mp, and the other assumes the stack is in the processor, Pc, using fast registers. The hardware implementations are held roughly constant (the Pc-Mp data rate) and the architecture is varied in order to compare the effect on performance. Note the difference in the various measures in what should fundamentally be about the same performance for a given problem.

The benchmark program is the simple expression, $A := B + C$

	<u>Stack (top in Mp)</u>	<u>Stack (top in Pc)</u>	<u>1-address or general reg.</u>	<u>3-address</u>
Program	push B push C add pop A	push B push C add pop A	load B add C store A	add B,C,A
No. of Instns.	4	4	3	1
Accesses	$4op'+3a+6d$	$4op'+3a+3d$	$3op+3a+3d$	$1op''+3a+3d$
Program size (bits*)	64	64	72	60
Bits accessed	$16+48+192$ =266	$16+48+96$ =160	$24+48+96$ =168	$12+48+96$ =156
Time to execute** (microseconds)	$0.5+1.5+6$ =8	$0.5+1.5+3$ =5	$.75+1.5+3$ =5.25	$.37+1.5+3$ =4.87
Statement exec. rate(actual performance)	$1/8 = .125m$	$1/5 = .2m$	$1/5.25 = .19m$	$1/4.87 = .21m$
Oper.rate	$2/8 = .25m$	$2/5 = .4m$	$2/5.25 = .38m$	$2/4.87 = .42m$
Inst. rate	$4/8 = .5m$	$4/5 = .8m$	$3/5.25 = .57m$	$1/4.87 = .21m$
Pc(i.rate)/ word length	$32m = 1m$	$32m = 1m$	$32m = 1m$	$32m = 1m$

*assumes: address/a = 16b; data/d = 32b; op = 8b; op' = 4b; op'' = 12b

**assumes a memory limited processor which can access 32b/microsecond

The statement execution rate (the actual performance) is the highest for the 3-address machine, reflecting the highest performance whereas the conventional measure (instns/sec) shows it to have the lowest performance (by a factor of 4) over the fastest machine. A more subtle measure, operation-rate, is correlated

with the true benchmark statement execution rate. It should be noted, (ignoring the first machine, a stack machine with stack in Mp) that the information-rate is a good performance indicator - versus the conventional, but poor, instruction-rate measure. For more unconventional machines, instructions/sec. tends to become a significantly poorer measure. The various vector/array machines (e.g., ILLIAC IV, CDC STAR, CRAY-1) have single instructions to operate on at least 64 operands per instruction, hence instructions/second would be a poor measure.

Similarly hand held calculators have single instructions such as Sin, Polar to Cartesian co-ordinate conversion, and using anything but a final benchmark problem would be unfair. Accesses/sec. will be used as a Pc performance measure.

The multiprocessor case

For multi-processors the number of processors x the memory accesses/sec. roughly gives the total Pc.rate. Pc.rate can be computed more precisely by using the number of primary memory modules, Mp, and their data-rate as can be seen in Chapter 00 on the C.mmp computer. For a system where the memory access time, and the memory rewrite time equal the time for a Pc to operate on a word, the performance (Strecker, 1970) is roughly:

$$\text{Pc.speed (in accesses/sec.)} = (m/t.\text{access}) \times (1 - (1 - 1/m)^p)$$

where m = # of Mp modules, and p = # of Pc's

Note that when p = m = large, the performance reaches an assymtote:

$$= m/tc \times (1/e)$$

In the case of multiprogramming systems (e.g., real time, transaction, and timesharing), the time to switch from job to job is critical. This measure, process context switching rate is one of the main attributes since most computer systems operate with some form of multiprogramming.

The memory sizes (in bytes) for both primary and secondary memory gives memory capability. The memory transfer rates are needed as secondary measures, especially to compute memory interference when multiple processors are used. This measure also permits system performance to be computed by subtracting the secondary memory transfers and external interface transfers. For file systems which require multiple accesses to secondary memory for single items, the file access rate capability is needed in order to compute performance. Similarly, for multiprogrammed systems which use secondary memory to hold programs, the access rate is needed.

Communications capability with humans, other computers, and other electronic encoded processes are equally important structure and performance attributes. Each channel (e.g., a typewriter) has a certain data rate and direction (full

duplex for simultaneous two way communication). Collectively, the data rates and the number of channels connected to each of the 3 different environments (people, computers, other electronically encoded processes) signify quite different styles of computing capability, structure and ultimately use. For example, having no communications connection to other computers implies a standalone system. Having only interconnection to mechanical processes via electronically encoded links implies a real time structure. Similarly only human intercommunication with multiple terminals denotes a timesharing or transaction-processing orientation.

Figure PLOT uses a Kiviat graph to display the above six main resource dimensions of processing, primary and secondary memory capacity, and the three communication channels in a single 6-d graph, with 3 additional dimensions. Each dimension is logarithmic over a factor of 1,000,000 with the value 0, denoting the absence of an attribute (e.g., there is not communication with external systems beyond human and standard communication). Secondary measures and unit quantities are denoted by separate numbers by each dimension. Unit quantities can either be multipliers, x (denoting the measure is for 1 unit such as a disk) or divisors, /, (denoting the measure is for all the combined units of the system). A number-sign, #, is equivalent to a multiplier, x, denoting the graph value is for just one unit. Additional attribute: values are plotted parallel to or as marks on a given dimensional scale. Occasionally dimensions are further specified (e.g., audio, video). Arrows denote directionality of information flow. Note that if the Pc speed is "balanced" with Mp size according to Amdahl's constant, then the value of the two should be about the same. (Here Pc is accessing 300,000 bytes/sec. corresponding to say 100,000 instructions/sec., with Mp of 100,000 bytes). The graph conventions include subtleties of showing fixed points (i.e., ROM or hardwired), and averages, range and overhead due to other resources.

The arrangement of the six dimensions allows easy recognition of a structure in terms of the relative mix of the resource/performance attributes. Figure BPMSR gives a diagram of a computer system in the same order as the graph's dimensions.

Figure EXAMPLES shows how the 6-d plot can be used to represent and differentiate various computing structures in which we're interested. The first two structures are keyboard i/o, i.e., a single information transducer we know as the typewriter which has half-duplex i/o at 10 characters (or bytes) per second. A 10 char./sec. teletype is formed by adding a line interface.

The simple, early fixed function hand-held calculator, e.g., the HP35, which has a fixed processing/memory structure with about 4×10 digits (or 20 bytes) to be more precise, of primary memory and store, limited keyboard input and 10 light LED output at about 10 char/sec. The internal fixed program is stored in about 2,000 ROM bytes--hence there is a single, fixed point; and the operation-rate of the unit is fixed at about 100 accesses/sec. of the HP35's powerful data-types. The HP65 programmable calculator is shown next with various fixed functions being replaced by programs, and Mp and Ms are each 500 bytes. The functions in ROM, though still present are not apparent to the user, hence are removed.

The second line gives graphs of various terminal structures beginning with a fixed function operating at 10,000 accesses/sec. (or 100 usec) with about 1,000 bytes of local memory and 2400 bits/sec. or 300 bytes/sec. access to a computer. The unit can be made programmable at 20,000 accesses/sec. by providing processing on a 4,000 byte primary memory. Mass storage, here a floppy disk, is also added in the second case--which also serves as a communication link. Communication to the external world is at 2,400 baud, or 300 bytes/sec. Output to the screen is at 2,400 bytes/sec. or 19,200 bits/sec. with input at 10 char/sec.

The next two systems are remote job entry stations, the first is fixed function and the second programmable. There are two i/o channels, one of 2,400 baud (i.e., 300 8-bit bytes/sec.) for the card reader and 4,800 baud (or 300 lines/min. = 5 lines/sec. at 120 bytes/line = 600 bytes/sec.) for the line printer connected via a 4,800 baud full duplex link. The second RJE terminal also includes a Pc at 50,000 accesses/sec. and an Mp of 16Kbytes. A tape unit of 50Kbytes/sec. which holds 300 Mbytes.

The next system is a programmable, store and forward system with 16 Kbytes, with a Pc which has an access rate of 100,000, with a context switching time of 1 millisecond. There are 32 lines of 10 to 150 bytes/sec. The four communication links to other computers operate at 600 or 1,200 bytes/sec. (or 4,800 or 9,600 baud). The next system is a fixed function, remote full duplex analog multiplexor with 16 channels operating at 16 x 100 bytes/sec. and multiplexed into a 1,200 byte/sec. (9,600 baud) line--hence the line limits the maximum sampling rate.

The next system is a programmable, remote, standalone process control system. Note the absence of any lines to communicate with other machines. A secondary memory system of 10 million bytes is used for communication with other computers. Both gross and net Pc (2,000 accesses/sec.) (2,000 bytes) resources are given. Net capabilities are after the other resources are managed. One-hundred transducers are sampled each 10 milliseconds with 3 transducers connected to humans at a data-rate of 30 bytes/sec.

The last series of systems are, general purpose, multiprogrammed computers. The first is a batch system with card and line printer. The next is an 11/70 with 100 real time inputs, 60 terminals, and 2 connections to other computers. The KL10 is a large, multi-user (100) timesharing system. Finally the largest computer, the CRAY-1 is given, showing the dependence on external computers for Ms, and terminals.

ISP (Architecture) Parameters

Whereas the hardware structure and operation rates mainly determines performance, the architecture does have a minor effect as seen in the previous example. Within a given machine clan (say minis), we believe architecture has a minor effect on performance provided the data-types are embedded. A simple, yet effective single metric is the address size. The values for the data-types dimension is given in order of increasing complexity in Table computer-space. However, it is difficult to order the dimensions, except by complexity, because

the issue relevant to performance is whether a given problem requires the embedded data-type.

One can compare architecture relatively precisely (eliminating the effect of particular hardware implementations) by comparing a count of the number of bits which are statically required to encode the algorithm (s-measure) and the number of bits that are dynamically flow between the Pc and (Mpmeasure) Mp. A third measure gave the activity of the internal register processor (R-measure).

In the CFA study [Barbacci et al, 1977; Wald and Salisbury, 1977] which lead to the selection of the PDP-11 architecture, benchmarking was used to compare several architectures.

The benchmarks (see Table 3; from [Barbacci et al, 1977]), oriented to real time use were each programmed by programmers using assembly language. The resultant programs were run on a simulator that interpreted the formal ISPS descriptions of the machines. The ISPS interpreter was instrumented to give the above measures.

The CFA project also developed a single architectural measure based on a weighted average of various ISP parameters. The weightings were determined by the CFA user community and each parameter was evaluated in comparison with several competitive architectures. The parameters and their weights are given in Table 1 [from Barbacci et al, 1977].

Table 1 - Absolute Criteria for CFA Evaluation

1. Virtual Memory Support. - The architecture must support a virtual to physical translation mechanism.
2. Protection. - The architecture must have the capability to add new, experimental (i.e., not fully debugged) programs that may include I/O without endangering reliable operation of existing programs.
3. Floating Point Support. - The architecture must explicitly support one or more floating point data types with at least one of the formats yielding more than 10 decimal digits of significance in the mantissa.
4. Interrupts and Traps. - It must be possible to write a trap handler that is capable of executing a procedure to respond to any trap condition and then resume operation of the program. The architecture must be defined such that it is capable of resuming execution following any interrupt.
5. Subsetability. - At least the following components of an architecture must be able to be factored out of the full architecture:

Virtual-to-Physical Address Translation Mechanism

Floating Point Instructions and Registers (if separate from general purpose registers)

Decimal Instructions Set (if present in full architecture)

Protection Mechanism

6. Multiprocessor Support. - The architecture must allow for multiprocessor configurations. Specifically, it must support some form of "test-and-set" instruction to allow the implementation of synchronization functions such as P and V.
7. Controllability of I/O. - A processor must be able to exercise control over any I/O Processor and/or I/O Controller.
8. Extendability. - The architecture must have some method for adding instructions to the architecture consistent with existing formats. There must be at least one undefined code point in the existing opcode space of the instruction formats.
9. Read Only Code. - The architecture must allow programs to be kept in a read-only section of primary memory.

Quantitative Criteria for CFA Evaluation

	<u>Weight (%)</u>
1. Virtual Address Space	
(a) V_1 : The size of the virtual address space in bits.	4.3
(b) V_2 : Number of addressable units in the virtual address space.	5.3
2. Physical Address Space	
(a) P_1 : The size of physical address space in bits.	6.1
(b) P_2 : The number of addressable units in the physical address space.	5.1
3. Fraction of Instruction Space Unassigned	6.0
4. Size of Central Processor State	
(a) $C_s 1$: The number of bits in the processor state of the full	4.9
(b) $C_s 2$: The number of bits in the processor state of the minimum subset of the architecture (i.e., without Floating Point, Decimal, Protection, or Address Translation Registers.	3.7
(c) $C_m 1$: The number of bits that must be transferred between the processor and primary memory to first save the processor state of the full architecture upon interruption and then restore the processor state prior to resumption.	6.0
(d) $C_m 2$: The measure analogous to $C_m 1$ for the minimum subset of	4.5

the architecture.

5. Virtualizability

K: is unity if the architecture is virtualizable as defined in [Popek and Goldberg, 1974] otherwise K is zero. 5.6

6. Usage Base

(a) B_1 : Number of computers delivered as of the latest date for which data exists prior to 1 June 1976. 3.1

(b) B_2 : Total dollar value of the installed computer base as of the latest date for which data exists prior to 1 June 1976. 2.5

7. I/O Initiation

I: The minimum number of bits which must be transferred between main memory and any processor (central, or I/O) in order to output one 8-bit to a standard peripheral device. 12.4

8. Direct Instruction Addressability

D: The maximum number of bits of primary memory which one instruction can directly address given a single base register which may be used but not modified. 10.2

9. Maximum Interrupt Latency

Let L be the maximum number of bits which may need to be transferred between memory and any processor (CP, IOC, etc.) between the time an interrupt is requested and the time that the computer starts processing that interrupt (given that interrupts are enabled). 9.2

10. Subroutine Linkage

J_1 : The number of bits which must be transferred between the processor and memory to save the user state, transfer to the called routine, restore the user state, and return to the calling routine, for the full architecture. No parameters are passed. 6.3

J_2 : The analogous measure to S_1 above for the minimum architecture (e.g., without Floating Point registers). 4.5

Actual (i.e., Compound PMS/ISP) Performance Measure

In order to measure the performance of a specific computer (e.g., an 11/55), it is necessary to know the ISP, the hardware performance and the frequency of use for the various instructions. That is the execution time, T, is the dot

product of the fractional utilization of each instruction U_i times the T_i time to execute each instruction, T_i .

There are three ways to estimate the instruction utilization, U and hence obtain T ; each providing increasingly better answers. The first, simply defines either a typical or average instruction. The second uses "standard" benchmarks to characterize a machine's performance precisely. In this way machines can be compared and there is an absolute measure. Finally, since the actual use has not been characterized in terms of the standard benchmark (and may even not be easily characterized in terms of it) a specific test (i.e., unique benchmark) may be necessary. This later characterization is quite possibly needed for real time and transaction processing where computer selection and installation is predicated on exactly doing the job.

Typical instructions

The simplest, single parameter of performance is the instruction time for some simple operation (e.g., add). These were used in the first few generations especially since high level languages were less used. Such a metric is an approximation to the average instruction time and assumes all machines have about the same ISP and hence there is little difference among instructions, or that a specific data-type will be used more heavily than another, or that a typical add time will be given (e.g., the operand is in a random location in primary memory call versus being cached or in a fast register).

Although it is possible to take the average instruction time by executing one of every possible instructions, since the instruction use depends so much on the data they interpret, this average is relatively meaningless. A better measure is to keep statistics about the use of all programs and to give the average instruction time based on use on all programs. Again, such a measure while useful for comparing two machines implementations of models of the same architecture, is also relatively useless, when it comes to particular specific useage.

Many years ago, there were attempts to make better characterizations by weighting the instructions use (i.e., forming a typical U) as to what they did, (e.g., floating point versus indexing and character handling) to give a better performance measure. We found for such instructions mixes that began to better approximate performance. These mixes, from Bell and Newell (1971) are given in Table Mix.

Table Mix Instruction-mix weights for evaluating computer power

	Arbuckle[1966]	Givson ¹	Knight(scientific)	Knight(commercial)
Fixed +/-	...	6	10(25) ²	25(45) ²
X	...	3	6	1
Divide	...	1	2	
Floating +/-	9.5		10	
Floating X	5.6			
Floating divide	2.0			

Load/store	28.5	25(move)	
Indexing	22.5		
Conditional branch	13.2	20	
Compare	...	24	
Branch on character	...	10	
Edit	...	4	
I/O initiate	...	7	
Other	18.7	...	72 74

¹Published reference unknown.

²Extra weight for either indirect addressing or index registers.

The best known Gibson mix is still used even today. It has a decidedly commercial flavor, and quite possibly reflects the proportion of machines executing commercial mixes with character operations as opposed to scientific, switching and control where proportional more integer and floating-point data types are used. Such mixes are still better approximations than a single instruction average, because use enters in. We must warn that if the data-type operation is not present in the machine, the programmed subroutine time must be given -- typically a factor of 10-20 greater than for built-in operations.

Standard Benchmarks

The best estimate of real use comes from carefully designed "standard" benchmarks. Several organizations, particularly those who purchase or use many machines extensively have one or more programs that they believe characterize their own work load. Whether a standard benchmark can be of value in characterizing performance depends on the degree it is typical of the actual computers use. A further advantage of benchmarks is that they are the language that the computer is to be used, and hence, reflect the application and also characterize the language machine architecture. To illustrate the variability in the scientific FORTRAN benchmark metrics, performance of a number of machines, VAX-11/780 with floating point accelerator option, is compared with the 11/70 and with the 2050 Model B for 17 benchmarks. Two scientific benchmarks of the National Physical Laboratory in the UK [Witchman??] are singled out as being the most useful because of the extensive effort (e.g., frequencies of the trigonometric functions, subroutine calls, and I/O were considered) and considerations into designing them as typical. Although these characterize scientific mix with FORTRAN, they can be used to compare various languages.

There are similar benchmarks for commercial processing which generally use the COBOL language.

Exact use characterization

In the event a machine has to be fully characterized before installation, there is no alternative to running the exact problem which will be run on the final

system. This is the most expensive alternative to characterize performance and should be avoided because of the dynamic nature of use. Showing that an application will yield a given performance on a particular machine is a weak guarantee about performance if any part of the problem changes.

Popek, G. J. and Goldberg R. P., Formal Requirements for Virtualizable Third Generation Architectures, Communications of the ACM, vol. 17, no. 7, July 1974, 412-421.

Wald, B. and Salisbury, A., Editors, "The Computer Family Architecture Project: Service Perspectives and Overview", Special Issue of Computer, vol. 10, no. 10, Oct. 1977, 9-43.

Barbacci, M. R., Burr, W. E., Fuller, S. H. and Siewiorek, W. E., Editors Evaluation of Alternative Computer Architectures, Dept. of Computer Science, Carnegie-mellon University, Pittsburgh, Pa. Feb. 1977.

Knight, K. E. Changes in Computer Performance Datamation, vol. 12, no. 9, pp. 40-54, Sept. 1966.

Arbuckle, R. A., Computer Analysis and Throughput Evaluation Computers and Automation p. 13, Jan 1966.

Wichtman, ?

Turn, Rein, Computers in the 1980s, Columbia University Press, N.Y., 1974.

Sharpe, W. F., The Economics of Computers, Columbia University press, N.Y., 1969.

Phister, M. 1976

Asimov, Morris, Introduction to Design, Prentice Hall, 1964.

Simon, Herbert A., The Sciences of the Artificial M.I.T. Press, 1969.