Chapter 38

A New Architecture for Mini-Computers: The DEC PDP-11[1]

G. Bell / R. Cady / H. McFarland / B. DeLagi / J. O'Laughlin / R. Noonan / W. Wulf

Introduction

The mini-computer[2] has a wide variety of uses: communications controller; instrument controller; large-system pre-processor; real-time data acquisition systems . . . ; desk calculator. Historically, Digital Equipment Corporation's PDP-8 Family, with 6,000 installations has been the archetype of these mini-computers.

In some applications current mini-computers have limitations. These limitations show up when the scope of their initial task is increased (e.g., using a higher level language, or processing more variables). Increasing the scope of the task generally requires the use of more comprehensive executives and system control programs, hence larger memories and more processing. This larger system tends to be at the limit of current mini-computer capability, thus the user receives diminishing returns with respect to memory, speed efficiency and program development time. This limitation is not surprising since the basic architectural concepts for current mini-computers were formed in the early 1960's. First, the design was constrained by cost, resulting in rather simple processor logic and register configurations. Second, application experience was not available. For example, the early constraints often created computing designs with what we now consider weaknesses:

1 Limited addressing capability, particularly of larger core sizes

2 Few registers, general registers, accumulators, index registers, base registers

3 No hardware stack facilities

4 Limited priority interrupt structures, and thus slow context switching among multiple programs (tasks)

5 No byte string handling

6 No read only memory facilities

7 Very elementary I/O processing

8 No larger model computer, once a user outgrows a particular model

9 High programming costs because users program in machine language.

In developing a new computer the architecture should at least solve the above problems. Fortunately, in the late 1960's integrated circuit semiconductor technology became available so that newer computers could be designed which solve these problems at low cost. Also, by 1970 application experience was available to influence the design. The new architecture should thus lower programming cost while maintaining the low hardware cost of mini-computers.

The DEC PDP-11, Model 20 is the first computer of a computer family designed to span a range of functions and performance. The Model 20 is specifically discussed, although design guidelines are presented for other members of the family. The Model 20 would nominally be classified as a third generation (integrated circuits), 16-bit word, 1 central processor with eight 16-bit general registers, using two's complement arithmetic and addressing up to 2^{16} eight bit bytes of primary memory (core). Though classified as a general register processor, the operand accessing mechanism allows it to perform equally well as a 0-(stack), 1-(general register) and 2-(memory-to-memory) address computer. The computer's components (processor, memories, controls, terminals) are connected via a single switch, called the Unibus.

[1] *AFIPS Proc. SJCC*, 1970, pp. 657–675.

[2] The PDP-11 design is predicated on being a member of one (or more) of the micro, midi, mini, . . . , maxi (computer name) markets. We will define these names as belonging to computers of the third generation (integrated circuit to medium scale integrated circuit technology), having a core memory with cycle time of .5 ~ 2 microseconds, a clock rate of 5 ~ 10 *M hz* . . . , a single processor with interrupts and usually applied to doing a particular task (e.g., controlling a memory or communications lines, pre-processing for a larger system, process control). The specialized names are defined as follows:

	Maximum addressable primary memory (words)	*Processor and memory cost (1970 kilodollars)*	*Word length (bits)*	*Processor state (words)*	*Data types*
Micro	8 K	~ 5	8 ~ 12	2	Integers, words, boolean vectors
Mini	32 K	5 ~ 10	12 ~ 16	2–4	Vectors (i.e., indexing)
Midi	65 ~ 128 K	10 ~ 20	16 ~ 24	4–16	Double length floating point (occasionally)

The machine is described using the PMS and ISP notation of Bell and Newell [1971] at different levels. The following descriptive sections correspond to the levels: external design constraints level; the PMS level—the way components are interconnected and allow information to flow; the program level or ISP (Instruction Set Processor)—the abstract machine which interprets programs; and finally, the logical design level. (We omit a discussion of the circuit level—the PDP-11 being constructed from TTL integrated circuits.)

Design Constraints

The principal design objective is yet to be tested; namely, do users like the machine? This will be tested both in the market place and by the features that are emulated in newer machines; it will indirectly be tested by the life span of the PDP-11 and any offspring.

Word Length

The most critical constraint, word length (defined by IBM) was chosen to be a multiple of 8 bits. The memory word length for the Model 20 is 16 bits, although there are 32- and 48-bit instructions and 8- and 16-bit data. Other members of the family might have up to 80 bit instructions with 8-, 16-, 32- and 48-bit data. The internal, and preferred external character set was chosen to be 8-bit ASCII.

Range and Performance

Performance and function range (extendability) were the main design constraints; in fact, they were the main reasons to build a new computer. DEC already has (4) computer families that span a range[1] but are incompatible. In addition to the range, the initial machine was constrained to fall within the small-computer product line, which means to have about the same performance as a PDP-8. The initial machine outperforms the PDP-5, LINC, and PDP-4 based families. Performance, of course, is both a function of the instruction set and the technology. Here, we're fundamentally only concerned with the instruction set performance because faster hardware will always increase performance for any family. Unlike the earlier DEC families, the PDP-11 had to be designed so that new models with significantly more performance can be added to the family.

A rather obvious goal is maximum performance for a given model. Designs were programmed using benchmarks, and the results compared with both DEC and potentially competitive machines. Although the selling price was constrained to lie in the $5,000 to $10,000 range, it was realized that the decreasing cost of logic would allow a more complex organization than earlier DEC computers. A design which could take advantage of medium- and eventually large-scale integration was an important consideration. First, it could make the computer perform well; and second, it would extend the computer family's life. For these reasons, a general registers organization was chosen.

Interrupt Response. Since the PDP-11 will be used for real time control applications, it is important that devices can communicate with one another quickly (i.e., the response time of a request should be short). A multiple priority level, nested interrupt mechanism was selected; additional priority levels are provided by the physical position of a device on the Unibus. Software polling is unnecessary because each device interrupt corresponds to a unique address.

Software

The total system including software is of course the main objective of the design. Two techniques were used to aid programmability: first benchmarks gave a continuous indication as to how well the machine interpreted programs; second, systems programmers continually evaluated the design. Their evaluation considered: what code the compiler would produce; how would the loader work; ease of program relocability; the use of a debugging program; how the compiler, assembler and editor would be coded—in effect, other benchmarks; how real time monitors would be written to use the various facilities and present a clean interface to the users; finally the ease of coding a program.

Modularity

Structural flexibility (sometimes called modularity) for a particular model was desired. A flexible and straightforward method for interconnecting components had to be used because of varying user needs (among user classes and over time). Users should have the ability to configure an optimum system based on cost, performance and reliability, both by interconnection and, when necessary, constructing new components. Since users build special hardware, a computer should be easily interfaced. As a by-product of modularity, computer components can be produced and stocked, rather than tailor-made on order. The physical structure is almost identical to the PMS structure discussed in the following section; thus, reasonably large building blocks are available to the user.

Microprogramming

A note on microprogramming is in order because of current interest in the "firmware" concept. We believe microprogramming, as we understand it [Wilkes, 1951], can be a worthwhile

[1]PDP-4, 7, 9, 15 family; PDP-5, 8, 8/S, 8/I, 8/L family; LINC, PDP-8/LINC, PDP-12 family; and PDP-6, 10 family. The initial PDP-1 did not achieve family status.

technique as it applies to processor design. For example, microprogramming can probably be used in larger computers when floating point data operators are needed. The IBM System/360 has made use of the technique for defining processors that interpret both the System/360 instruction set and earlier family instruction sets (e.g., 1401, 1620, 7090). In the PDP-11 the basic instruction set is quite straightforward and does not necessitate microprogrammed interpretation. The processor-memory connection is asynchronous and therefore memory of any speed can be connected. The instruction set encourages the user to write reentrant programs; thus, read-only memory can be used as part of primary memory to gain the permanency and performance normally attributed to microprogramming. In fact, the Model 10 computer which will not be further discussed has a 1024-word read only memory, and a 128-word read-write memory.

Understandability

Understandability was perhaps the most fundamental constraint (or goal) although it is now somewhat less important to have a machine that can be quickly understood by a novice computer user than it was a few years ago. DEC's early success has been predicated on selling to an intelligent but inexperienced user. Understandability, though hard to measure, is an important goal because all (potential) users must understand the computer. A straightforward design should simplify the systems programming task; in the case of a compiler, it should make translation (particularly code generation) easier.

PDP-11 Structure at the PMS Level[1]

Introduction

PDP-11 has the same organizational structure as nearly all present day computers (Fig. 1). The primitive PMS components are: the primary memory (Mp) which holds the programs while the central processor (Pc) interprets them; io controls (Kio) which manage data transfers between terminals (T) or secondary memories (Ms) to primary memory (Mp); the components outside the computer at periphery (X) either humans (H) or some external process (e.g., another computer); the processor console (T. console) by which humans communicate with the computer and observe its behavior and affect changes in its state; and a switch (S) with its control (K) which allows all the other components to communicate with one another. In the case of PDP-11, the central logical switch structure is implemented using a bus or chained switch (S) called the Unibus, as shown in Fig. 2. Each physical component has a

[1]A descriptive (block-diagram) level [Bell and Newell, 1971] to describe the relationship of the computer components: processors memories, switches, controls, links, terminals and data operators.

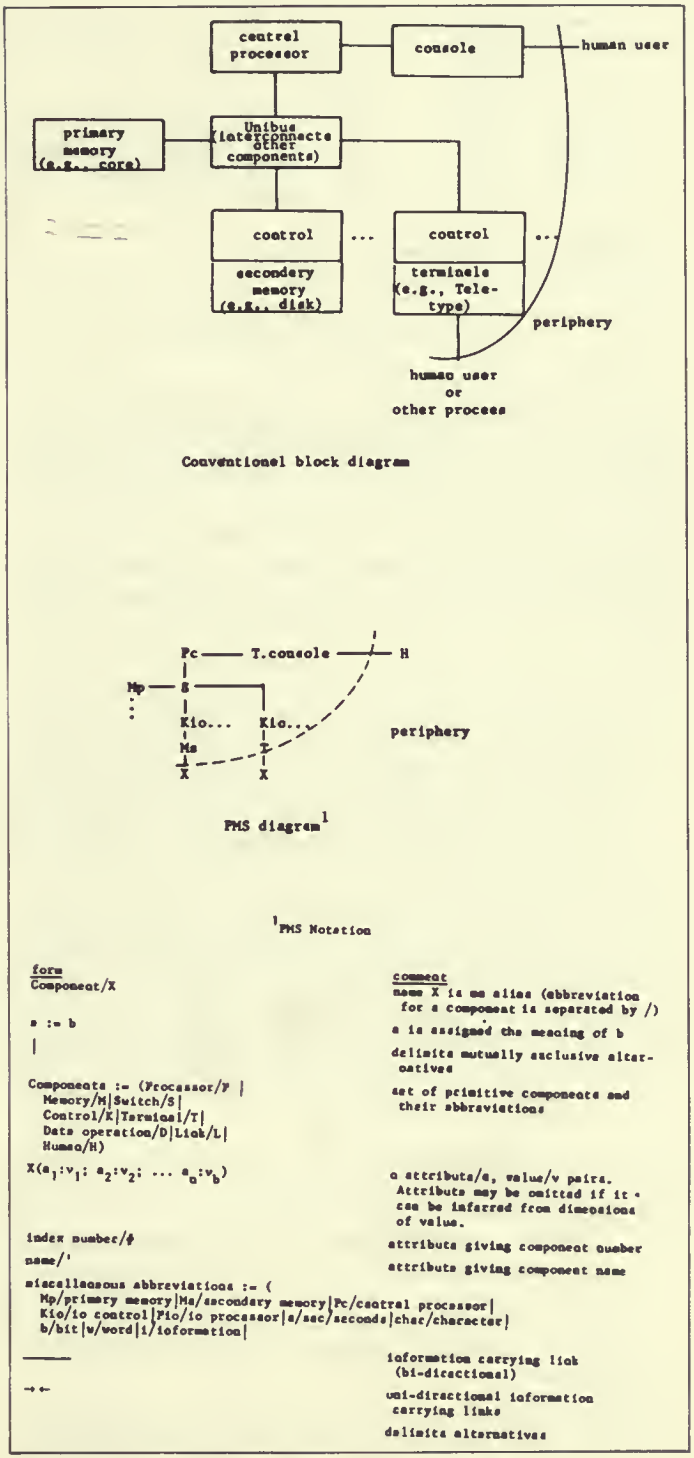

Fig. 1. Conventional block diagram and PMS diagram of PDP-11.

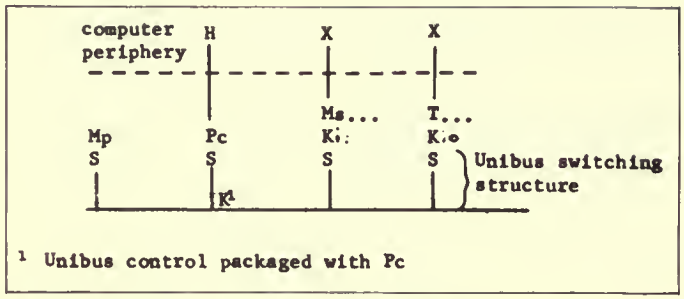

Fig. 2. PDP-11 physical structure PMS diagram.

switch for placing messages on the bus or taking messages off the bus. The central control decides the next component to use the bus for a message (call). The S (Unibus) differs from most switches because any component can communicate with any other component.

The types of messages in the PDP-11 are along the lines of the hierarchical structure common to present day computers. The single bus makes conventional and other structures possible. The message processes in the structure which utilize S (Unibus) are:

1 The central processor (Pc) requests that data be read or written from or to primary memory (Mp) for instructions and data. The processor calls a particular memory module by concurrently specifying the module's address, and the address within the modules. Depending on whether the processor requests reading or writing, data is transmitted either from the memory to the processor or vice versa.

2 The central processor (Pc) controls the initialization of secondary memory (Ms) and terminal (T) activity. The processor sets status bits in the control associated with a particular Ms or T, and the device proceeds with the specified action (e.g., reading a card, or punching a character into paper tape). Since some devices transfer data vectors directly to primary memory, the vector control information (i.e., the memory location and length) is given as initialization information.

3 Controls request the processor's attention in the form of interrupts. An interrupt request to the processor has the effect of changing the state of the processor; thus the processor begins executing a program associated with the interrupting process. Note, the interrupt process is only a signaling method, and when the processor interruption occurs, the interruptee specifies a unique address value to the processor. The address is a starting address for a program.

4 The central processor can control the transmission of data between a control (for T or Ms) and either the processor or a primary memory for program controlled data transfers. The device signals for attention using the interrupt dialogue and the central processor responds by managing the data transmission in a fashion similar to transmitting initialization information.

5 Some device controls (for T or Ms) transfer data directly to/from primary memory without central processor intervention. In this mode the device behaves similar to a processor; a memory address is specified, and the data is transmitted between the device and primary memory.

6 The transfer of data between two controls, e.g., a secondary memory (disk) and say a terminal/T.display is not precluded, provided the two use compatible message formats.

As we show more detail in the structure there are, of course, more messages (and more simultaneous activity). The above does not describe the shared control and its associated switching which is typical of a magnetic tape and magnetic disk secondary memory systems. A control for a DECtape memory (Fig. 3) has an S(' DECtape bus) for transmitting data between a single tape unit and the DECtape transport. The existence of this kind of structure is based on the relatively high cost of the control relative to the cost of the tape and the value of being able to run concurrently with other tapes. There is also a dialogue at the periphery between X-T and X-Ms which does not use the Unibus. (For example, the removal of a magnetic tape reel from a tape unit or a human user (H) striking a typewriter key are typical dialogues.)

All of these dialogues lead to the hierarchy of present computers (Fig. 4). In this hierarchy we can see the paths by which the above messages are passed (Pc-Mp; Pc-K; K-Pc; Kio-T and Kio-Ms; and Kio-Mp; and, at the periphery, T-X and T-Ms; and T.console-H).

Model 20 Implementation

Figure 5 shows the detailed structure of a uni-processor, Model 20 PDP-11 with its various components (options). In Fig. 5 the Unibus characteristics are suppressed. (The detailed properties of the switch are described in the logical design section.)

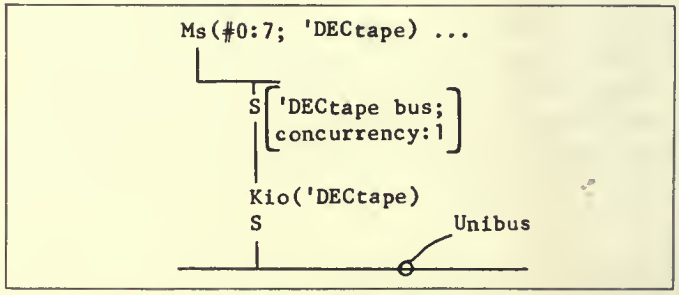

Fig. 3. DECtape control switching PMS diagram.

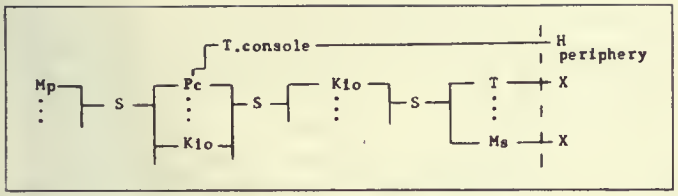

Fig. 4. Conventional hierarchy computer structure.

Extensions to Increase Performance

The reader should note (Fig. 5) that the important limitations of the bus are: a concurrency of one, namely, only one dialogue can occur at a given time, and a maximum transfer rate of one 16-bit word per .75 μsec., giving a transfer rate of 21.3 megabits/second. While the bus is not a limit for a uni-processor structure, it is a limit for multiprocessor structures. The bus also imposes an artificial limit on the system performance when high speed devices (e.g., TV cameras, disks) are transferring data to multiple primary memories. On a larger system with multiple independent memories the supply of memory cycles is 17 megabits/second times the number of modules. Since there is such a large supply of memory cycles/second and since the central processor can only

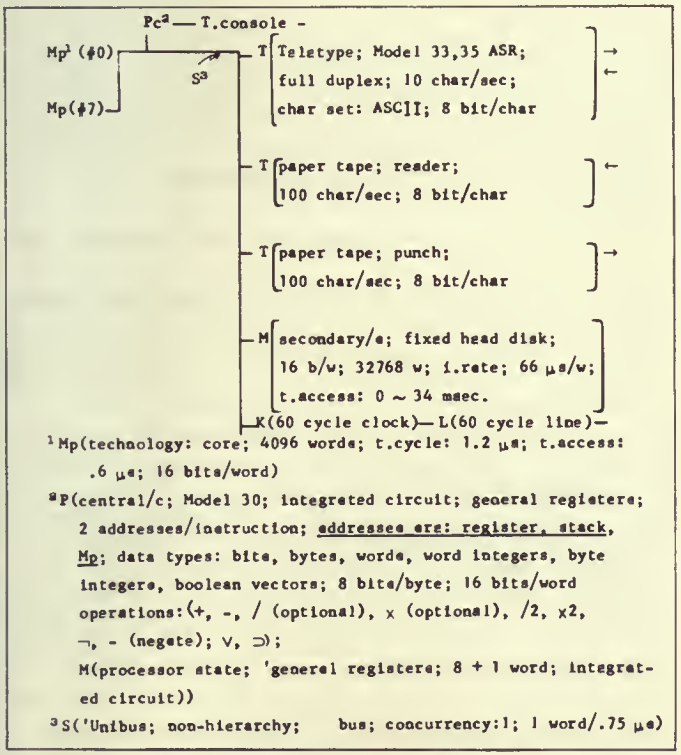

Fig. 5. PDP-11 structure and characteristics PMS diagram.

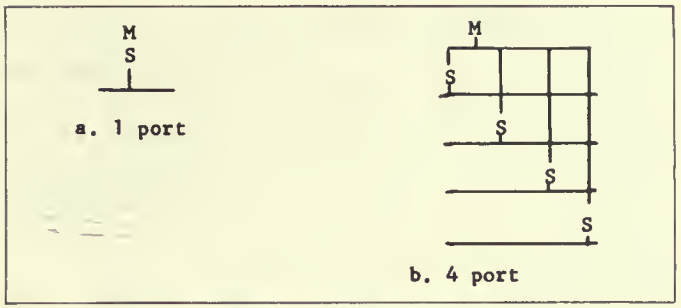

Fig. 6. 1 and 4 port memory modules PMS diagram.

absorb approximately 16 megabits/second, the simple one Unibus structure must be modified to make the memory cycles available. Two changes are necessary: first, each of the memory modules have to be changed so that multiple units can access each module on an independent basis; and second, there must be independent control accessing mechanisms. Figure 6 shows how a single memory is modified to have more access ports (i.e., connect to 4 Unibusses).

Figure 7 shows a system with 3 independent memory modules which are accessed by 2 independent Unibusses. Note that two of the secondary memories and one of the transducers are connected to both Unibusses. It should be noted that devices which can potentially interfere with Pc-Mp accesses are constructed with two ports; for simple systems, the two ports are both connected to the same bus, but for systems with more busses, the second connection is to an independent bus.

Figure 8 shows a multiprocessor system with two central processors and three Unibusses. Two of the Unibus controls are included within the two processors, and the third bus is controlled by an independent control unit. The structure also has a second switch to allow either of two processors (Unibusses) to access common shared devices. The interrupt mechanism allows either processor to respond to an interrupt and similarly either processor may issue initialization information on an anonymous basis. A

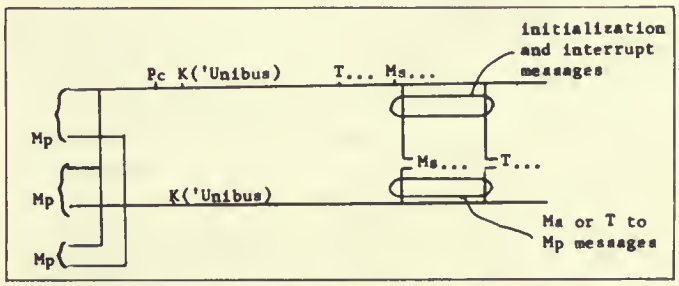

Fig. 7. Three Mp, 2 S('Unibus) structure PMS diagram.

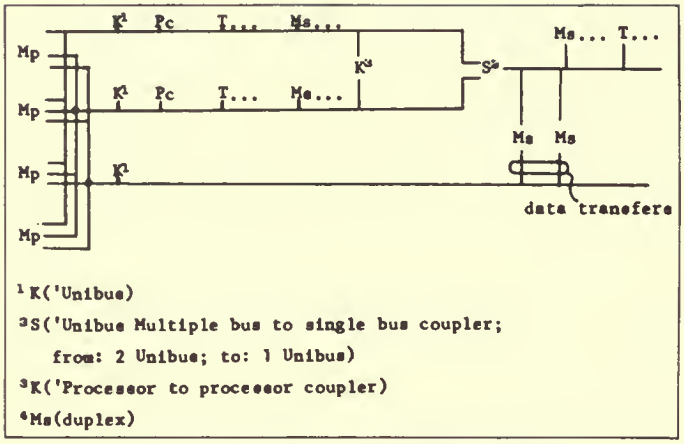

Fig. 8. Dual Pc multiprocessor system PMS diagram.

control unit is needed so that two processors can communicate with one another; shared primary memory is normally used to carry the body of the message. A control connected to two Pc's (see Fig. 8) can be used for reliability; either processor or Unibus could fail, and the shared Ms would still be accessible.

Higher Performance Processors

Increasing the bus width has the greatest effect on performance. A single bus limits data transmission to 21.3 megabits/second, and though Model 20 memories are 16 megabits/second, faster (or wider) data path width modules will be limited by the bus. The Model 20 is not restricted, but for higher performance processors operating on double word (fixed point) or triple word (floating point) data two or three accesses are required for a single data type. The direct method to improve the performance is to double or triple the primary memory and central processor data path widths. Thus, the bus data rate is automatically doubled or tripled.

For 32- or 48-bit memories a coupling control unit is needed so that devices of either width appear isomorphic to one another. The coupler maps a data request of a given width into a higher- or lower-width request for the bus being coupled to, as shown in Fig. 9. (The bus is limited to a fixed number of devices for electrical reasons; thus, to extend the bus a bus repeating unit is needed. The bus repeating control unit is almost identical to the bus coupler.) A computer with a 48-bit primary memory and processor and 16-bit secondary memory and terminals (transducers) is shown in Fig. 9.

In summary, the design goal was to have a modular structure providing the final user with freedom and flexibility to match his needs. A secondary goal of the Unibus is open-endedness by

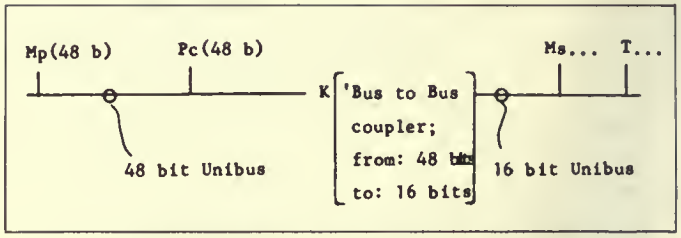

Fig. 9. Computer with 48 bit Pc, Mp with 16 bit Ms, T PMS diagram.

providing multiple busses and defining wider path busses. Finally, and most important, the Unibus is straightforward.

The Instruction Set Processor (ISP) Level-Architecture[1]

Introduction, Background and Design Constraints

The Instruction Set Processor (ISP) is the machine defined by hardware and/or software which interprets programs. As such, an ISP is independent of technology and specific implementations.

The instruction set is one of the least understood aspects of computer design; currently it is an art. There is currently no theory of instruction sets, although there have been attempts to construct them [Maurer, 1966], and there has also been an attempt to have a computer program design an instruction set [Haney, 1968]. We have used the conventional approach in this design: first a basic ISP was adopted and then incremental design modifications were made (based on the results of the benchmarks).[2]

Although the approach to the design was conventional, the

[1]The word architecture has been operationally defined [Amdahl, Blaauw, and Brooks, 1964] as "the attributes of a system as seen by a 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]A predecessor multiregister computer was proposed which used a similar design process. Benchmark programs were coded on each of 10 "competitive" machines, and the object of the design was to get a machine which gave the best score on the benchmarks. This approach had several fallacies: the machine had no basic character of its own; the machine was difficult to program since the multiple registers were assigned to specific functions and had inherent idiosyncrasies to score well on the benchmarks; the machine did not perform well for programs other than those used in the benchmark test; and finally, compilers which took advantage of the machine appeared to be difficult to write. Since all "competitive machines" had been hand-coded from a common flowchart rather than separate flowcharts for each machine, the apparent high performance may have been due to the flowchart organization.

resulting machine is not. A common classification of processors is as zero-, one-, two-, three-, or three-plus-one-address machines. This scheme has the form:

op l1, l2, l3, l4

where *l1* specifies the location (address) in which to store the result of the binary operation (*op*) of the contents of operand locations *l2* and *l3*, and *l4* specifies the location of the next instruction.
The action of the instruction is of the form:

$l1 \leftarrow l2$ *op* $l3$; *goto* $l4$

The other addressing schemes assume specific values for one or more of these locations. Thus, the one-address von Neumann [Burks, Goldstine, and von Neumann, 1962] machines assumes $l1 = l2 =$ the "accumulator" and *l4* is the location following that of the current instruction. The two-address machine assumes $l1 = l2$; *l4* is the next address.

Historically, the trend in machine design has been to move from a 1 or 2 word accumulator structure as in the von Neumann machine towards a machine with accumulator and index register(s).[1] As the number of registers is increased the assignment of the registers to specific functions becomes more undesirable and inflexible; thus, the general-register concept has developed. The use of an array of general registers in the processor was apparently first used in the first-generation, vacuum-tube machine, PEGASUS [Elliott et al., 1956] and appears to be an outgrowth of both 1- and 2-address structures. (Two alternative structures—the early 2- and 3-address per instruction computers may be disregarded, since they tend to always access primary memory for results as well as temporary storage and thus are wasteful of time and memory cycles, and require a long instruction.) The stack concept (zero-address) provides the most efficient access method for specifying algorithms, since very little space, only the access addresses and the operators, needs to be given. In this scheme the operands of an operator are always assumed to be on the "top of the stack." The stack has the additional advantage that arithmetic expression evaluation and compiler statement parsing have been developed to use a stack effectively. The disadvantage of the stack is due in part to the nature of current memory technology. That is, stack memories have to be simulated with random access memories, multiple stacks are usually required, and even though small stack memories exist, as the stack overflows, the primary memory (core) has to be used.

Even though the trend has been toward the general register concept (which, of course, is similar to a two-address scheme in which one of the addresses is limited to small values), it is important to recognize that any design is a compromise. There are situations for which any of these schemes can be shown to be "best." The IBM System/360 series uses a general register structure, and their designers [Amdahl, Blaauw, and Brooks, 1964] claim the following advantages for the scheme:

1 Registers can be assigned to various functions: base addressing, address calculation, fixed point arithmetic and indexing.

2 Availability of technology makes the general registers structure attractive.

The System/360 designers also claim that a stack organized machine such as the English Electric KDF 9 [Allmark and Lucking, 1962] or the Burroughs B5000 [Lonergan and King, 1961] has the following disadvantages:

1 Performance is derived from fast registers, not the way they are used.

2 Stack organization is too limiting and requires many copy and swap operations.

3 The overall storage of general registers and stack machines are the same, considering point 2.

4 The stack has a bottom, and when placed in slower memory there is a performance loss.

5 Subroutine transparency is not easily realized with one stack.

6 Variable length data is awkward with a stack.

We generally concur with points 1, 2, and 4. Point 5 is an erroneous conclusion, and point 6 is irrelevant (that is, general register machines have the same problem). The general-register scheme also allows processor implementations with a high degree of parallelism since instructions of a local block all can operate on several registers concurrently. A set of truly general purpose registers should also have additional uses. For example, in the DEC PDP-10, general registers are used for address integers, indexing, floating point, boolean vectors (bits), or program flags and stack pointers. The general registers are also addressable as primary memory, and thus, short program loops can reside within them and be interpreted faster. It was observed in operation that PDP-10 stack operations were very powerful and often used (accounting for as many as 20% of the executed instructions, in some programs, e.g., the compilers.)

The basic design decision which sets the PDP-11 apart was based on the observation that by using *truly* general registers and by suitable addressing mechanisms it was possible to consider the

[1] Due in part to needs, but mainly technology which dictates how large the structure can be.

machine as a zero-address (stack), one-address (general register), or two-address (memory-to-memory) computer. Thus, it is possible to use whichever addressing scheme, or mixture of schemes, is most appropriate.

Another important design decision for the instruction set was to have only a few data types in the basic machine, and to have a rather complete set of operations for each data type. (Alternative designs might have more data types with few operations, or few data types with few operations.) In part, this was dictated by the machine size. The conversion between data types must be easily accomplished either automatically or with 1 or 2 instructions. The data types should also be sufficiently primitive to allow other data types to be defined by software (and by hardware in more powerful versions of the machine). The basic data type of the machine is the 16 bit integer which uses the two's complement convention for sign. This data type is also identical to an address.

PDP-11 Model 20 Instruction Set (Basic Instruction Set)

A formal description of the basic instruction set is given in Appendix 1 using the ISP notation [Bell and Newell, 1971]. The remainder of this section will discuss the machine in a conventional manner.

Primary Memory. The primary memory (core) is addressed as either 2^{16} bytes or 2^{15} words using a 16 bit number. The linear address space is also used to access the input-output devices. The device state, data and control registers are read or written like normal memory locations.

General Register. The general registers are named: R[0:7] <15:0>[1]; that is, there are 8 registers each with 16 bits. The naming is done starting at the left with bit 15 (the sign bit) to the least significant bit 0. There are synonyms for R[6] and R[7]:

> Stack Pointer/SP<15:0> := R[6]<15:0>. Used to access a special stack which is used to store the state of interrupts, traps and subroutine calls
>
> Program Counter/PC<15:0> := R[7]<15:0>. Points to the current instruction being interpreted. It will be seen that the fact that PC is one of the general registers is crucial to the design.

Any general register, R[0:7], can be used as a stack pointer. The special Stack Pointer (SP) has additional properties that force it to be used for changing processor state interrupts, traps, and subroutine calls (It also can be used to control dynamic temporary storage subroutines.)

[1]A definition of the ISP notation used here may be found in Chapter 4.

In addition to the above registers there are 8 bits used (from a possible 16) for processor status, called PS<15:0> register. Four bits are the Condition Codes (CC) associated with arithmetic results; the T-bit controls tracing; and three bits control the priority of running programs Priority <2:0>. Individual bits are mapped in PS as shown in Appendix 1.

Data Types and Primitive Operations. There are two data lengths in the basic machine: bytes and words, which are 8 and 16 bits, respectively. The non-trivial data types are word length integers (w.i.); byte length integers (by.i); word length boolean vectors (w.bv), i.e., 16 independent bits (booleans) in a 1 dimensional array; and byte length boolean vectors (by.bv). The operations on byte and word boolean vectors are identical. Since a common use of a byte is to hold several flag bits (booleans), the operations can be combined to form the complete set of 16 operations. The logical operations are: "clear," "complement," "inclusive or," and "implication" ($x \supset y$ or $\neg x \vee y$).

There is a complete set of arithmetic operations for the word integers in the basic instruction set. The arithmetic operations are: add, subtract, multiply (optional), divide (optional), compare, add one, subtract one, clear, negate, and multiply and divide by powers of two (shift). Since the address integer size is 16 bits, these data types are most important. Byte length integers are operated on as words by moving them to the general registers where they take on the value of word integers. Word length integer operations are carried out and the results are returned to memory (truncated).

The floating point instructions defined by software (not part of the basic instruction set) require the definition of two additional data types (of length two and three), i.e., double word (d.w.) and triple (t.w.) words. Two additional data types, double integer (d.i.) and triple floating point (t.f. or f) are provided for arithmetic. These data types imply certain additional operations and the conversion to the more primitive data types.

Address (Operand) Calculation. The general methods provided for accessing operands are the most interesting (perhaps unique) part of the machine's structure. By defining several access methods to a set of general registers, to memory, or to a stack (controlled by a general register), the computer is able to be a 0, 1 and 2 address machine. The encoding of the instruction Source (S) fields and Destination (D) fields are given in Fig. 10 together with a list of the various access modes that are possible. (Appendix 1 gives a formal description of the effective address calculation process.)

It should be noted from Fig. 10 that all the common access modes are included (direct, indirect, immediate, relative, indexed, and indexed indirect) plus several relatively uncommon ones. Relative (to PC) access is used to simplify program loading,

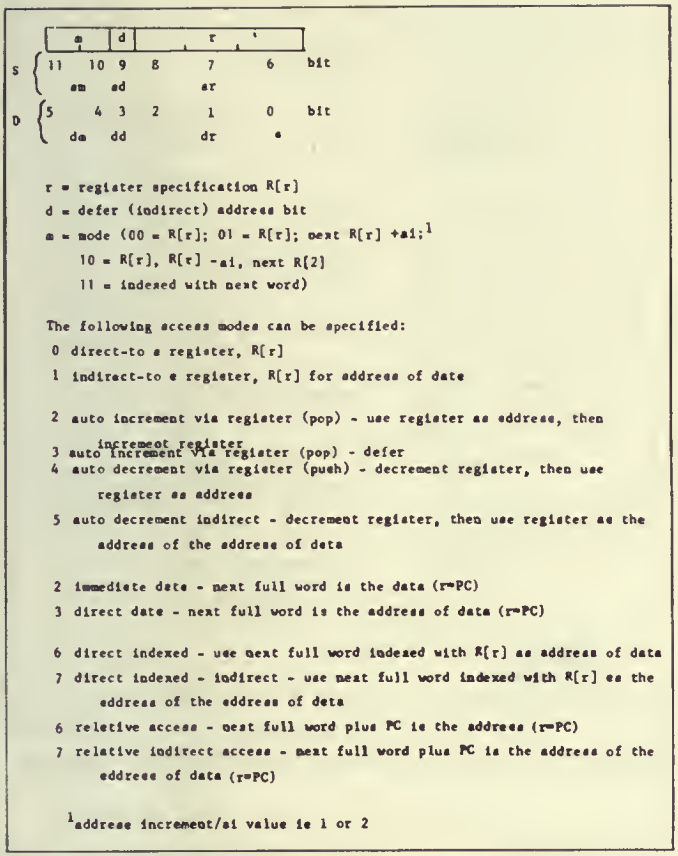

Fig. 10. Address calculation formats.

Assembler Format	Effect	Description
Two Address Machine format:		
MOVE B,A[1]	A ← B	replace A with contents of B
MOVE #N,A	A ← N	replace A with number, N
MOVE B(RZ), A(RZ)	A[I] ← B[I]	replace element of a connector
MOVE (R_3) +, (R_4) +	A[I] ← B[I]; I ← I + 1	replace element of a vector, move to next element
General Register Machine format:		
MOVE A,R1	R1 ← A	load register
MOVE R1, A	A ← R1	store register
MOVE @A,R1	R1 ← M[A]	load or store indirect via element A
MOVE R1, R3	R1 ← R3	register to register transfer
MOVE R1, A(RZ)	A[I] ← R1	store indexed (load indexed) (or store)
MOVE @A(R0),R1	R1 ← M[A[I]]	load (or store) indexed indirect
MOVE (R1), R3	R1 ← M[R2]	load indirect via register
MOVE (R1) +, R3	R3 ← M[I]	load (or store) element indirect via register, move to next element
Stack Machine format:		
MOVE #N, -(R0)	S ← N	load stack with literal
MOVE A, -(R0)	S ← A	load stack with contents of A
MOVE @(R0)+, -(R0)	S ← M[S]	load stack with memory specified by top of stack
MOVE (R0)+, A	A ← S	store stack in A
MOVE (R0)+, @(R0)+	$M[S_2]$ ← S_1	store stack top in memory addressed by stack top -1
MOVE (R0), -(R0)	S ← S	duplicate top of stack

[1]Assembler format:
() denotes contents of memory addressed by
- decrement register first
+ increment register after
@ indirect
# literal

Fig. 11. Coding for the MOVE instruction to compare with conventional machines.

while immediate mode speeds up execution. The relatively uncommon access modes, auto-increment and auto-decrement, are used for two purposes: access to stack under control of the registers[1] and access to bytes or words organized as strings or vectors. The indirect access mode allows a stack to hold addresses of data (instead of data). This mode is desirable when manipulating longer and variable-length data types (e.g., strings, double fixed and triple floating point). The register auto increment mode may be used to access a byte string; thus, for example, after each access, the register can be made to point to the next data item. This is used for moving data blocks, searching for particular elements of a vector, and byte-string operations (e.g., movement, comparisons, editing).

This addressing structure provides flexibility while retaining the same, or better, coding efficiency than classical machines. As an example of the flexibility possible, consider the variations possible with the most trivial word instruction MOVE (see Fig. 11). The MOVE instruction is coded as it would appear in conventional 2-address, 1-address (general register) and 0-address (stack) computers. The two-address format is particularly nice for MOVE, because it provides an efficient encoding for the common operation: A ← B (note, the stack and general registers are not involved). The vector move A[I] ← B(I) is also efficiently encoded. For the general register (and 1-address format), there are about 13 MOVE operations that are commonly used. Six moves can be encoded for the stack (about the same number found in stack machines).

Instruction Formats. There are several instruction decoding formats depending on whether 0, 1, or 2 operands have to be explicitly referenced. When 2 operands are required, they are identified as Source/S and Destination/D and the result is placed at Destination/D. For single operand instructions (unary operators) the instruction action is D ← u D; and for two operand instructions (binary operators) the action is D ← D b S (where u and b are unary and binary operators, e.g., ¬, − and +, −, ×, /,

[1]Note, by convention a stack builds toward register 0, and when the stack crosses 400_8, a stack overflow occurs.

respectively. Instructions are specified by a 16-bit word. The most common binary operator format (that for operations requiring two addresses) is shown below.

15 . . . 12:	11 . . . 6:	5 . . . 0
op	D	S

The other instruction formats are given in Fig. 12.

Instruction Interpretation Process. The instruction interpretation process is given in Fig. 13, and follows the common fetch-execute cycle. There are three major states: (1) interrupting—the PC and PS are placed on the stack accessed by the Stack Pointer/SP, and the new state is taken from an address specified by the source requesting the trap or interrupt; (2) trace (controlled by T-bit)—essentially one instruction at a time is executed as a trace trap occurs after each instruction; and (3) normal instruction interpretation. The five (lower) states in the diagram are concerned with instruction fetching, operand fetching, executing the operation specified by the instruction and storing the result. The non-trivial details for fetching and storing the operands are not shown in the diagram but can be constructed from the effective address calculation process (Appendix 1). The state diagram, though simplified, is similar to 2- and 3-address computers, but is distinctly different than a 1 address (1 accumulator) computer.

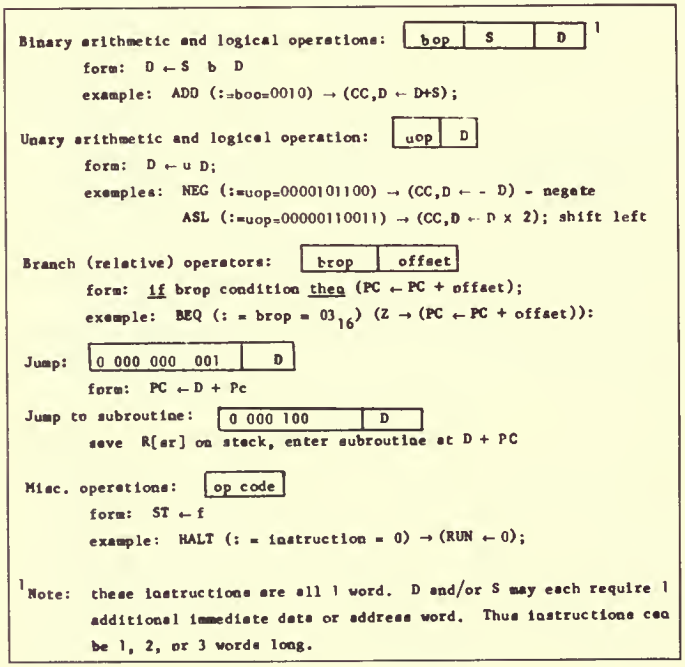

Fig. 12. PDP-11 instruction formats (simplified).

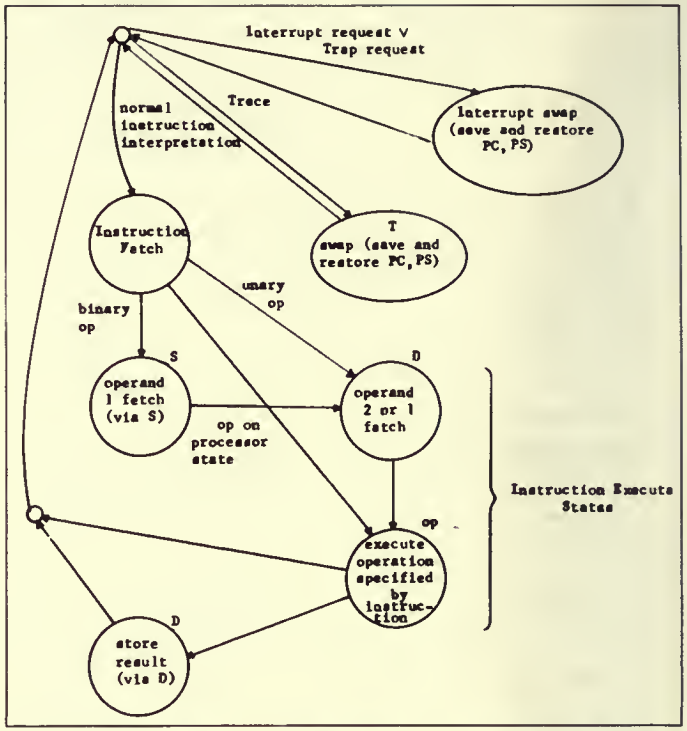

Fig. 13. PDP-11 instruction interpretation process state diagram.

The ISP description (Appendix 1) gives the operation of each of the instructions, and the more conventional diagram (Fig. 12) shows the decoding of instruction classes. The ISP description is somewhat incomplete; for example, the add instruction is defined as: ADD (:= bop = 0010) ← (CC,D ← D + S); *addition* does not exactly describe the changes to the Condition Codes/CC (which means whenever a binary opcode [bop] of 0010_2 occurs the ADD instruction is executed with the above effect). In general, the CC are based on the result, that is, Z is set if the result is zero, N if negative, C if a carry occurs, and V if an overflow was detected as a result of the operation. Conditional branch instructions may thus follow the arithmetic instruction to test the results of the CC bits.

Examples of Addressing Schemes

Use as a Stack (Zero Address) Machine. Figure 14 lists typical zero-address machine instructions together with the PDP-11 instructions which perform the same function. It should be noted that translation (compilation) from normal infix expressions to reverse Polish is a comparatively trivial task. Thus, one of the primary reasons for using stacks is for the evaluation of expressions in reverse Polish form.

Common stack instruction:	Equivalent PDP-11 instruction:
place address value A on stack	MOVE #A, -(R0)[1]
load stack from memory address specified by stack	MOVE @(R0)+, - (R0)
load stack from memory location A	MOVE A, -(R0)
store stack at memory address specified by stack	MOVE (R0)+, @(R0)+
store stack at memory location A	MOVE (R0)+, A
duplicate top of stack	MOVE (R0), -(R0)
+, add 2 top data of stack to stack	ADD (R0) +, @R0
-, x, /; subtract, multiply, divide	(see add)
-; negate top data of stack	NEG @R0
clear top data of stack	CLR @R0
v; "inclusive or" 2 top data of stack "and" 2 top data of stack	BSET (R0)+, @R0
–; complement top of stack	COM @R0
test top of stack (set branch indicators)	TST @R0
branch on indicator	BR (=, ≠, >, ≥, <, ≤)
jump unconditional	JUMP

add addressed location A to top of stack - (not common for stack machine) equivalent to: load stack, add swap top 2 stack data	ADD A, @R0
	MOVE (R0)+, R1
	MOVE (R0)+, R2
	MOVE R1, -(R0)
	MOVE R2, -(R0)

reset stack location to N	MOVE #N, R0 COM @R0
A, "and" 2 top stack data	BCLR (R0)+, @R0

[1] Stack pointer has been arbitrarily used as register R0 for this example.

Fig. 14. Stack computer instructions and equivalent PDP-11 instructions.

Consider an assignment statement of the form

$$D \leftarrow A + B/C$$

which has the reverse Polish form

$$DABC/+ \leftarrow$$

and would normally be encoded on a stack machine as follows

```
load stack address of D
load stack A
load stack B
load stack C
/
+
store
```

However, with the PDP-11 there is an address method for improving the program encoding and run time, while not losing the stack concept. An encoding improvement is made by doing an operation to the top of the stack from a direct memory location (while loading). Thus the previous example could be coded as:

```
load stack B
divide stack by C
add A to stack
store stack D
```

Use as a One-Address (General Register) Machine. The PDP-11 is a general register computer and should be judged on that basis. Benchmarks have been coded to compare the PDP-11 with the larger DEC PDP-10. A 16 bit processor performs better than the DEC PDP-10 in terms of bit efficiency, but not with time or memory cycles. A PDP-11 with a 32 bit wide memory would, however, decrease time by nearly a factor of two, making the times essentially comparable.

Use as a Two-Address Machine. Figure 15 lists typical two-address machine instructions together with the equivalent PDP-11 instructions for performing the same operations. The most useful instruction is probably the MOVE instruction because it does not use the stack or general registers. Unary instructions which operate on and test primary memory are also useful and efficient instructions.

Extensions of the Instruction Set for Real (Floating Point) Arithmetic

The most significant factor that affects performance is whether a machine has operators for manipulating data in a particular format. The inherent generality of a stored program computer allows any computer by subroutine to simulate another—given enough time and memory. The biggest and perhaps only factor that separates a small computer from a large computer is whether floating point data is understood by the computer. For example, a small computer with a cycle time of 1.0 microseconds and 16 bit memory width might have the following characteristics for a floating point add, excluding data accesses:

Programmed	250 microseconds
Programmed but (special normalize and differencing of exponent instructions)	75 microseconds
Microprogrammed hardware	25 microseconds
Hardwired	2 microseconds

It should be noted that the ratios between programmed and hardwired interpretation varies by roughly two orders of magnitude. The basic hardwiring scheme and the programmed scheme should allow binary program compatibility, assuming there is an interpretive program for the various operators in the Model 20. For example, consider one scheme which would add eight 48 bit registers which are addressable in the extended instruction set.

Two Address Computer	PDP-11
A ← B; transfer B to A	MOVE B,A
A ← A+B; add	ADD B,A
-, x, /	(see add)
A ← -A; negate	NEG A
A ← A ∨ B; inclusive or	%SETB,A
A ← ¬ A; not	COM
Jump unconditioned	JUMP
Test A, and transfer to B	TST A
	BR (=, ≠, >, ≥, <, ≤) B

Fig. 15. Two address computer instructions and equivalent PDP-11 instructions.

The eight floating registers, F, would be mapped into eight double length (32 bit) registers, D. In order to access the various parts of F or D registers, registers F0 and F1 are mapped onto registers R0 to R2 and R3 to R5.

Since the instruction set operation code is almost completely encoded already for byte and word length data, a new encoding scheme is necessary to specify the proposed additional instructions. This scheme adds two instructions: enter floating point mode and execute one floating point instruction. The instructions for floating point and double word data would be:

binary ops			*op*	*floating point/f*	*and double word/d*
bop'	S	D	←	FMOVE	DMOVE
			+	FADD	DADD
			−	FSUB	DSUB
			×	FMUL	DMUL
			/	FDIV	DDIV
			compare	FCMP	DCMP
unary ops					
uop'	D		−	FNEG	DNEG

Logical Design of S(Unibus) and Pc

The logical design level is concerned with the physical implementation and the constituent combinatorial and sequential logic elements which form the various computer components (e.g., processors, memories, controls). Physically, these components are separate and connected to the Unibus following the lines of the PMS structure.

Unibus Organization

Figure 16 gives a PMS diagram of the Pc and the entering signals from the Unibus. The control unit for the Unibus, housed in Pc for the Model 20, is not shown in the figure.

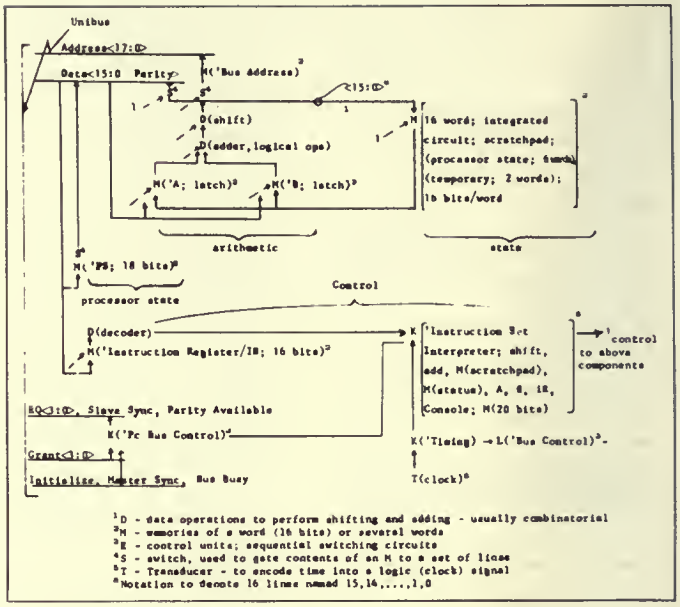

Fig. 16. PDP-11 Pc structure.

The PDP-11 Unibus has 56 bi-directional signals conventionally used for program-controlled data transfers (processor to control), direct-memory data transfers (processor or control to memory) and control-to-processor interrupt. The Unibus is interlocked; thus transactions operate independently of the bus length and response time of the master and slave. Since the bus is bi-directional and is used by all devices, any device can communicate with any other device. The controlling device is the master, and the device to which the master is communicating is the slave. For example, a data transfer from processor (master) to memory (always a slave) uses the Data Out dialogue facility for writing and a transfer from memory to processor uses the Data In dialogue facility for reading.

Bus Control. Most of the time the processor is bus master fetching instructions and operands from memory and storing results in memory. Bus mastership is determined by the current processor priority and the priority line upon which a bus request is made and the physical placement of a requesting device on the linked bus. The assignment of bus mastership is done concurrently with normal communication (dialogues).

Unibus Dialogues

Three types of dialogues use the Unibus. All the dialogues have a common protocol which first consists of obtaining the bus mastership (which is done concurrently with a previous transaction)

followed by a data exchange with the requested device. The dialogues are: Interrupt; Data In and Date In Pause; and Data Out and Data Out Byte.

Interrupt. Interrupt can be initiated by a master immediately after receiving bus mastership. An address is transmitted from the master to the slave on Interrupt. Normally, subordinate control devices use this method to transmit an interrupt signal to the processor.

Data In and Data In Pause. These two bus operations transmit slave's data (whose address is specified by the master) to the master. For the Data In Pause operation data is read into the master and the master responds with data which is to be rewritten in the slave.

Data Out and Data Out Byte. These two operations transfer data from the master to the slave at the address specified by the master. For Data Out a word at the address specified by the address lines is transferred from master to slave. Data Out Byte allows a single data byte to be transmitted.

Processor Logical Design

The Pc is designed using TTL logical design components and occupies approximately eight 8″ × 12″ printed circuit boards. The organization of the logic is shown in Fig. 16. The Pc is physically connected to two other components, the console and the Unibus. The control for the Unibus is housed in the Pc and occupies one of the printed circuit boards. The most regular part of the Pc, the arithmetic and state section, is shown at the top of the figure. The 16-word scratch-pad memory and combinatorial logic data operators, D(shift) and D(adder, logical ops), form the most regular part of the processor's structure. The 16-word memory holds most of the 8-word processor state found in the ISP, and the 8 bits that form the Status word are stored in an 8-bit register. The input to the adder-shift network has two latches which are either memories or gates. The output of the adder-shift network can be read to either the data or address parts of the Unibus, or back to the scratch-pad array.

The instruction decoding and arithmetic control are less regular than the above data and state and these are shown in the lower part of the figure. There are two major sections: the instruction fetching and decoding control and the instruction set interpreter (which in effect defines the ISP). The later control section operates on, hence controls, the arithmetic and state parts of the Pc. A final control is concerned with the interface to the Unibus (distinct from the Unibus control that is housed in the Pc).

Conclusions

In this paper we have endeavored to give a complete description of the PDP-11 Model 20 computer at four descriptive levels. These present an unambiguous specification at two levels (the PMS structure and the ISP), and, in addition, specify the constraints for the design at the top level, and give the reader some idea of the implementation at the bottom level logical design. We have also presented guidelines for forming additional models that would belong to the same family.

References

Allmark and Lucking [1962]; Amdahl, Blaauw, and Brooks [1964]; Bell and Newell [1971]; Burks, Goldstine, and Von Neumann [1962]; Elliott, Owen, Devonald, and Maudsley [1956]; Haney [1968]; Lonergan and King [1961]; Maurer [1966]; Rothman [1959]; Wilkes [1951].

APPENDIX 1 PDP-11 ISP

```
PDP11 :=
    begin

! This is a summary description of a PDP-11/70 processor written
! in the ISPS language.

! This summary explicitly defines the instruction fetch and execute
! cycles of the PDP-11/70.

! Most of the actual instruction execution descriptions have been
! eliminated.  However, at least one instruction from each of
! the major instruction classes is described in full.

! The memory management description has been eliminated from this summary.

! The register mapping ROM initialization has been eliminated
! from the summary.  If simulations are performed, REGROM[63:0]
! should be initialized by use of an external READ file.

**MP.State**

! Macro definitions to allow easy change of memory configuration.
! The 11/70 allows addressing up to 2M * 2 bytes.  A smaller
! memory is declared for simulator space efficiency.

macro max.byte := |#167777 |.      ! (26k * 2 bytes)

MB[max.byte:0]<7:0>,               ! The addressing space
MW[max.byte:0]<15:0>{increment:2} := MU[max.byte:0]<7:0>,
MB10[#17777777:#17760000]<7:0>,    ! The i/o page (4k)
MW10[#17777777:#17760000]<15:0>{increment:2} := MB10[#17777777:#17760000]<7:0>,

MAR\Memory.addr.reg<21:0>,

MBR\Memory.buff.reg<15:0>,
    bmbr\byte.mbr<7:0>  := MBR<7:0>,


**PC.State**

R\register[15:0]<15:0>, ! Register file including two sets of general
                        ! registers: R0-R5 (address 0000-0101, 1000-1101),
                        ! One program counter (address 0111), and three
                        ! Stack pointers (address 0110,1110,1111)

PC<15:0>   := R['0111]<15:0>,                    ! Only 1 program counter

macro SP   := |R[cmode<0>@'110@(cmode<1> and cmode<0>)] |.  ! Stack pointer (3)
macro link := |R[rs@'101] |.                     ! Two R5's (subroutine link)

PS<15:0>   := MB10[#17777777:#17777776]<7:0>,    ! Program status word
      cm\current.mode<1:0>     := PS<15:14>,     ! Current address space
                                                 ! (kernel/supervisor/user)
      macro kernel             := |(cmode eql '00) |.
      macro super              := |(cmode eql '01) |.
      macro user               := |(cmode eql '11) |.
      pm\previous.mode<1:0>    := PS<13:12>,     ! Previous address space
      p\priority<2:0>          := PS<7:5>,       ! Current process priority
      rs\register.set<>        := PS<11>,
      t\trace<>                := PS<4>,
      cc\condition.codes<3:0>  := PS<3:0>,
          N\negative<>         := cc<3>,
          Z\zero<>             := cc<2>,
          V\overflow<>         := cc<1>,
          C\carry<>            := cc<0>,

! External interrupt requests


br7\bus.request.7<>,                             ! External interrupt requests
br6\bus.request.6<>,
br5\bus.request.5<>,
br4\bus.request.4<>,

ERRREG\cpu.error.register<15:0>          := MB10[#17777767:#17777766]<7:0>,
      illhlt\illegal.halt<>              := ERRREG<7>,
      oddadd\odd.address<>               := ERRREG<6>,
      nomem\non.existent.memory<>        := ERRREG<5>,
      timeout\unibus.time.out<>          := ERRREG<4>,
      yellow\yellow.zone.stack.limit<>   := ERRREG<3>,
      red\red.zone.stack.limit<>         := ERRREG<2>,

SYSID\system.id<15:0>    := MB10[#17777765:#17777764]<7:0>, ! Hardwired Sys No.
a\activity<0:1>,

      macro go           := |(a eql '00)|,
      macro WAIT         := |(a eql '01)|,
      macro HALT         := |(a eql '10)|,

! Trap vector addresses:  The associated error conditions cause execution
! to switch to the PC and PS stored in the two words at the trap address.

      macro cpu.errors               := |#004 |,
      macro ill.instr                := |#010 |,
      macro res.instr                := |#010 |,
      macro bpt.trap                 := |#014 |,
      macro iot.trap                 := |#020 |,
      macro power.fail               := |#024 |,
      macro emt.trap                 := |#030 |,
      macro trap.trap                := |#034 |,

**Implementation.Declarations**

bus.error<>,             ! Bus error detected
byte.access<>,           ! 1 for byte read/write
cmode<1:0>,              ! Temporary for all processing using current mode
busreg<15:0>,            ! Contains trap vector when a trap is set up
dr\destination.reg.addr.in.register.file<3:0>,
iflag<>,                 ! Used with to force I space access
oldval<16:0>,            ! Register value before auto increment/decrement
pc.temp<15:0>,           ! Used during trap routines
pmode<1:0>,              ! Set by mtp and mfp instructions:
                         !       if 0 then normal read/write
                         !       if 1 then use previous instruction space
                         !       if 2 then use previous data space
ps.temp<15:0>,           ! Used during trap routines
regflg<>,                ! Designates register access to read/write procedures
REGROM\register.mapping.read.only.memory[63:0]<3:0>,
sr\source.reg.addr.in.register.file<3:0>,
state<1:0>,              ! Current state 0 = instruction fetch/decode
                         !               1 = execute
                         !               2 = service
                         !               3 = unused
temp<17:0>,
temp1<3:0>,
trace.flag<>,            ! Trace trap bit temporary
trap.instr<>,            ! Set by emt, trap, bpt, and iot to inhibit
var<21:0>,               ! Virtual address register used in read and write
zeros<63:0>,             ! 64 bits of zeros

**Instruction.Format**

i\instruction<15:0>,
    bop\binary.operation<2:0>   := i<14:12>,

IR\Instruction.register<15:0>   := i<15:0>,

    s\source.field<5:0>         := i<11:6>,       ! Source address information
        sm\source.mode<1:0>             := s<5:4>,
        sd\source.deferred<>            := s<3>,
        srcreg\source.reg<2:0>          := s<2:0>,

! Special handling if register 6 (Stack Pointer) or register 7 (Program
! Counter) is used in autoincrement/autodecrement addressing modes.

    macro sr67  := |(s<2:1> eql '11)|,

    d\destination.field<5:0>    := i<5:0>,        ! Destination address info.
        dm\destination.mode<1:0>        := d<5:4>,
        dd\destination.deferred<>       := d<3>,
        desreg\destination.reg<2:0>     := d<2:0>,

    macro dr67  := |(d<2:1> eql '11)|,

! Instruction decoding fields.

    uop\unary.operation<2:0>            := i<8:6>,
    offset<7:0>                         := IR<7:0>,
    rop\register.operation<1:0>         := i<7:6>,
    jetop\jsr.emulator.trap.op<>        := i<15>,
    hynp\byte.operation<>               := i<16>,
    etop\emulator.trap.op<>             := i<8>,
    concop\condition.code.op<10:0>      := i<15:5>,
    cpuop\cpu.control.op<2:0>           := i<2:0>,
    contop\cpu.control.class.op<2:0>    := i<5:3>,
    brop\branch.op.code<2:0>            := i<10:8>,
    intop\extended.integer.op<2:0>      := i<11:9>,
    typeop\class.op.code.bits<1:0>      := i<10:9>,
    resop\reserve.op<>                  := i<11>,
    ccop\condition.code.second.op<>     := i<4>,

**Address.Calculation**

! Source loads the value of the source operand into register source.
! Dest loads the address of the destination operand into register dest
! and fetches the operand to the MBR.

    source()<15:0> :=
        begin
        iflag = 0 next
        DECODE sm =>
            begin
            0 := source = R[sr],                 ! Register mode: registers
                                                 ! are addressed directly.
            1 := begin                           ! Autoincrement mode: use
                 iflag = (sr eql{us} #7);        ! the contents of the specified
                 MAR = R[sr] next                ! register as an address.
                 DECODE (sr67 or sd) =>          ! Increment after use.
                     begin
                     R[sr] = R[sr] + (2 -{us} byte.access),
                     R[sr] = R[sr] + 2
                     end  next
                 read(byte.access *{us} (1 -{us} sd)) next
                 source = MBR; iflag = 0
                 end,
            2 := begin                           ! Autodecrement mode:
                 DECODE (sr67 or sd) =>          ! decrement contents of the
                     begin                       ! specified register before
                                                 ! use.
                     R[sr] = R[sr] - (2 -{us} byte.access),
                     R[sr] = R[sr] - 2
                     end  next
```

APPENDIX 1 *(cont'd.)*

```
                MAR = R[sr] next
                read(byte.access *[us] (1 -[us] sd)) next
                source = MBR
                end,
            3 := begin                          ! Index mode: fetch the next
                 iflag = 1; MAR = PC next       ! word from memory and add
                 PC = PC + 2 next               ! add it to the contents
                                                ! of the specified register
                 read(0) next iflag = 0;        ! to form the effective address.
                 MAR = (MBR + R[sr])<15:0> next ! The register contents
                                                ! are unmodified.
                 read(byte.access *[us] (1 -[us] sd)) next
                 source = MBR
                 end
            end  next

        IF sd =>                                ! Correction for all deferred
            begin                               ! mode addresses.  Use "source"
            MAR = source next                   ! generated above as an address
            read(byte.access) next              ! to a pointer to the true
            source = MBR                        ! source.
            end
        end,
    dest()<15:0> :=
        begin
        iflag = 0; oldval = R[dr] next
        DECODE dm =>
            begin
            0 := begin                          ! Register mode: registers
                 dest = 0; regflg = 1           ! are addressed directly
                 end,
            1 := begin                          ! Autoincrement mode: use
                 dest = R[dr] next              ! of the specified register
                 DECODE (dr07 or dd) =>         ! the contents of the
                     begin                      ! as an address, increment
                                                ! the register after use.
                     R[dr] = R[dr] + (2 -[us] byte.access),
                     R[dr] = R[dr] + 2
                     end next
                 iflag = (dr eql[us] #7)        ! Force I space if using PC
                 end,

            2 := begin                          ! Autodecrement mode:
                 DECODE (dr07 or dd) =>         ! decrement the register
                     begin                      ! then use the contents
                                                ! as an address.
                     R[dr] = R[dr] - (2 -[us] byte.access),
                     R[dr] = R[dr] - 2
                     end  next
                 dest = R[dr]
                 end,

            3 := begin                          ! Index mode: fetch the
                 iflag = 1; MAR = PC next       ! next word from memory and
                 PC = PC + 2 next               ! add it to the contents
                                                ! contents of the specified
                 read(0) next                   ! register to form the
                 iflag = 0; dest = MBR + R[dr]  ! effective address.
                 end                            ! Register contents remain
                                                ! unmodified.
            end  next
        MAR = dest next

        IF dd =>                                ! Correction for all deferred
            begin                               ! mode addresses.  Use
            read(0) next                        ! "destination" generated
            iflag = regflg = 0;                 ! above as an address to a
            MAR = MDR                           ! pointer to the true
            end                                 ! destination.
        end,
**Service.Facilities**

    stkref\stack.reference :=
        begin
        regflg = 0;                             ! Stack op cannot go to regs.
        SP = SP - 2
        end,

    odderr\odd.address.error :=
        begin
        oddadd = bus.error = 1 next
        ckstate(1)
        end,

    ckstate(aborttl<>) :=       ! Check state
        begin
        DECODE state =>
            begin
            no.op(),            ! Ifetch
            LEAVE exec,         ! Execute
            LEAVE service,      ! Service
            no.op()             ! Unused
            end
        end,

    sd.read(byte.acc<>) :=      ! Source, dest, read
        begin
        source() next
        dest() next
        read(byte.acc)
        end,

    d.read(byte.acc<>) :=       ! Dest, read
        begin
        dest() next
        read(byte.acc)
        end,
    read(byte.acc<>) :=
        begin
        IF MAR<0> and not byte.acc => odderr(); var = MAR next
        IF (var<15:13> eql[us] #7) => var<21:6> = #77 next
        DECODE var<21:18> eql[us] #17 =>
            begin
            0 := DECODE regflg =>                           ! Register accessing
                     begin                                  ! check
                     0 := DECODE byte.acc =>                ! not register
                              begin
                              MBR = MW[var],
                              MBR = MD[var]
                              end,
                     1 := DECODE byte.acc =>                ! Register
                              begin
                              MBR =     R[dr],
                              MBR = R[dr]<7:0>
                              end
                     end,
            1 := IF var<17:13> eql[us] #37 =>               ! Yes
                     begin
                     DECODE byte.acc =>                     ! IO.page
                         begin
                         MBR = MWIO[var],
                         MBR = MBIO[var]
                         end
                     end
            end
    end,

    write(byte.acc<>) :=
        begin
        IF MAR<0> and not byte.acc => odderr(); var = MAR next
        IF (var<15:13> eql[us] #7) => var<21:16> = #77 next
        DECODE var<21:18> eql[us] #17 =>
            begin
            0 := DECODE regflg =>                           ! Register accoss
                     begin                                  ! check
                     0 := DECODE byte.acc =>                ! not register
                              begin
                              MW[var] = MBR,
                              MD[var] = bmbr
                              end,
                     1 := DECODE byte.acc =>                ! Register
                              begin
                              R[dr] = MBR,
                              R[dr]<7:0> = bmbr
                              end
                     end,
            1 := IF var<17:13> eql[us] #37 =>               ! Yes
                     begin
                     DECODE byte.acc =>                     ! IO.page
                         begin
                         MWIO[var] = MBR,
                         MBIO[var] = bmbr
                         end
                     end
            end
        end,

! Condition code setting and branch operations

    setcc(n.<15:0>, v.<15:0>, z.<15:0>) :=
        begin
        DECODE byte.access =>
            begin
            0 := begin                              ! Word operation
                 N = n.<15>;
                 V = v.<15:0> eql[us] #100000;
                 Z = z.<15:0> eql 0
                 end,
            1 := begin                              ! Byte operation
                 N = n.<7>;
                 V = v.<7:0> eql[us] #200;
                 Z = z.<7:0> eql 0
                 end
            end
        end,

    branch(condition<>) :=
        begin
        IF condition => PC = PC + (offset s10 1)
        end,

! Interrupt service routines

    bus.reset := (no.op()),

    intvec\interrupt.trap.vector.setup(vector<8:0>) :=
        begin
        MAR = busreg = vector;
        cmode = byte.access = 0 next            ! Map all traps thru
        read(byte.access) next                  ! kernel space
        pc.temp = MBR next
        MAR = busreg + 2 next
        read(byte.access) next
```

APPENDIX 1 *(cont'd.)*

```
        ps.temp = MBR next
        stkref(); MBR = PS next                 ! Old PC and PS saved
        MAR = SP next                           ! in new space
        write(byte.access) next
        stkref(); MBR = PC next
        MAR = SP next
        write(byte.access) next
        pm = cm next
        PC = pc.temp; PS = ps.temp
        end,

    instr.trap\instruction.trap(trap.vector<8:0>) :=    ! Reserved and illegal
        begin                                           ! Opcode service
        intvec(trap.vector) next
        IF bus.error => a = 2   ! Halt the processor if bus error occurs here
        end,

! Trap  and interrupt service  routines.  Service is  called after each
! instruction is complete.  The trap pending of the highest priority is
! activated.   If  a  trap   was  set  by  illegal,  reserved  or  trap
! instructions then the PC and  PS have already been pushed and the new
! PC and  PS are loaded.  An additional trap  is permitted.

    grant\bus.grant.processing.routine(type.request<15:0>) :=
        begin
        a = 0 next
        intvec(type.request) next
        LEAVE service
        end,

    service :=
        begin
        IF bus.error =>
            begin
            bus.error = 0 next                  ! Bus error
            intvec(cpu.errors) next
            IF bus.error =>
                begin
                a = 2 next
                LEAVE service
                end next
            LEAVE service
            end
        end
**Instruction.Interpretation**

    ! Initialization sequence

    start(main) :=
        begin
        zeros = 0;              ! Initialize zeros
        ERRREG = 0;             ! Clear all cpu errors
        a = 0 next              ! Clear activity
        run()
        end,

    ! Main run cycle of the ISP

    run\instruction.interpretation :=
        begin
        IF go =>
            begin
            state = trap.instr = 0;
            MAR = PC next
            DECODE MAR<0> =>            ! Must be even here
                begin
                0 := begin                          ! Even
                     cmode = cm; regflg = 0 next
                     read(0) next                   ! Instruction fetch
                     IR = MBR; PC = PC + 2 next
                     byte.access = byop; trace.flag = t;
                     sr = REGROM[cmode @ rs @ srcreg];
                     dr = REGROM[cmode @ rs @ desreg];
                     state = 1 next
                     exec()
                     end,
                1 := odderr()                       ! Call error routine for
                                                    ! Odd address error processing
                end
            end  next
        IF WAIT => STOP() next
        state = 2 next
        service() next
        RESTART run
        end
**Instruction.Execution**

    exec\instruction.execution :=
        begin
        DECODE bop =>
            begin
            #0 := reserop(),                        ! Reserved op code
            #1 := MOV(),                            ! Move instruction
            #2 := CMP := no.op(),                   ! Compare instruction
            #3 := BIT := no.op(),                   ! Bit test instruction
            #4 := BIC := no.op(),                   ! Bit clear instruction
            #5 := BIS := no.op(),                   ! Bit set instruction
            #6 := begin                             ! Add and subtract
                  DECODE byte.access =>
                      begin
                      '0 := ADD := no.op(),
                      '1 := SUB := no.op()
                      end
                  end,
            #7 := no.op()                           ! Extended instruction set
            end

        end,

    reserop\reserve.op.code :=
        begin
        DECODE resop =>
            begin
            0 := branop(),
            1 := classop()
            end
        end,

    branop\branch.op.codes :=
        begin
        DECODE (jetop @ brop)<3:0> =>
            begin
            #00 := regop(),                         ! Register instruction
            #01 := branch('1),                      ! Branch (br op #00004)
            #02 := BNE  := branch(not Z),           ! Branch if not equal
            #03 := BEQ  := branch(Z),               ! Branch if equal
            #04 := BGE  := branch(N eqv V),         ! Branch if gtr or equal
            #05 := BLT  := branch(N xor V),         ! Branch if less than
            #06 :=                                  ! Branch if greater than
                   BGT  := branch(not (Z or (N xor V))),
            #07 := BLE  := branch(Z or (N xor V)),  ! Branch if less or equal
            #10 := BPL  := branch(not N),           ! Branch if plus
            #11 := BMI  := branch(N),               ! Branch if minus
            #12 := BHI  := branch(not (C or Z)),    ! Branch if higher
            #13 := BLOS := branch(C or Z),          ! Branch if lower or same
            #14 := BVC  := branch(not V),           ! Branch if overflow clear
            #15 := BVS  := branch(V),               ! Branch if overflow set
            #16 := BCC  := branch(not C),           ! Branch if carry clear
            #17 := BCS  := branch(C)                ! Branch if carry set
            end
        end,

    regop\register.operations :=
        begin DECODE rop =>
            begin
            0 := begin
                 IF contop eql 0 =>
                     begin
                     DECODE cpuop =>
                         begin
                         #0 := HLT(),               ! Halt
                         #1 := WAIT. := no.op(),    ! Wait for interrupt
                         #2 := RTI.RIT := no.op(),  ! Return from interrupt
                         #3 := BPT := no.op(),      ! Breakpoint trap
                         #4 := IOT := no.op(),      ! Input/output trap
                         #5 := RESET := no.op(),    ! Reset external bus
                         #6 := RTI.RIT := no.op(),  ! Return from trap
                         #7 := instr.trap(res.instr)  ! Unused opcode
                         end
                     end
                 end,
            1 := JMP(),                             ! Jump
            2 := begin
                 DECODE contop =>
                     begin
                     #0 := RTS(),                   ! Return from subroutine
                   #1:#2 := instr.trap(res.instr),
                     #3 := SPL := no.op(),          ! Set priority level
                   #4:#7 := cco := no.op()          ! Condition code ops
                     end
                 end,
            3 := SWAB()                             ! Swap bytes
            end
        end,


    classop\secondary.decode.into.classes :=
        begin
        DECODE typeop =>
            begin
            subemt(),               ! Subroutine/emulator trap
            singlop(),              ! Single operand class
            shiftop(),              ! Shift operators
            instr.trap(res.instr) ! Unused op codes
            end
        end,

    subemt\subroutine.emulator.trap.and.trap.instructions :=
        begin
        DECODE jetop =>
            begin
            0 := JSR(),             ! Jump to subroutine
            1 := begin
                 DECODE i<8> =>     ! EMT or TRAP
                     begin
                     0 := EMT(),
                     1 := TRAP()
                     end
                 end
            end
        end,
```

APPENDIX 1 *(cont'd.)*

```
        singlop\single.operand.instructions :=
            begin
            DECODE uop =>
                begin
                #0 := CLR(),            ! Clear/byte
                #1 := COM := no.op(), ! Complement/byte
                #2 := INC := no.op(), ! Increment/byte
                #3 := DEC := no.op(), ! Decrement/byte
                #4 := NEG := no.op(), ! Negate/byte
                #5 := ADC := no.op(), ! Add carry/byte
                #6 := SBC := no.op(), ! Subtract carry/byte
                #7 := TEST := no.op() ! Test/byte
                end
            end,

        shiftop\shift.instructions :=
            begin
            DECODE uop =>
                begin
                #0 := ROR(),            ! Rotate right/byte
                #1 := ROL := no.op(), ! Rotate left/byte
                #2 := ASR := no.op(), ! Arithmetic shift right/byte
                #3 := ASL := no.op(), ! Arithmetic shift left/byte
                #4 := MARK := no.op(),! Mark
                #5 := MFP := no.op(), ! Move from previous instruction
                #6 := MTP := no.op(), ! Move to previous instruction
                #7 := SXT := no.op()  ! Sign extend
                end
            end,

    MOV :=      ! Move and Move Byte
                ! MOV opcode #01, MOVB op code #11
        begin
        source() next
        dest() next
        IF regflg and byte.access =>
            begin
            source <= source<7:0>;
            byte.access = 0
            end next
        MBR = source next
        setcc(MBR, 0, MBR);
        write(byte.access)
        end,

!       . . . .                         ! Indicates instruction descriptions
!       . . . .                         ! not included in this summary

! Subroutine, Emulator Trap, and Trap instruction execution

    JSR :=      ! Jump to subroutine, JMP op code #004
        begin
        DECODE (dm @ dd) eql 0 =>
            begin
            0 := begin                                  ! False
                 dest() next
                 temp = MAR<15:0> next
                 stkref() next
                 MAR = SP; MDR = R[sr] next
                 write(byte.access) next
                 R[sr] = PC next
                 PC = temp<15:0>
                 end,
            1 := instr.trap(ill.instr)                  ! True
            end
        end,

    EMT :=      ! Emulator trap op codes, EMT op code #104000:#104377
        begin
        intvec(emt.trap); trap.instr = 1
        end,

    TRAP :=     ! Trap op codes, TRAP op code #104400:#104777
        begin
        intvec(trap.trap); trap.instr = 1
        end,

! Single operand instruction execution

    CLR :=      ! Clear and clear byte,
                ! CLR op code #0050, CLRB op code #1050
        begin
        cc = '0100 next
        dest() next
        MBR = 0 next
        write(byte.access)
        end,

!       . . . .
!       . . . .
```

```
! Jump, swab execution and registor operation decode

    JMP :=              ! Jump, JUMP op code #0001
        begin
        DECODE (dm @ dd) eql 0 =>
            begin


        0 := (dest() next PC = MAR),                ! False
        1 := instr.trap(ill.instr)                  ! True
        end
    end,



    SWAB :=             ! Swap bytes, SWAB op code #0003
        begin
        d.read(byte.access) next
        MDR = bmbr @ MBR<15:8> next
        C = V = 0; N = MBR<7>; Z = MBR<7:0> eql 0;
        IF d neq #07 => write(byte.access)
        end,



! Shift instruction execution

    ROR :=          ! Rotate right and rotate right byte,
                    ! ROR op code #0060, RORB op code #1060
        begin
        d.read(byte.access) next
        DECODE byte.access =>
            begin
            0 := (temp<16:0> = (c @ MBR) srr 1 next
                    c = temp<16>; MBR = temp<15:0>),
            1 := (temp<8:0> = (c @ bmbr) srr 1 next
                    c = temp<0>; bmbr = temp<7:0>)
            end  next
        setcc(temp, 0, temp) next
        V = N xor C next
        write(byte.access)
        end,

!       . . . .
!       . . . .

! CPU control instruction execution

    HLT :=              ! Halt, HALT op code #000000
        begin
        DECODE kernel =>
            begin
            0 := (illhlt = 1; intvec(cpu.errors)),      ! No
            1 := a = 2                                  ! Yes
            end
        end,

    RTS :=              ! Return from subroutine, RTS op code #00020
        begin
        PC = R[dr] next MAB = SP next
        read(byte.access) next
        SP = SP + 2 next
        R[dr] = MBR
        end,

!       . . . .
!       . . . .

end             ! end of description
```

Chapter 39

Implementation and Performance Evaluation of the PDP-11 Family

Edward A. Snow / Daniel P. Siewiorek

In order that methodologies useful in the design of complex systems may be developed, existing designs must be studied. The DEC PDP-11 was selected for a case study because there are a number of designs (eight are considered here), because the designs span a wide range in basic performance (7 to 1) and component technology (bipolar SSI to MOS LSI), and because the designs represent relatively complex systems.

The goals of the chapter are twofold: (1) to provide actual data about design tradeoffs and (2) to suggest design methodologies based on these data. An archetypical PDP-11 implementation is described.

Two methodologies are presented. A top-down approach uses microcycle and memory-read-pause times to account for 90 percent of the variation in processor performance. This approach can be used in initial system planning. A bottom-up approach uses relative frequency of functions to determine the impact of design tradeoffs on performance. This approach can be used in design-space exploration of a single design. Finally, the general cost/performance design tradeoffs used in the PDP-11 are summarized.

1. Introduction

As semiconductor technology has evolved, the digital systems designer has been presented with an ever-increasing set of primitive components from which to construct systems: standard SSI, MSI, and LSI, as well as custom LSI components. This expanding choice makes it more difficult to arrive at a near-optimal cost/performance ratio in a design. In the case of highly complex systems, the situation is even worse, since different primitives may be cost-effective in different subareas of such systems.

Historically, digital system design has been more of an art than a science. Good designs have evolved from a mixture of experience, intuition, and trial and error. Only rarely have design methodologies been developed (among those that have are two-level combinational logic minimization and wire-wrap routing schemes, for example). Effective design methodologies are essential for the cost-effective design of more complex systems. In addition, if the methodologies are sufficiently detailed, they can be applied in high-level design automation systems [Siewiorek and Barbacci, 1976].

Design methodologies may be developed by studying the results of the human design process. There are at least two ways to study this process. The first involves a controlled design experiment where several designers perform the same task. By contrasting the results, the range of design variation and technique can be established [Thomas and Siewiorek, 1977]. However, this approach is limited to fairly small design situations because of the redundant use of the human designers.

The second approach examines a series of existing designs that meet the same functional specification while spanning a wide range of design constraints in terms of cost, performance, etc. This paper considers the second approach and uses the DEC PDP-11[1] minicomputer line as a basis of study. The PDP-11 was selected on account of the large number of implementations (eight are considered here) with designs spanning a wide range in performance (roughly 7 to 1) and component technology (bipolar SSI, MSI, MOS custom LSI). The designs are relatively complex and seem to embody good design tradeoffs as ultimately reflected by their price/performance and commercial success.

Attention here is focused mainly upon the CPU. Memory performance enhancements such as caching are considered only insofar as they impinge upon CPU performance.

This paper is divided into three major parts. The first part (Sec. 2) provides an overview of the PDP-11 functional specification (its architecture) and serves as background for subsequent discussion of design tradeoffs. The second part (Sec. 3) presents an archetypical implementation. The last part (Secs. 4 and 5) presents methodologies for determining the impact of various design parameters on system performance. The magnitude of the impact is quantified for several parameters, and the use of the results in design situations is discussed.

2. Architectural Overview

The PDP-11 family is a set of small- to medium-scale stored-program central processors with compatible instruction sets [Bell et al., 1970]. The family evolution in terms of increased performance, constant cost, and constant performance successors is traced in Fig. 1.[2] Since the 11/45, 11/55, and 11/70 use the same processor, only the 11/45 is treated in this study.

A PDP-11 system consists of three parts: a PDP-11 processor, a collection of memories and peripherals, and a link called the Unibus over which they all communicate (Fig. 2).

A number of features, not otherwise considered here, are available as options on certain processors. These include memory management and floating-point arithmetic. The next three sub-

[1]DEC, PDP, LSI-11, Unibus, and Fastbus are registered trademarks of Digital Equipment Corporation.

[2]The original equipment manufacturer (OEM) versions of the 11/10, 11/20, and 11/40 are the 11/05, 11/15, and 11/35 respectively. The OEM machines are electrically identical (or nearly so) to their end-user counterparts, the distinction being made for marketing purposes only.

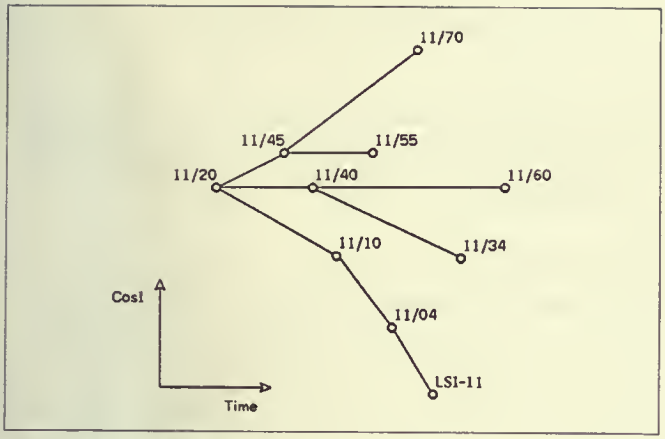

Fig. 1. PDP-11 family tree.

sections summarize the major architectural features of the PDP-11, including memory organization, processor state, addressing modes, instruction set, and Unibus protocol. The references list a number of processor handbooks and other documents which provide a more precise definition of the PDP-11 architecture than is possible here.

2.1 Memory and Processor State

The central processor contains the control logic and data paths for instruction fetching and execution. Processor instructions act upon operands located either in memory or in one of eight general registers. These operands may be either 8-bit bytes or 16-bit words.

Memory is byte- or word-addressable. Word addresses must be even. If N is a word address, then N is the byte address of the low-order byte of the word and N + 1 is the byte address of the high-order byte of the word. The control and data registers of peripheral devices are also accessed through the memory address space, and the top 4 kilowords of the space are reserved for this purpose.

The general registers are 16 bits in length and are referred to as R0 through R7. R6 is used as the system stack pointer (SP) to maintain a push-down list in memory upon which subroutine and interrupt linkages are kept. R7 is the program counter (PC) and always points to the next instruction to be fetched from memory. With minor exceptions (noted below) the SP and PC are accessible in exactly the same manner as any of the other general registers (R0 through R5).

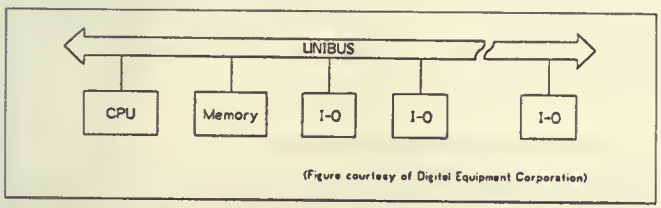

Fig. 2. Typical PDP-11 configuration.

Data-manipulation instructions fall into two categories: arithmetic instructions (which interpret their operands as 2's complement integers) and logic instructions (which interpret their operands as bit vectors). A set of condition code flags is maintained by the processor and is updated according to the sign and presence of carry/overflow from the result of any data manipulation instruction. The condition codes, processor interrupt priority, and a flag enabling program execution tracing are contained in a processor status word (PS), which is accessible as a word in the memory addressing space.

2.2 Addressing Modes and Instruction Set

The PDP-11 instruction set allows source and destination operands to be referenced via eight different addressing modes. An operand reference consists of a field specifying which of the eight modes is to be used and a second field specifying which of the eight general registers is to be used. The addressing modes are:

Mode 0 *Register.* The operand is contained in the specified register.

Mode 1 *Register deferred.* The contents of the specified register are used to address the memory location containing the operand.

Mode 2 *Autoincrement.* The contents of the specified register are used to address the memory location containing the operand, and the register is then incremented.

Mode 3 *Autoincrement deferred.* The contents of the specified register address a word in memory containing the address of the operand in memory. The specified register is incremented after the reference.

Mode 4 *Autodecrement.* The contents of the specified register are first decremented and then used to address the memory location containing the operand.

Mode 5 *Autodecrement deferred.* The contents of the specified register are first decremented and then used to address a word in memory containing the address of the operand in memory.

Mode 6 *Indexed.* The word following the instruction is fetched and added to the contents of the specified general register to form the address of the memory location containing the operand.

Mode 7 *Indexed deferred.* The word following the instruction is fetched and added to the contents of the specified general register to form the address of a word in memory containing the address of the operand in memory.

The various addressing modes simplify the manipulation of

diverse data structures such as stacks and tables. When used with the program counter these modes enable immediate operands and absolute and PC-relative addressing. The deferred modes permit indirect addressing.

The PDP-11 instruction set is made up of the following types of instructions:

Single-operand instructions. A destination operand is fetched by the CPU, modified in accordance with the instruction, and then restored to the destination.

Double-operand instructions. A source operand is fetched, followed by the destination operand. The appropriate operation is performed on the two operands and the result restored to the destination. In a few double-operand instructions, such as Exclusive OR (XOR), source mode 0 (register addressing) is implicit.

Branch instructions. The condition specified by the instruction is checked, and if it is true, a branch is taken using a field contained in the instruction as a displacement from the current instruction address.

Jumps. Jump instructions allow sequential program flow to be altered either permanently (in a jump) or temporarily (in a jump to subroutine).

Control, trap, and miscellaneous instructions. Various instructions are available for subroutine and interrupt returns, halts, etc.

Floating-point instructions. A floating-point processor is available as an option with several PDP-11 CPUs. Floating-point implementation will not be considered in this paper.

For the purpose of looking at the instruction execution cycle of the various PDP-11 processors, each cycle shall be broken into five distinct phases:[1]

Fetch. This phase consists of fetching the current instruction from memory and interpreting its opcode.

Source. This phase entails fetching the source operand for double-operand instructions from memory or a general register and loading it into the appropriate register in the data paths in preparation for the execute phase.

Destination. This phase is used to get the destination operand for single- and double-operand instructions into the data paths for manipulation in the execute phase. For JMP and JSR instructions the jump address is calculated.

Execute. During this phase the operation specified by the current instruction is performed and any result rewritten into the destination.

Service. This phase is only entered between execution of the last instruction and fetch of the next to grant a pending bus request, acknowledge an interrupt, or enter console mode after the execution of a HALT instruction or activation of the console halt key.

[1]N.B.: The instruction phase names are identical to those used by DEC; however, their application here to a state within a given machine may differ from DEC's since the intent here is to make the discussion consistent over all machines.

2.3 The Unibus

All communication among the components of a PDP-11 system takes place on a set of bidirectional lines referred to collectively as the Unibus. The LSI-11 is an exception and uses an adaptation of the Unibus. The Unibus lines carry address, data, and control signals to all memories and peripherals attached to the CPU. Transactions on the Unibus are asynchronous with the processor. At any given time there will be one device which it addresses, the addressed device becoming the bus slave. This communication may consist of data transfers or, in the case where the processor is slave, an interrupt request. The data transfers which may be initiated by the master are:

DATO	Data out—A word is transferred from master to slave.
DATOB	Data out, byte—A byte is transferred from master to slave.
DATI	Data in—A word is transferred from slave to master.
DATIP	Data in, pause—A word is transferred from slave to master and the slave awaits a transfer from master back to slave to replace the information that was read. The Unibus control allows no other data transfer to intervene between the read and the write cycles. This makes possible the reading and alteration of a memory location as an indivisible operation. In addition it permits the use of a read/modify/write cycle with core memories in place of the longer sequence of a read cycle followed by a write cycle.

3. PDP-11 Implementation

The midrange PDP-11's have comparable implementations, yet their performances vary by a factor of 7. This section discusses the features common to these implementations and the variations found between machines which provide the dimensions along which they may be characterized.

3.1 Common Implementation Features

All PDP-11 implementations can be decomposed into a set of data paths and a control unit. The data paths store and operate upon byte and word data and interface to the Unibus, which permits

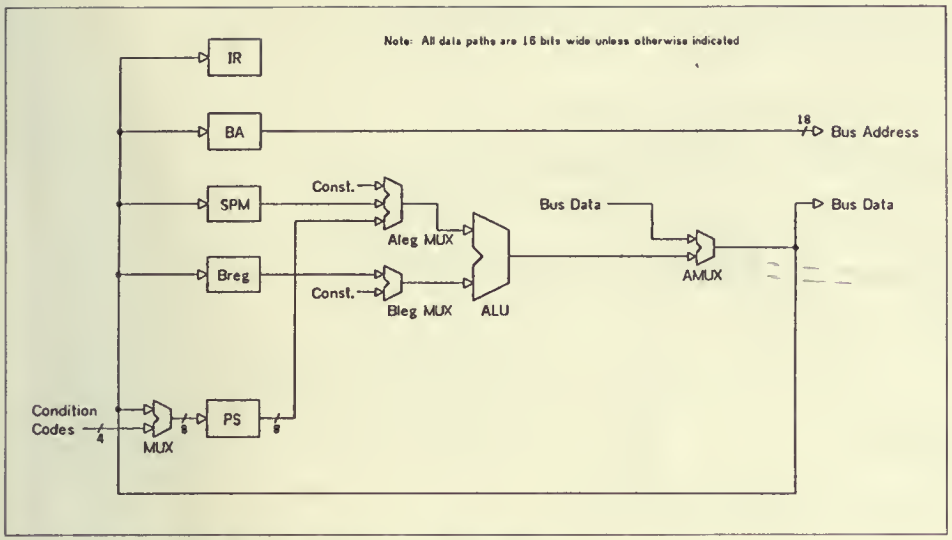

Fig. 3. Archetypical medium-range PDP-11 data paths.

them to read from and write to memory and peripheral devices. The control unit provides all the signals necessary to evoke the appropriate operations in the data paths and Unibus interface. All PDP-11's have comparable data-path and control unit implementations that allow them to be contrasted in a uniform way. In this section a basis for comparing these machines shall be established and used to characterize them.

3.1.1 Data Paths. An archetype may be constructed from which the data paths of all midrange PDP-11's differ but minimally. This archetype is diagrammed in Fig. 3. All major registers and processing elements, as well as the links and switches which interconnect them, are indicated. The data-path illustrations for individual implementations are shown in Figs. 5 through 7. These figures are laid out in a common format to encourage comparison. Note that with very few exceptions all data paths are 16 bits wide (the PDP-11 word size).

The heart of the data paths is the arithmetic logic unit or ALU, through which all data circulate and where most of the processing actually takes place. Among the operations performed by the ALU are addition, subtraction, 1's and 2's complementation, and logical ANDing and ORing.

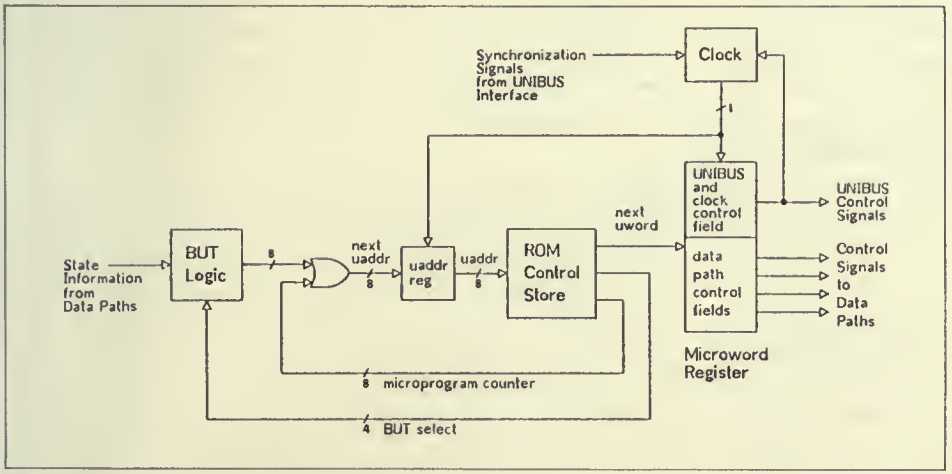

Fig. 4. Archetypical microprogrammed PDP-11 control unit.

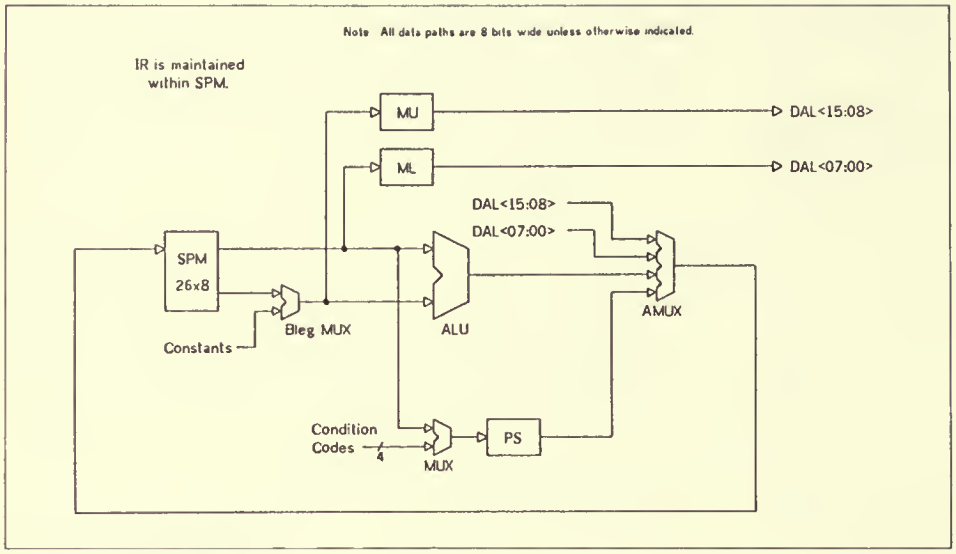

Fig. 5. LSI-11 data paths.

The inputs to the ALU are the A leg and the B leg. The A leg is normally fed from a multiplexer (Aleg MUX), which may select from an operand supplied it from the scratch-pad memory (SPM) and possibly from a small set of constants and/or the processor status register (PS). The B leg also is typically fed from its own MUX (Bleg MUX), its selections being among the B register and certain constants. In addition, the Bleg MUX may be configured so that byte selection, sign extension, and other functions may be performed on the operand which it supplies to the ALU.

Following the ALU is a multiplexer (the AMUX) typically used to select between the output of the ALU, the data lines of the Unibus, and certain constants. The output of the AMUX provides

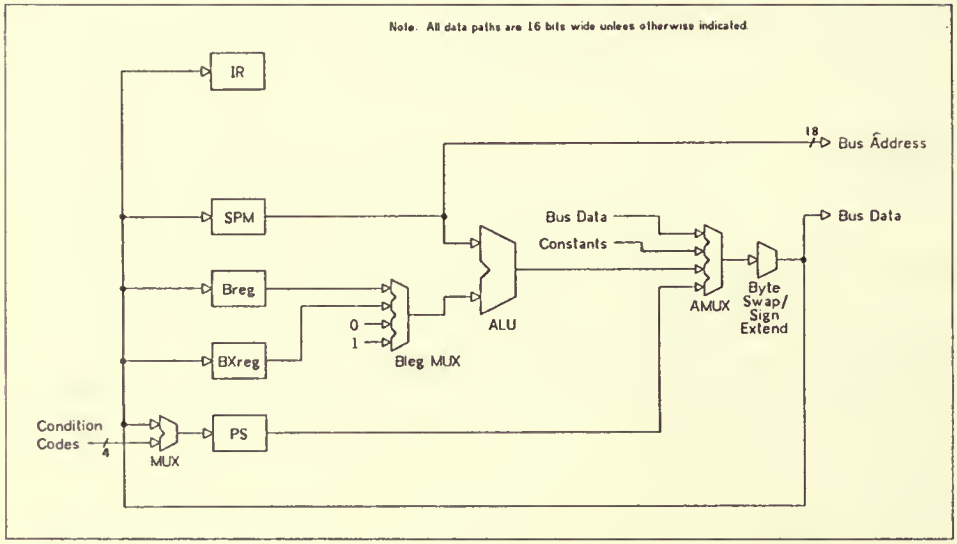

Fig. 6. PDP-11/34 data paths.

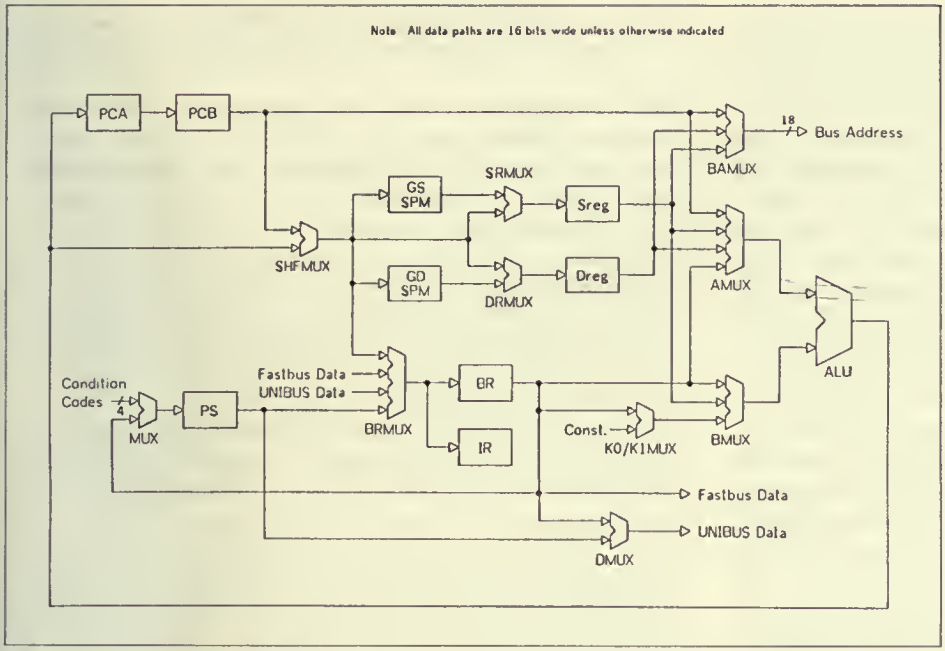

Fig. 7. PDP-11/45 data paths.

the only feedback path in all midrange PDP-11 implementations except the 11/60 and acts as an input to all major processor registers.

The internal registers lie at the beginning of the data paths. The instruction register (IR) contains the current instruction. The bus address register (BA) holds the address placed on the Unibus by the processor. The program status register (PS) contains the processor priority, memory-management-unit modes, condition code flags, and instruction trace-trap enable bit. The scratch-pad memory (SPM) is an array of 16 individually addressable registers which include the general registers (R0 to R7) plus a number of internal registers not accessible to the programmer. The B register (Breg) is used to hold the B leg operand supplied to the ALU.

The variations from this archetype are surprisingly minor. The most frequently used elements (such as the ALU and SPM) are relatively fixed in their position in the data paths from implementation to implementation. Elements which are less frequently used, and hence have less of an impact on performance, can be seen to occupy positions which vary more between implementations. Variations to be encountered include routings for the bus address and processor status register; the point of generation for certain constants; the position of the byte swapper, sign extender, and rotate/shift logic; and the use of certain auxiliary registers present in some designs and not others.

3.1.2 Control Unit. The control unit for all PDP-11 processors (with the exception of the PDP-11/20) is microprogrammed [Wilkes and Stringer, 1953]. The considerations leading to the use of this style of control implementation in the PDP-11 are discussed in O'Loughlin [1975]. The major advantage of microprogramming is flexibility in the derivation of control signals to gate register transfers, to synchronize with Unibus logic, to control microcycle timing, and to evoke changes in control flow. The way in which a microprogrammed control unit accomplishes all of these actions impacts performance.

Figure 4 represents the archetypical PDP-11 microprogrammed control unit. The contents of the microaddress register determine the current control-unit state and are used to access the next microinstruction word from the control store. Pulses from the clock generator strobe the microword and microaddress registers, loading them with the next microword and next microaddress, respectively. Repeated clock pulses thus cause the control unit to sequence through a series of states. The period spent by the control unit in one state is called a *microcycle* (or simply *cycle* when this does not lead to confusion with memory or instruction cycles), and the duration of the state as determined by the clock is known as the *cycle time*. The microword register shortens cycle time by allowing the next microword to be fetched from the control store while the current microword is being used.

Most of the fields of the microword supply signals for conditioning and clocking the data paths. Many of the fields act directly or with a small amount of decoding, supplying their signals to multiplexers and registers to select routings for data and to enable registers to shift, increment, or load on the master clock. Other fields are decoded according to the state of the data paths. An instance of this is the use of auxiliary ALU control logic to generate function-select signals for the ALU as a function of the instruction contained in the IR. Performance as determined by microcycle count is in large measure established by the connectivity of the data paths and the degree to which their functionality can be evoked by the data-path control fields of the microprogram word.

The complexity of the clock logic varies with each implementation. Typically the clock is fixed at a single period and duty cycle; however, processors such as the 11/34 and 11/40 can select from two or three different clock periods for a given cycle depending upon a field in the microword register. This can significantly improve performance in machines where the longer cycles are necessary only infrequently.

The clock logic must provide some means for synchronizing processor and Unibus operation, since the two operate asynchronously with respect to one another. Two alternate approaches are employed in midrange implementations. Interlocked operation, the simpler approach, shuts off the processor clock when a Unibus operation is initiated and turns it back on when the operation is complete. This effectively keeps microprogram flow and Unibus operation in lockstep with no overlap. Overlapped operation is a somewhat more involved approach which continues processor clocking after a DATI or DATIP is initiated. The microinstruction requiring the result of the operation has a function bit set which turns off the processor clock until the result is available. This approach makes it possible for the processor to continue running for several microcycles while a data transfer is being performed, improving performance.

The sequence of states through which the control unit passes would be fixed if it were not for the branch-on-microtest (BUT) logic. This logic generates a modifier based upon the current state of the data paths and Unibus interface (contents of the instruction register, current bus requests, etc.) and a BUT field in the microword currently being accessed from the control store, which selects the condition on which the branch is to be based. The modifier (which will be zero in the case that no branch is selected or that the condition is false) is ORed in with the next microinstruction address so that the next control-unit state is not only a function of the current state but also a function of the state of the data paths. Instruction decoding and addressing mode decoding are two prime examples of the application of BUTs. Certain code points in the BUT field do not select branch conditions, but rather provide control signals to the data paths, Unibus interface, or the control unit itself. These are known as active or working BUTs.

The JAM logic is a part of the microprogram flow-altering mechanism. This logic forces the microaddress register to a known state in the event of an exceptional condition such as a memory access error (bus timeout, stack overflow, parity error, etc.) or power-up by ORing all 1s into the next microaddress through the BUT logic. A microroutine beginning at the address of all 1s handles these trapped conditions. The old microaddress is not saved (an exception to this occurs in the case of the PDP-11/60); consequently, the interrupted microprogram sequence is lost and the microtrap ends by restarting the instruction interpretation cycle with the fetch phase.

The structure of the microprogram is determined largely by the BUTs available to implement it and by the degree to which special cases in the instruction set are exploited by these BUTs. This may have a measurable influence on performance as in the case of instruction decoding. The fetch phase of the instruction cycle is concluded by a BUT that branches to the appropriate point in the microcode based upon the contents of the instruction register. This branch can be quite complex, since it is based upon source mode for double-operand instructions, destination mode for single-operand instructions, and op code for all other types of instructions. Some processors can perform the execute phase of certain instructions (such as set/clear condition code) during the last cycle of the fetch phase; this means that the fetch or service phase for the next instruction might also be entered from BUT IRDECODE. Complicating the situation is the large number of possibilities for each phase. For instance, there are not only eight different destination addressing modes, but also subcases for each that vary for byte and word and for memory-modifying, memory-nonmodifying, MOV, and JMP/JSR instructions.

Some PDP-11 implementations such as the 11/10 make as much use of common microcode as possible to reduce the number of control states. This allows much of the IR decoding to be deferred until some later time into a microroutine which might handle a number of different cases; for instance, byte- and word-operand addressing is done by the same microroutine in a number of PDP-11s. Since the cost of control states has been dropping with the cost of control-store ROM, there has been a trend toward providing separate microroutines optimized for each special case, as in the 11/60. Thus more special cases must be broken out at the BUT IRDECODE, and so the logic to implement this BUT becomes increasingly involved. There is a payoff, though, because there are a smaller number of control states for IR decoding and fewer BUTs. Performance is boosted as well, since frequently occurring special cases such as MOV register to destination can be optimized.

4. Measuring the Effect of Design Tradeoffs on Performance

There are two alternative approaches to the problem of determining just how the particular binding of different design decisions affects the performance of each machine:

1 *Top-down approach.* Attempt to isolate the effect of a particular design tradeoff over the entire space of implementations by fitting the individual performance figures for the whole family of machines to a mathematical model which treats the design parameters as independent variables and performance as the dependent variable.

2 *Bottom-up approach.* Make a detailed sensitivity analysis of a particular tradeoff within a particular machine by comparing the performance of the machine both with and without the design feature while leaving all other design features the same.

Each approach has its assets and liabilities for assessing design tradeoffs. The first method requires no information about the implementation of a machine, but does require a sufficiently large collection of different implementations, a sufficiently small number of independent variables, and an adequate mathematical model in order to explain the variance in the dependent variable to some reasonable level of statistical confidence. The second method, on the other hand, requires a great deal of knowledge about the implementation of the given system and a correspondingly great amount of analysis to isolate the effect of the single design decision on the performance of the complete system. The information that is yielded is quite exact, but applies only to the single point chosen in the design space and may not be generalized to other points in the space unless the assumptions concerning the machine's implementation are similarly generalizable. In the following subsections the first method is used to determine the dominant tradeoffs and the second method is used to estimate the impact of individual implementation tradeoffs.

4.1 Quantifying Performance

Measuring the change in performance of a particular PDP-11 processor model due to design changes presupposes the existence of some performance metric. Average instruction execution time was chosen because of its obvious relationship to instruction-stream throughput. Neglected are such overhead factors as direct memory access, interrupt servicing, and, on the LSI-11, dynamic memory refresh. Average instruction execution times may be obtained by benchmarking or by calculation from instruction frequency and timing data. The latter method was chosen because of its freedom from the extraneous factors noted above and from the normal clock rate variations found from machine to machine of a given model. This method also allows us to calculate the change in average instruction execution time that would result from some change in the implementation. Such frequency-driven design has already been applied in practice to the PDP-11/60 [Mudge, 1977].

The instruction frequencies are tabulated in Appendix 1 and include the frequencies of the various addressing modes. These figures were calculated from measurements made by Strecker[1] on 7.6 million instruction executions traced in 10 different PDP-11 instruction streams encountered in various applications. While there is a reasonable amount of variation of frequencies from one stream to the next, the figures should be representative.

Instruction times were tabulated for each of the eight PDP-11 implementations and reported in Snow and Siewiorek [1978]. These times were calculated from the engineering documents for each machine. The times differ from those published in the PDP-11 processor handbooks for two reasons. First, in the handbooks, times have been redistributed among phases to ease the process of calculating instruction times. In Snow and Siewiorek the attempt has been made to accurately characterize each phase. Second, there are inaccuracies in the handbooks arising from conservative timing estimates and engineering revisions. The figures included here may be considered more accurate.

A performance figure is arrived at for each machine by weighting its instruction times by frequency. The results, given in Table 1, form the basis of the analyses to follow.

4.2 Analysis of Variance of PDP-11 Performance: Top-Down Approach

The first method of analysis described above will be employed in an attempt to explain most of the variance in PDP-11 performance in terms of two parameters:

1 *Microcycle time.* The microcycle time is used as a measure of processor performance which excludes the effect of the memory subsystem.

2 *Memory-read-pause time.* The memory-read-pause time is defined as the period of time during which the processor clock is suspended during a memory read. For machines with processor/Unibus overlap, the clock is assumed to be turned off by the same microinstruction which initiates the memory access. Memory-read-pause time is used as a measure of the memory subsystem's impact on processor performance. Note that this time is less than the memory access time since all PDP-11 processor clocks will continue to run at least partially concurrently with a memory access.

[1]Private communication.

Table 1 Average PDP-11 Instruction Execution Times in Microseconds

	Fetch	*Source*	*Destination*	*Execute*	*Total*	*Speed relative to LSI-11*
LSI-11	2.514	0.689	1.360	1.320	5.883	1.000
PDP-11/04	1.940	0.610	0.811	0.682	4.043	1.455
PDP-11/10	1.500	0.573	0.929	1.094	4.096	1.436
PDP-11/20	1.490	0.468	0.802	0.768	3.529	1.667
PDP-11/34	1.630	0.397	0.538	0.464	3.029	1.942
PDP-11/40	0.958	0.260	0.294	0.575	2.087	2.819
PDP-11/45 (bipolar memory)	0.363	0.101	0.213	0.185	0.863	6.820
PDP-11/60 (87% cache hit ratio)	0.541	0.185	0.218	0.635	1.578	3.727

The choice of these two factors is motivated by their dominant contribution to, and (approximately) linear relationship with, performance. Keeping the number of independent variables low is also important because of the small number of data points being fitted to the model.

The model itself is of the form:

$$t_i = k_1 c_{1i} + k_2 c_{2i}$$

where t_i = the average instruction execution time of machine i from Table 1
c_{1i} = the microcycle time of machine i (for machine with selectable microcycle times, the predominant time is used)
c_{2i} = the memory-read-pause time of machine i

This model is only an approximation, since it assumes k_1 and k_2 will be constant over all machines. In general this will not be the case. k_1 is the number of microcycles expected in a canonical instruction. This number will be a function mainly of data-path connectivity, and strictly speaking, another factor should be included to take that variability into account; however, since the data-path organizations of all PDP-11 implementations considered here (except the 11/03, 11/45, and 11/60) are quite comparable, the simplifying assumption of calling them all identical at the price of explaining somewhat less of the variance shall be made. k_2 is the number of memory accesses expected in a canonical instruction and also exhibits some variability from machine to machine. A small part of this is due to the fact that some PDP-11's actually take more memory cycles to perform a given instruction than do others (this is really only a factor in certain 11/10 instructions, notably JMP and JSR, and the 11/20 MOV instruction). A more important source of variability is the Unibus-processor overlap logic incorporated into some PDP-11 implementations, which effectively reduces the actual contribution of the $k_2 c_{2i}$ term by overlapping more memory access time with processor operation than is excluded from the memory-read-pause time.

Given the model and the dependent and independent data for each machine as given in Table 2, a linear regression was applied to determine the coefficients k_1 and k_2 and to find out how much of the variance is explained by the model.

If the regression is applied over all eight processors, $k_1 = 11.580, k_2 = 1.162$, and $R^2 = 0.904$. R^2 is the amount of variance accounted for by the model, or 90.4 percent. If the regression is applied to just the six midrange processors, $k_1 = 10.896$, $k_2 = 1.194$, and $R^2 = 0.962$. R^2 increases to 96.2 percent partly because fewer data points are being fitted to the model and partly because the LSI-11 and 11/45 can be expected to have different k coefficients from those of the midrange machines and hence do not fit the model as well. Note that if two midrange machines, the 11/04 and the 11/40, are eliminated instead of the LSI-11 and 11/45, then R^2 decreases to 89.3 percent rather than increasing. The k coefficients are close to what should be expected for average microcycle and memory cycle counts. Since k_1 is much larger than

Table 2 Top-Down Model Parameters in Microseconds

	Independent variables		*Dependent variable*
	Microcycle time	*Memory-read-pause-time*	*Average instruction execution time*
LSI-11	0.400	0.400	5.883
PDP-11/04	0.260	0.940	4.043
PDP-11/10	0.300	0.600	4.096
PDP-11/20	0.280	0.370	3.529
PDP-11/34	0.180	0.940	3.029
PDP-11/40	0.140	0.500	2.087
PDP-11/45 (bipolar memory)	0.150	0.000	0.863
PDP-11/60 (87% cache hit ratio)	0.170	0.140	1.578

k_2, average instruction time is more sensitive to microcycle time than to memory-read-pause time by a factor of k_1/k_2 or approximately 10. The implication for the designer is that much more performance can be gained or lost by perturbing the microcycle time than the memory-read-pause time.

Although this method lacks statistical rigor, it is reasonably safe to say that memory and microcycle speed do have by far the largest impact on performance and that the dependency is quantifiable to some degree.

4.3 Measuring Second-Order Effects: Bottom-up Approach

It is a great deal harder to measure the effect of other design tradeoffs on performance. The approximate methods employed in the previous section cannot be used, because the effects being measured tend to be swamped out by first-order effects and often either cancel or reinforce one another, making linear models useless. For these reasons such tradeoffs must be evaluated on a design-by-design basis as explained above. This subsection will evaluate several design tradeoffs in this way.

4.3.1 Effect of Adding a Byte Swapper to the 11/10. The PDP-11/10 uses a sequence of eight shifts to swap bytes and access odd bytes. While saving the cost of a byte swapper, this has a negative effect on performance. In this subsection the performance gained by the addition of a byte swapper either before the B register or as part of the Bleg multiplexer is calculated. Adding a byte swapper would change five different parts of the instruction interpretation process: the source and destination phases where an odd-byte operand is read from memory, the execute phase where a swap byte instruction is executed in destination mode 0 and in destination modes 1 through 7, and the execute phase where an odd-byte address is modified. In each of these cases seven fast shift cycles would be eliminated and the remaining normal-speed shift cycle could be replaced by a byte swap cycle resulting in a saving of seven fast shift cycles or 1.050 μs. None of this time would be overlapped with Unibus operations; hence, all would be saved. This saving is only effected, however, when a byte swap or odd-byte access is actually performed. The frequency with which this occurs is just the sum of the frequencies of the individual cases noted above, or 0.0640. Multiplying by the time saved per occurrence gives a saving of 0.0672 μs or 1.64 percent of the average instruction execution time. The insignificance of this saving can well be used to support the decision for leaving the byte swapper out of the PDP-11/10.

4.3.2 Effect of adding Processor/Unibus Overlap to the 11/04. Processor/Unibus overlap is not a feature of the 11/04 control unit. Adding this feature involves altering the control unit/Unibus synchronization logic so that the processor clock continues to run until a microcycle requiring the Unibus data from a DATI or DATIP is detected. A bus address register must also be added to drive the Unibus lines after the microcycle initiating the DATIP is completed. This alteration allows time to be saved in two ways. First, processor cycles may be overlapped with memory read cycles, as explained in Subsection 3.1.2. Second, since Unibus data are not read into the data paths during the cycle in which the DATIP occurs, the path from the ALU through the AMUX and back to the registers is freed. This permits certain operations to be performed in the same cycle as the DATIP; for example, the microword BA←PC; DATI; PC←PC+2 could be used to start fetching the word pointed to by the PC while simultaneously incrementing the PC to address the next word. The cycle following could then load the Unibus data directly into a scratch-pad register rather than loading the data into the Breg and then into the scratch-pad on the following cycle, as is necessary without overlap logic. A saving of two microcycle times would result.

DATI and DATIP operations are scattered liberally throughout the 11/04 microcode; however, only those cycles in which an overlap would produce a time saving need be considered. An average of 0.730 cycles can be saved or overlapped during each instruction. If all of the overlapped time is actually saved, then 0.190 μs, or 4.70 percent, will be pared from the average instruction execution time. This amounts to a 4.93 percent increase in performance.

4.3.3 Effect of Caching on the 11/60. The PDP-11/60 uses a cache to decrease its effective memory-read-pause time. The degree to which this time is reduced depends upon three factors: the cache-read–hit pause time, the cache-read–miss pause time, and the ratio of cache-read hits to total memory read accesses. A write-through cache is assumed; therefore, the timing of memory write accesses is not affected by caching and only read accesses need be considered. The performance of the 11/60 as measured by average instruction execution time is modeled exactly as a function of the above three parameters by the equation

$$t = k_1 + k_2(k_3 a + k_4[1-a])$$

where t = the average instruction execution time
a = the cache hit ratio
k_1 = the average execution time of a PDP-11/60 instruction excluding memory-read-pause time but including memory-write-pause time (1.339μs)
k_2 = the number of memory reads per average instruction (1.713)
k_3 = the memory-read-pause time for a cache hit (0.000μs)
k_4 = the memory-read-pause time for a cache miss (1.075μs)

The above equation can be rearranged to yield:

$$t = (k_1 + k_2k_4) - k_2(k_4 - k_3)a$$

The first term and the coefficient of the second term in the equation above are equivalent to 3.181 μs and 1.842 μs respectively with the given k parameter values. This reduces the average instruction time to a function of the cache hit ratio, making it possible to compare the effect of various caching schemes on 11/60 performance in terms of this one parameter.

The effect of various cache organizations on the hit ratio is described for the PDP-11 family in general in Strecker [1976b] and for the PDP-11/60 in particular in Mudge [1977]. If no cache is provided, the hit ratio is effectively 0 and the average instruction execution time reduces to the first term in the model, or 3.181 μs. A set-associative cache with a set size of 1 word and a cache size of 1,024 words has been found through simulation to give a .87 hit ratio. An average instruction time of 1.578 μs results in a 101.52 percent improvement in performance over that without the cache.

The cache organization described above is that actually employed in the 11/60. It has the virtue of being relatively simple to implement and therefore reasonably inexpensive. Set size or cache size can be increased to attain a higher hit ratio at a correspondingly higher cost. One alternative cache organization is a set size of 2 words and a cache size of 2,048 words. This organization boosts the hit ratio to .93, resulting in an instruction time of 1.468 μs, an increase in performance of 7.53 percent. This increased performance must be paid for, however, since twice as many memory chips are needed. Because the performance increment derived from the second cache organization is much smaller than that of the first while the cost increment is approximately the same, the first is more cost-effective.

4.3.4 Design Tradeoffs Affecting the Fetch Phase. The fetch phase holds much potential for performance improvement, since it consists of a single short sequence of microoperations that, as Table 1 clearly shows, involves a sizable fraction of the average instruction time because of the inevitable memory access and possible service operations. In this subsection two approaches to cutting this time are evaluated for four different processors.

The Unibus interface logic of the PDP-11/04 and that of the 11/34 are very similar. Both insert a delay into the initial microcycle of the fetch phase to allow time for bus-grant arbitration circuitry to settle so that a microbranch can be taken if a serviceable condition exists. If the arbitration logic were redesigned to eliminate this delay, the average instruction execution time would drop by 0.220 μs for the 11/04 and 0.150 μs for the 11/34.[1] The resulting increases in performance would be 5.75 percent and 5.21 percent respectively.

[1]These figures are typical. Since the delay is set by an *RC* circuit and Schmitt trigger, the delay may vary considerably from machine to machine of a given model.

Another example of a design feature affecting the fetch phase is the operand–instruction fetch overlap mechanism of the 11/40, 11/45, and 11/60. From the normal fetch times in the appendix and the actual average fetch times given in Table 1, the saving in fetch phase time alone can be calculated to be 0.162 μs for the 11/40, 0.087 μs for the 11/45, and 0.118 μs for the 11/60, or an increase of 7.77 percent, 10.07 percent, and 8.11 percent over what their respective performances would be if fetch phase time were not overlapped.

These examples demonstrate the practicality of optimizing sequences of control states that have a high frequency of occurrence rather than just those which have long durations. The 11/10 byte-swap logic is quite slow, but it is utilized infrequently, so that its impact upon performance is small; while the bus arbitration logic of the 11/34 exacts only a small time penalty but does so each time an instruction is executed and results in a larger performance impact. The usefulness of frequency data should thus be apparent, since the bottlenecks in a design are often not where intuition says they should be.

5. Summary and Use of the Methodologies

The PDP-11 offers an interesting opportunity to examine an architecture with numerous implementations spanning a wide range of price and performance. The implementations appear to fall into three distinct categories: the midrange machines (PDP-11/04/10/20/34/40/60); an inexpensive, relatively low-performance machine (LSI-11); and a comparatively expensive but high-performance machine (PDP-11/45). The midrange machines are all minor variations on a common theme with each implementation introducing much less variability than might be expected. Their differences reside in the presence or absence of certain embellishments rather than in any major structural differences. This common design scheme is still quite recognizable in the LSI-11 and even in the PDP-11/45. The deviations of the LSI-11 arise from limitations imposed by semiconductor technology rather than directly from cost or performance considerations, although the technology decision derives from cost. In the PDP-11/45, on the other hand, the quantum jump in complexity is purely motivated by the desire to squeeze the maximum performance out of the architecture.

From the overall performance model presented in Sec. 4.2 of this chapter, it is evident that instruction-stream processing can be speeded up by improving either the performance of the memory subsystem or the performance of the processor. Memory subsystem performance depends upon the number of memory accesses in a canonical instruction and the effective memory-read-pause time. There is not much that can be done about the first number, since it is a function of the architecture and thus largely fixed. The second number may be improved, however, by the use

of faster memory components or techniques such as caching.

The performance of the PDP-11 processor itself can be enhanced in two ways: by cutting the number of processor cycles to perform a given function or by cutting the time used per microcycle. Several approaches to decreasing the effective microcycle count have been demonstrated:

1 *Structure the data paths for maximum parallelism.* The PDP-11/45 can perform much more in a given microcycle than any of the midrange PDP-11's and thus needs fewer microcycles to complete an instruction. To obtain this increased functionality, however, a much more elaborate set of data paths is required in addition to a highly developed control unit to exercise them to maximum potential. Such a change is not an incremental one and involves rethinking the entire implementation.

2 *Structure the microcode to take best advantage of instruction features.* All processors except the 11/10 handle JMP/JSR addressing modes as a special case in the microcode. Most do the same for the destination modes of the MOV instruction because of its high frequency. Varying degrees of sophistication in instruction dispatching from the BUT IRDECODE at the end of every fetch is evident in different machines and results in various performance improvements.

3 *Cut effective microcycle count by overlapping processor and Unibus operation.* The PDP-11/10 demonstrates that a large microcycle count can be effectively reduced by placing cycles in parallel with memory access operations whenever possible.

Increasing microcycle speed is perhaps more generally useful, since it can often be applied without making substantial changes to an entire implementation. Several of the midrange PDP-11's achieve most of their performance improvement by increasing microcycle speed in the following ways:

1 *Make the data paths faster.* The PDP-11/34 demonstrates the improvement in microcycle time that can result from the judicious use of Schottky TTL in such heavily traveled points as the ALU. Replacing the ALU and carry/look-ahead logic alone with Schottky equivalents saves approximately 35 ns in propagation delay. With cycle times running 300 ns and less, this amounts to better than a 10 percent increase in speed.

2 *Make each microcycle take only as long as necessary.* The 11/34 and 11/40 both use selectable microcycle times to speed up cycles that do not entail long data-path propagation delays.

Circuit technology is perhaps the single most important factor in performance. It is only stating the obvious to say that doubling circuit speed will double total performance. Aside from raw speed, circuit technology dictates what it is economically feasible to build, as witnessed by the SSI PDP-11/20, the MSI PDP-11/40, and the LSI-11. Just the limitations of a particular circuit technology at a given point in time may dictate much about the design tradeoffs that can be made, as in the case of the LSI-11.

Turning to the methodologies, the two presented in Sec.4 of this chapter can be used at various times during the design cycle. The top-down approach can be used to estimate the performance of a proposed implementation, or to plan a family of implementations, given only the characteristics of the selected technology and a general estimate of data-path and memory-cycle utilization.

The bottom-up approach can be used to perturb an existing or planned design to determine the performance payoff of a particular design tradeoff. The relative frequencies of each function (e.g., addressing modes and instructions), while required for an accurate prediction, may not be available. There are, however, alternative ways to estimate relative frequencies. Consider the three following situations:

1 *At least one implementation exists.* An analysis of the implementation in typical usage (i.e., benchmark programs for a stored-program computer) can provide the relative frequencies.

2 *No implementation exists, but similar systems exist.* The frequency data may be extrapolated from measurements made on a machine with a similar architecture.

3 *No implementation exists and there are no prior similar systems.* From knowledge of the specifications, a set of most-used functions can be estimated (e.g., instruction fetch, register and relative addressing, move and add instructions for a stored-program computer). The design is then optimized for these functions.

Of course, the relative-frequency data should always be updated to take into account new data.

Our purpose in writing this chapter has been twofold: to provide data about design tradeoffs and to suggest design methodologies based on these data. It is hoped that the design data will stimulate the study of other methodologies while the results of the design methodologies presented here have demonstrated their usefulness to designers.

References

Bell et al. [1970]; Mudge [1977]; O'Loughlin [1975]; Siewiorek and Barbacci [1976]; Snow and Siewiorek [1978]; Strecker [1976b]; Thomas and Siewiorek [1977]; Wilkes and Stringer [1953]. The following Digital Equipment Corporation documents define the architecture and instruction set of the PDP-11 in addition to detailing features peculiar to individual processor implementations: DEC [1971]; DEC [1975]; DEC [1976a–e]; DEC [1977].

APPENDIX 1 INSTRUCTION TIME COMPONENT FREQUENCIES

This appendix tabulates the frequencies of PDP-11 instructions and addressing modes. These data were derived as explained in Subsection 4.1. Frequencies are given for the occurrence of each phase (e.g., source, which occurs only during double-operand instructions), each subcase of each phase (e.g., jump destination, which occurs only during jump or jump to subroutine instructions), and each instance of each phase, such as a particular addressing mode or instruction. The frequency with which the phase is skipped is listed for source and destination phases. Source and destination odd-byte–addressing frequencies are listed as well because of their effect on instruction timing.

	Frequency
Fetch	1.0000
Source	0.4069
Mode	
0 R	0.1377
1 @R or (R)	0.0338
2 (R)+	0.1587
3 @(R)+	0.0122
4 −(R)	0.0352
5 @−(R)	0.0000
6 X(R)	0.0271
7 @X(R)	0.0022
No Source	0.5931
NOTE: Frequency of odd-byte addressing (SM1–7) = 0.0252.	
Destination Mode	0.6872
Data Manipulation	0.6355
Mode	
0 R	0.3146
1 @R or R	0.0599
2 (R)+	0.0854
3 @(R)+	0.0307
4 −(R)	0.0823
5 @−(R)	0.0000
6 X(R)	0.0547
7 @X(R)	0.0080
NOTE: Frequency of odd-byte addressing (DM1–7) = 0.0213.	
Jump (JMP/JSR)	0.0517
Operand Mode	
0 R	0.0000 (ILLEGAL)
1 @R or (R)	0.0000
2 (R)+	0.0000
3 @(R)+	0.0079
4 −(R)	0.0000
5 @−(R)	0.0000
6 X(R)	0.0438
7 @X(R)	0.0000
No Destination	0.3128

	Frequency
Execute Instruction	1.0000
Double operand	0.4069
ADD	0.0524
SUB	0.0274
BIC	0.0309
BICB	0.0000
BIS	0.0012
BISB	0.0013
CMP	0.0626
CMPB	0.0212
BIT	0.0041
BITB	0.0014
MOV	0.1517
MOVB	0.0524
XOR	0.0000
Single operand	0.2286
CLR	0.0186
CLRB	0.0018
COM	0.0000
COMB	0.0000
INC	0.0224
INCB	0.0000
DEC	0.0809
DECB	0.0000
NEG	0.0038
NEGB	0.0000
ADC	0.0070
ADCB	0.0000
SBC	0.0000
SBCB	0.0000
ROR	0.0036
RORB	0.0000
ROL	0.0059
ROLB	0.0000
ASR	0.0069
ASRB	0.0000
ASL	0.0298
ASLB	0.0000
TST	0.0329
TSTB	0.0079
SWAB	0.0038
SXT	0.0000
Branch	0.2853
All branches (true)	0.1744
All branches (false)	0.1109
SOB (true)	0.0000
SOB (false)	0.0000

	Frequency
Jump	0.0517
JMP	0.0272
JSR	0.0245
Control, trap and miscellaneous	0.0270
Set/clear condition codes	0.0017
MARK	0.0000
RTS	0.0236
RTI	0.0000
RTT	0.0000
IOT	0.0000
EMT	0.0017
TRAP	0.0000
BPT	0.0000

NOTES: Frequency of destination odd-byte addressing (DM1-7) = 0.0213.
Execution frequencies indicated as 0.0000 have an aggregate frequency < 0.0050.

Chapter 47

The Evolution of the PDP-11[1]

C. G. Bell / J. C. Mudge

In the original 1970 PDP-11 paper (Chap. 38), a set of design goals and constraints were given, beginning with a discussion of the weaknesses frequently found in minicomputers. The designers of the PDP-11 faced each of these known minicomputer weaknesses, and their goals included a solution to each one. This section reviews the original goals, commenting on the success or failure of the PDP-11 in meeting each of them.

The weaknesses of prior designs that were noted were limited addressability, a small number of registers, absence of hardware stack facilities, elementary I/O processing, absence of growth-path family members, and high programming costs.

The first weakness of minicomputers was their limited addressing capability. The biggest (and most common) mistake that can be made in a computer design is that of not providing enough address bits for memory addressing and management. The PDP-11 followed this hallowed tradition of skimping on address bits, but it was saved by the principle that a good design can evolve through at least one major change.

For the PDP-11, the limited address problem was solved for the short run, but not with enough finesse to support a large family of minicomputers. That was indeed a costly oversight, resulting in both redundant development and lost sales. It is extremely embarassing that the PDP-11 had to be redesigned with memory management[2] only two years after writing the paper that outlined the goal of providing increased address space. All earlier DEC designs suffered from the same problem, and only the PDP-10 evolved over a long period (15 years) before a change occurred to increase its address space. In retrospect, it is clear that another address bit is required every two or three years, since memory prices decline about 30 percent yearly, and users tend to buy constant price successor systems.

A second weakness of minicomputers was their tendency to skimp on registers. This was corrected for the PDP-11 by providing eight 16-bit registers. Later, six 64-bit registers were added as the accumulators for floating-point arithmetic. This number seems to be adequate: there are enough registers to allocate two or three registers (beyond those already dedicated to program counter and stack pointer) for program global purposes and still have registers for local statement computation.[3] More registers would increase the context switch time and worsen the register allocation problem for the user.

A third weakness of minicomputers was their lack of hardware stack capability. In the PDP-11, this was solved with the autoincrement/autodecrement addressing mechanism. This solution is unique to the PDP-11, has proved to be exceptionally useful, and has been copied by other designers. The stack limit check, however, has not been widely used by DEC operating systems.

A fourth weakness, limited interrupt capability and slow context switching, was essentially solved by the Unibus interrupt vector design. The basic mechanism is very fast, requiring only four memory cycles from the time an interrupt request is issued until the first instruction of the interrupt routine begins execution. Implementations could go further and save the general registers, for example, in memory or in special registers. This was not specified in the architecture and has not been done in any of the implementations to date. VAX-11 provides explicit load and save process context instructions.

A fifth weakness of earlier minicomputers, inadequate character handling capability, was met in the PDP-11 by providing direct byte addressing capability. String instructions were not provided in the hardware, but the common string operations (move, compare, concatenate) could be programmed with very short loops. Early benchmarks showed that this mechanism was adequate. However, as COBOL compilers have improved and as more understanding of operating systems string handling has been obtained, a need for a string instruction set was felt, and in 1977 such a set was added.

A sixth weakness, the inability to use read-only memories as primary memory, was avoided in the PDP-11. Most code written for the PDP-11 tends to be reentrant without special effort by the programmer, allowing a read-only memory (ROM) to be used directly. Read-only memories are used extensively for bootstrap loaders, program debuggers, and for simple functions. Because large read-only memories were not available at the time of the original design, there are no architectural components designed specifically with large ROMs in mind.

A seventh weakness, one common to many minicomputers, was primitive I/O capabilities. The PDP-11 answers this to a certain extent with its improved interrupt structure, but the completely general solution of I/O computers has not yet been implemented. The I/O processor concept is used extensively in display processors, in communication processors, and in signal processing.

[1]Excerpted from C. G. Bell, J. C. Mudge, and J. E. McNamara, *Computer Engineering: A DEC View of Hardware Systems Design*, Digital Press, Maynard, Mass., 1978, pp. 379–408.

[2]The memory management served two other functions besides expanding the 16-bit processor-generated addresses into 18-bit Unibus addresses: program relocation and protection.

[3]Since dedicated registers are used for each Commercial Instruction Set (CIS) instruction, this was no longer true when CIS was added.

Having a single machine instruction that transmits a block of data at the interrupt level would decrease the central processor overhead per character by a factor of 3; it should have been added to the PDP-11 instruction set for implementation on all machines. Provision was made in the 11/60 for invocation of a micro-level interrupt service routine in writable control store (WCS), but the family architecture is yet to be extended in this direction.

Another common minicomputer weakness was the lack of system range. If a user had a system running on a minicomputer and wanted to expand it or produce a cheaper turnkey version, he frequently had no recourse, since there were often no larger and smaller models with the same architecture. The PDP-11 has been very successful in meeting this goal.

A ninth weakness of minicomputers was the high cost of programming caused by programming in lower level languages. Many users programmed in assembly language, without the comfortable environment of high-level languages, editors, file systems, and debuggers available on bigger systems. The PDP-11 does not seem to have overcome this weakness, although it appears that more complex systems are being successfully built with the PDP-11 than with its predecessors, the PDP-8 and the PDP-15. Some systems programming is done using higher level languages; however, the optimizing compiler for BLISS-11 at first ran only on the PDP-10. The use of BLISS has been slowly gaining acceptance. It was first used in implementing the FORTRAN-IV PLUS (optimizing) compiler. Its use in PDP-10 and VAX-11 systems programming has been more widespread.

One design constraint that turned out to be expensive, but worth it in the long run, was the necessity for the word length to be a multiple of eight bits. Previous DEC designs were oriented toward 6-bit characters, and DEC had a large investment In 12-, 18-, and 36-bit systems.

Microprogrammability was not an explicit design goal, partially because fast, large, and inexpensive read-only memories were not available at the time of the first implementation. All subsequent machines have been microprogrammed, but with some difficulty because some parts of the instruction set processor, such as condition code setting and instruction register decoding, are not ideally matched to microprogrammed control.

The design goal of understandability seems to have received little attention. The PDP-11 was initially a hard machine to understand and was marketable only to those with extensive computer experience. The first programmers' handbook was not very helpful. It is still unclear whether a user without programming experience can learn the machine solely from the handbook. Fortunately, several computer science textbooks [Gear, 1974; Eckhouse, 1975; Stone and Siewiorek, 1975] and other training books have been written based on the PDP-11.

Structural flexibility (modularity) for hardware configurations was an important goal. This succeeded beyond expectations and is discussed extensively in the Unibus Cost and Performance section.

Evolution of the Instruction Set Processor

Designing the instruction set processor level of a machine—that collection of characteristics such as the set of data operators, addressing modes, trap and interrupt sequences, register organization, and other features visible to a programmer of the bare machine—is an extremely difficult problem. One has to consider the performance (and price) ranges of the machine family as well as the intended applications, and difficult tradeoffs must be made. For example, a wide performance range argues for different encodings over the range; for small systems a byte-oriented approach with small addresses is optimal, whereas larger systems require more operation codes, more registers, and larger addresses. Thus, for larger machines, instruction coding efficiency can be traded for performance.

The PDP-11 was originally conceived as a small machine, but over time its range was gradually extended so that there is now a factor of 500 in price ($500 to $250,000) and memory size (8 Kbytes to 4 Mbytes[1]) between the smallest and largest models. This range compares favorably with the range of the IBM System 360 family (4 Kbytes to 4 Mbytes). Needless to say, a number of problems have arisen as the basic design was extended.

Chronology of the Extensions

A chronology of the extensions is given in Table 1. Two major extensions, the memory management and the floating point, occurred with the 11/45. The most recent extension is the Commercial Instruction Set, which was defined to enhance performance for the character string and decimal arithmetic data-types of the commercial languages (e.g., COBOL). It introduced the following to the PDP-11 architecture:

1. Data-types representing character sets, character strings, packed decimal strings, and zoned decimal strings.
2. Strings of variable length up to 65 Kcharacters.
3. Instructions for processing character strings in each data-type (move, add, subtract, multiply, divide).
4. Instructions for converting among binary integers, packed decimal strings, and zoned decimal strings.
5. Instructions to move the descriptors for the variable length strings.

The initial design did not have enough operation code space to

[1]Although 22 bits are used, only 2 megabytes can be utilized in the 11/70.

Table 1 Chronology of PDP-11 Instruction Set Processor (ISP) Evolution

Model(s)	*Evolution*
11/20	Base ISP (16-bit virtual address) and PMS (16-bit processor physical memory address) Unibus with 18-bit addressing
11/20	Extended Arithmetic Element (hardware multiply/divide)
11/45 (11/55,11/70, 11/60,11/34)	Floating-point instruction set with 6 additional registers (46 instructions) in the Floating-Point Processor
11/45 (11/55,11/70)	Memory management (KT11C). 3 modes of protection (Kernel, Supervisor, User); 18-bit processor physical addressing; 16-bit virtual addressing in 8 segments for both instruction and data spaces
11/45 (11/55,11/70)	Extensions for second set of general registers and program interrupt request
11/40 (11/03)	Extended Instruction Set for multiply/divide; floating-point instruction set (4 instructions)
11/40 (11/34,11/60)	Memory Management (KT11D), 2 modes of protection (Kernel, User); 18-bit processor physical addressing; 16-bit virtual addressing in 8 segments
11/70	22-bit processor physical addressing; Unibus map for peripheral controller 22-bit addressing
11/70 (11/60)	Error register accessibility for on-line diagnosis and retry (e.g., cache parity error)
11/03 (11/04,11/34)	Program access to processor status register via explicit instruction (versus Unibus address)
11/03	One level program interrupt
11/60	Extended Function Code for invocation of user-written microcode
VAX-11/780	VAX architectural extensions for 32-bit virtual addressing VAX ISP
11/03	Commercial Instruction Set (CIS)
11/70mP	Interprocessor Interrupt and System Timers for multiprocessor

accommodate instructions for new data-types. Ideally, the complete set of operation codes should have been specified at initial design time so that extensions would fit. With this approach, the uninterpreted operation codes could have been used to call the various operation functions, such as a floating-point addition. This would have avoided the proliferation of run-time support systems for the various hardware/software floating-point arithmetic methods (Extended Arithmetic Element, Extended Instruction Set, Floating Instruction Set, Floating-Point Processor). The extracode technique was used in the Atlas and Scientific Data Systems (SDS) designs, but these techniques are overlooked by most computer designers. Because the complete instruction set processor (or at least an extension framework) was unspecified in the initial design, completeness and orthogonality have been sacrificed.

At the time the PDP-11/45 was designed, several operation code extension schemes were examined: an escape mode to add the floating-point operations, bringing the PDP-11 back to being a more conventional general register machine by reducing the number of addressing modes, and finally, typing the data by adding a global mode that could be switched to select floating point instead of byte operations for the same operation codes. The floating-point instruction set, introduced with the 11/45, is a version of the second alternative.

It also became necessary to do something about the small address space of the processor. The Unibus limits the physical memory to the 262,144 bytes addressable by 18-bits. In the PDP-11/70, the physical address was extended to 4 Mbytes by providing a Unibus map so that devices in a 256 Kbyte Unibus space could transfer into the 4-Mbyte space via mapping registers. While the physical address limits are acceptable for both the Unibus and larger systems, the address for a single program is still confined to an instantaneous space of 16 bits, the user virtual address. The main method of dealing with relatively small addresses is via process-oriented operating systems that handle many small tasks. This is a trend in operating systems, especially for process control and transaction processing. It does, however, enforce a structuring discipline in (user) program organization. The RSX-11 series of operating systems for the PDP-11 are organized this way, and the need for large addresses is lessened.

The initial memory management proposal to extend the virtual memory was predicated on dynamic, rather than static, assignment of memory segment registers. In the current memory management scheme, the address registers are usually considered to be static for a task (although some operating systems provide functions to get additional segments dynamically).

With dynamic assignment, a user can address a number of segment names, via a table, and directly load the appropriate segment registers. The segment registers act to concatenate additional address bits in a base address fashion. There have been other schemes proposed that extend the addresses by extending the length of the general registers—of course, extended addresses propagate throughout the design and include double length address variables. In effect, the extended part is loaded with a base address.

With larger machines and process-oriented operating systems, the context switching time becomes an important performance factor. By providing additional registers for more processes, the time (overhead) to switch context from one process (task) to another can be reduced. This option has not been used in the operating system implementations of the PDP-11s to date, although the 11/45 extensions included a second set of general registers. Various alternatives have been suggested, and to accomplish this effectively requires additional operators to handle the many aspects of process scheduling. This extension appears to be relatively unimportant since the range of computers coupled with networks tends to alleviate the need by increasing the real parallelism (as opposed to the apparent parallelism) by having various independent processors work on the separate processes in parallel. The extensions of the PDP-11 for better control of I/O devices is clearly more important in terms of improved performance.

Architecture Management

In retrospect, many of the problems associated with PDP-11 evolution were due to the lack of an ongoing architecture management function. The notion of planned evolution was very strong at the beginning. However, a formal architecture control function was not set up until early in 1974. In some sense this was already too late—the four PDP-11 models designed by that date (11/20, 11/05, 11/40, 11/45) had incompatibilities between them. The architecture control function since then has ensured that no further divergence (except in the LSI-11) took place in subsequent models, and in fact resulted in some convergence. At the time the Commercial Instruction Set was added, an architecture extension framework was adopted. Insufficient encodings existed to provide a large number of additional instructions using the same encoding style (in the same space) as the basic PDP-11, i.e., the operation code and operand specifier addressing mode specifiers within a single 16-bit word. An instruction extension framework was adopted which utilized a full word as the opcode, with operand addressing mode specifiers in succeeding instruction stream words along the lines of VAX-11. This architectural extension permits 512 additional opcodes, and instructions may have an unlimited number of operand addressing mode specifiers. The architecture control function also had to deal with the Unibus address space problem.

With VAX-11, architecture management has been in place since the beginning. A definition of the architecutre was placed under formal change control well before the VAX-11/780 was built, and both hardware and software engineering groups worked with the same document. Another significant difference is that an extension framework was defined in the original architecture.

An Evaluation

The criteria used to decide whether or not to include a particular capability in an instruction set are highly variable and border on the artistic. Critics ask that the machine appear elegant, where elegance is a combined quality of instruction formats relating to mnemonic significance, operator/data-type completeness and orthogonality, and addressing consistency. Having completely general facilities (e.g., registers) which are not context dependent assists in minimizing the number of instruction types and in increasing understandability (and usefulness). The authors feel that the PDP-11 has provided this.

At the time the Unibus was designed, it was felt that allowing 4 Kbytes of the address space for I/O control registers was more than enough. However, so many different devices have been interfaced to the bus over the years that it is no longer possible to assign unique addresses to every device. The architectural group has thus been saddled with the chore of device address bookkeeping. Many solutions have been proposed, but none was soon enough; as a result, they are all so costly that it is cheaper just to live with the problem and the attendant inconvenience.

Techniques for generating code by the human and compiler vary widely and thus affect instruction set processor design. The PDP-11 provides more addressing modes than nearly any other computer. The eight modes for source and destination with dyadic operators provide what amounts to 64 possible ADD instructions. By associating the Program Counter and Stack Pointer registers with the modes, even more data accessing methods are provided. For example, 18 varieties of the MOVE instruction can be distinguished as the machine is used in two-address, general register, and stack machine program forms. (There is a price for this generality—namely, fewer bits could have been used to encode the address modes that are actually used most of the time.)

How the PDP-11 Is Used

In general, the PDP-11 has been used mostly as a general register (i.e., memory to registers) machine. This can be seen by observing the use frequency from Strecker's data (see Appendix 1 in Chap. 39). In one case, it was observed that a user who previously used a one-accumulator computer (e.g., PDP-8), continued to do so. A general register machine provides the greatest performance, and the cost (in terms of bits) is the same as when used as a stack machine. Some compilers, particularly the early ones, are stack oriented since the code production is easier. In principle, and with much care, a fast stack machine could be constructed. However, since most stack machines use primary memory for the stack, there is a loss of performance even if the top of the stack is cached. While a stack is the natural (and necessary) structure to interpret the nested block structure languages, it does

not necessarily follow that the interpretation of all statements should occur in the context of the stack. In particular, the predominance of register transfer statements are of the simple 2- and 3-address forms:

$$D \leftarrow S$$

and

$$D1(\text{index } 1) \leftarrow f(S2(\text{index } 2), S3(\text{index } 3))$$

These do not require the stack organization. In effect, appropriate assignment allows a general register machine to be used as a stack machine for most cases of expression evaluation. This has the advantage of providing temporary, random access to common subexpressions, a capability that is usually hard to exploit in stack architectures.

The Evolution of the PMS (Modular) Structure

The end product of the PDP-11 design is the computer itself, and in the evolution of the architecture one can see images of the evolution of ideas. In this section, the architectural evolution is outlined, with a special emphasis on the Unibus.

The Unibus is the architectural component that connects together all of the other major components. It is the vehicle over which data flow between pairs of components takes place.

In general, the Unibus has met all expectations. Several hundred types of memories and peripherals have been interfaced to it; it has become a standard architectural component of systems in the $3K to $100K price range (1975). The Unibus does limit the performance of the fastest machines and penalizes the lower performance machines with a higher cost. Recently it has become clear that the Unibus is adequate for large, high performance systems when a cache structure is used because the cache reduces the traffic between primary memory and the central processor since about one-tenth of the memory references are outside the cache. For still larger systems, supplementary buses were added for central processor to primary memory and primary memory to secondary memory traffic. For very small systems like the LSI-11, a narrower bus was designed.

The Unibus, as a standard, has provided an architectural component for easily configuring systems. Any company, not just DEC, can easily build components that interface to the bus. Good buses make good engineering neighbors, since people can concentrate on structured design. Indeed, the Unibus has created a secondary industry providing alternative sources of supply for memories and peripherals. With the exception of the IBM 360 Multiplexer-Selector Bus, the Unibus is the most widely used computer interconnection standard.

The Unibus has also turned out to be invaluable as an "umbilical cord" for factory diagnostic and checkout procedures. Although such a capability was not part of the original design, the Unibus is almost capable of controlling the system components (e.g., processor and memory) during factory checkout. Ideally, the scheme would let all registers be accessed during full operation. This is possible for all devices except the processor. By having all central processor registers available for reading and writing in the same way that they are available from the console switches, a second system can fully monitor the computer under test.

In most recent PDP-11 models, a serial communications line, called the ASCII Console, is connected to the console, so that a program may remotely examine or change any information that a human operator could examine or change from the front panel, even when the system is not running. In this way computers can be diagnosed from a remote site.

Difficulties with the Design

The Unibus design is not without problems. Although two of the bus bits were set aside in the original design as parity bits, they have not been widely used as such. Memory parity was implemented directly in the memory; this phenomenon is a good example of the sorts of problems encountered in engineering optimization. The trading of bus parity for memory parity exchanged higher hardware cost and decreased performance for decreased service cost and better data integrity. Because engineers are usually judged on how well they achieve production cost goals, parity transmission is an obvious choice to pare from a design, since it increases the cost and decreases the performance. As logic costs decrease and pressure to include warranty costs as part of the product design cost increases, the decision to transmit parity may be reconsidered.

Early attempts to build tightly coupled multiprocessor or multicomputer structures (by mapping the address space of one Unibus onto the memory of another), called Unibus windows, were beset with a logic deadlock problem. The Unibus design does not allow more than one master at a time. Successful multiprocessors required much more sophisticated sharing mechanisms such as shared primary memory.

Unibus Cost and Performance

Although performance is always a design goal, so is low cost; the two goals conflict directly. The Unibus has turned out to be nearly optimum over a wide range of products. It served as an adequate memory-processor interconnect for six of the ten models. However, in the smallest system, DEC introduced the LSI-11 Bus, which uses about half the number of conductors. For the largest systems, a separate 32-bit data path is used between processor and memory, although the Unibus is still used for communication with the majority of the I/O controllers (the slower ones).

The bandwidth of the Unibus is approximately 1.7 megabytes per second or 850 K transfers/second. Only for the largest configurations, using many I/O devices with very high data rates, is this capacity exceeded. For most configurations, the demand put on an I/O bus is limited by the rotational delay and head positioning of disks and the rate at which programs (user and system) issue I/O requests.

An experiment to further the understanding of Unibus capacity and the demand placed against it was carried out. The experiment used a synthetic workload; like all synthetic workloads, it can be challenged as not being representative. However, it was generally agreed that it was a heavy I/O load. The load simulated transaction processing, swapping, and background computing in the configuration shown in Fig. 1. The load was run on five systems, each placing a different demand on the Unibus.

Each run produced two numbers: (1) the time to complete 2,000 transactions, and (2) the number of iterations of a program called HANOI that were completed.

System	*Benchmark time (minutes)**	*Number of HANOI iterations*
11/60 cache on	15	12
11/60 cache off	15	2
11/40	15	3
11/70 MBCBUS	15	23
11/70 Unibus	26	38

*2,000 transactions plus swapping plus HANOI.

The results were interpreted as follows:

1 I/O throughput. For this workload the Unibus bandwidth was adequate. For systems 1 through 4, the I/O activity took the same amount of time.

2 11/70 Unibus. The run on this system (no use was made of the 32-bit wide processor/memory bus) took longer because of the retries caused by data lates (approximately 19,000) on the moving head disk (RP04). The extra time taken for the benchmark allowed more iterations of HANOI to occur. The PDP-11/70 Unibus had a bandwidth of about 1 megabyte. It was less than the usual Unibus (about 1.7 megabyte) because of the map delay (100 nanoseconds), the cache cycle (240 nanoseconds), and the main memory bus redriving and synchronization.

3 11/60 Cache. Systems 1 and 2 clearly show the effectiveness of a cache. Most memory references for HANOI were to the cache and did not involve the Unibus, which was the PDP-11/60s I/O Bus. Systems 2 and 3 were essentially equivalent, as expected. There are two reasons for the 11/40 having slightly more compute bandwidth than an 11/60 with its cache off. First, the 11/40 memory is faster than the 11/60 backing store, and second, the 11/40 processor relinquishes the Unibus for a direct memory access cycle; the 11/60 processor must request the Unibus for a processor cycle.

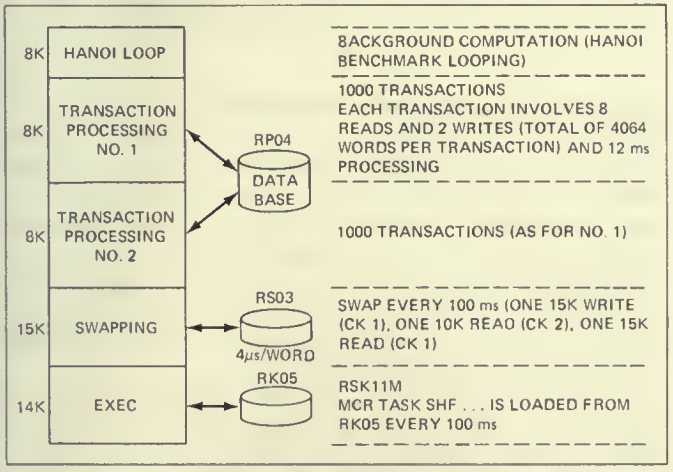

Fig. 1. The synthetic workload used to measure Unibus capacity.

There are several attributes of a bus that affect its cost and performance. One factor affecting performance is simply the data rate of a single conductor. There is a direct tradeoff involving cost, performance, and reliability. Shannon [1948] gives a relationship between the fundamental signal bandwidth of a link and the error rate (signal-to-noise ratio) and data rate. The performance and cost of a bus are also affected by its length. Longer cables cost proportionately more, since they require more complex circuitry to drive the bus.

Since a single-conductor link has a fixed data rate, the number of conductors affects the net speed of a bus. However, the cost of a bus is directly proportional to the number of conductors. For a given number of wires, time domain multiplexing and data encoding can be used to trade performance and logic complexity. Since logic technology is advancing faster than wiring technology, it seems likely that fewer conductors will be used in all future systems, except where the performance penalty of time domain multiplexing is unacceptably great.

If, during the original design of the Unibus, DEC designers could have foreseen the wide range of applications to which it would be applied, its design would have been different. Individual controllers might have been reduced in complexity by more central control. For the largest and smallest systems, it would have been useful to have a bus that could be contracted or expanded by multiplexing or expanding the number of conductors.

The cost-effectiveness of the Unibus is due in large part to the high correlation between memory size, number of address bits,

I/O traffic, and processor speed. Gene Amdahl's rule of thumb for IBM computers is that 1 byte of memory and 1 bit/sec of I/O are required for each instruction/sec. For traditional DEC applications, with emphasis in the scientific and control applications, there is more computation required per memory word. Further, the PDP-11 instruction sets do not contain the extensive commercial instructions (character strings) typical of IBM computers, so a large number of instructions must be executed to accomplish the same task. Hence, for DEC computers, it is better to assume 1 byte of memory for each 2 instructions/sec, and that 1 byte/sec of I/O occurs for each instruction/sec.

In the PDP-11, an average instruction accesses 3-5 bytes of memory, so assuming 1 byte of I/O for each instruction/sec, there are 4–6 bytes of memory accessed on the average for each instruction/sec. Therefore, a bus that can support 2 megabytes/sec of traffic permits instruction execution rates of 0.33–0.5 megainstructions/sec. This implies memory sizes of 0.16–0.25 megabytes, which matches well with the maximum allowable memory of 0.064–0.256 megabytes. By using a cache memory on the processor, the effective memory processor rate can be increased to balance the system further. If fast floating-point instructions were added to the instruction set, the balance might approach that used by IBM and thereby require more memory (an effect seen in the PDP-11/70).

The task of I/O is to provide for the transfer of data from peripheral to primary memory where it can be operated on by a program in a processor. The peripherals are generally slow, inherently asynchronous, and more error-prone than the processors to which they are attached.

Historically, I/O transfer mechanisms have evolved through the following four stages:

1 **Direct sequential I/O under central processor control.** An instruction in the processor causes a data transfer to take place with a device. The processor does not resume operation until the transfer is complete. Typically, the device control may share the logic of the processor. The first input/output transfer (IOT) instruction in the PDP-1 is an example: the IOT effects transfer between the Accumulator and a selected device. Direct I/O simplifies programming because every operation is sequential.

2 **Fixed buffer, 1-instruction controllers.** An instruction in the central processor causes a data transfer (of a word or vector), but in this case, it is to a buffer of the simple controller and thus at a speed matching that of the processor. After the high speed transfer has occurred, the processor continues while an asynchronous, slower transfer occurs between the buffer and the device. Communication back to the processor is via the program interrupt mechanism. A single instruction to a simple controller can also cause a complete block (vector) of data to be transmitted between memory and the peripheral. In this case, the transfer takes place via the direct memory access (DMA) link.

3 **Separate I/O processors—the channel.** An independent I/O processor with a unique ISP controls the flow of data between primary memory and the peripheral. The structure is that of the multiprocessor, and the I/O control program for the device is held in primary memory. The central processor informs the I/O processor about the I/O program location.

4 **I/O computer.** This mechanism is also asynchronous with the central processor, but the I/O computer has a private memory which holds the I/O program. Recently, DEC communications options have been built with embedded control programs. The first example of an I/O computer was in the CDC 6600 (1964).

The authors believe that the single-instruction controller is superior to the I/O processor as embodied in the IBM Channel mainly because the latter concept has not gone far enough. Channels are costly to implement, sufficiently complex to require their own programming environment, and yet not quite powerful enough to assume the processing, such as file management, that one would like to offload from the processor. Although the I/O traffic does require central processor resources, the addition of a second, general purpose central processor is more cost-effective than using a central processor-I/O processor or central processor-multiple I/O processor structure. Future I/O systems will be message-oriented, and the various I/O control functions (including diagnostics and file management) will migrate to the subsystem. When the I/O computer is an exact duplicate of the central processor, not only is there an economy from the reduced number of part types but also the same programming environment can be used for I/O software development and main program development. Notice that the I/O computer must implement precisely the same set of functions as the processor doing direct I/O.

Technology: Components of the Design

Computers are strongly influenced by the basic electronic technology of their components. The PDP-11 Family provides an extensive example of designing with improved technologies. Because design resources have been available to do concurrent implementations spanning a cost/performance range, PDP-11s offer a rich source of examples of the three different design styles: constant cost with increasing functionality, constant functionality with decreasing cost, and growth path.

Memory technology has had a much greater impact on PDP-11 evolution than logic technology. Except for the LSI-11, the one logic family (7400 series TTL) has dominated PDP-11 implementations since the beginning. Except for a small increase after the

Table 2 Characteristics of PDP-11 Models with Techniques Used to Span Cost and Performance Range

		Performance			*Range-spanning techniques*		
Model	*First shipment*	*Basic instructions per second (relative to PDP-11/03)*	*Floating-point arithmetic (whetstone instructions per second)*	*Memory range (Kbytes)*	*For high-performance*	*For low cost*	*Notable attributes*
11/03 (LSI-11)	6/75	1	26	8-56		8 bit wide datapath; LSI-11 Bus; tailored PLA control	LSI-4 chips; ODT; Floating-Point (FIS), CIS, WCS mid-life kickers
11/04	9/75	2.8	18	8-56		Standard package; ROM; PLA	Backplane compatible with 11/34 for field upgrade; built-in ASCII console; self-diagnosis
11/05	6/72	2.5	13	8-56		Microprogrammed; ROM	Minimal 11 (2 boards)
11/20	6/70	3.1	20	8-56			ISP; Unibus
11/34	3/76	3.5	204	16-256		Shared use of ALU; PLA; ROM; microprogrammed	Cost-performance balance; 11/34C mid-life kicker; bit-slice FPP
11/34C	5/78	7.3	262	32-256			Classic use of cache
11/40	1/73	3.6	57	16-256	Variable cycle length	Microprogrammed	FIS extension
11/60	6/77	27	592	32-256	Fetch overlap; dual scratch-pads; TTL/S	Heavily microprogrammed	Integral floating-point; WCS for local storage; RAMP
11/45	6/72	Core: 13 MOS: 23 Bipolar: 41	~260 ~335 ~362	8-256	Instruction prefetch; dual scratchpads; Fastbus; autonomous FPP; TTL/S		Pc speed to match 300 ns bipolar; high speed minicomputer FPP; memory management
11/55	6/76	41	725	16-64 (0-192 core)	All bipolar memory		
11/70	3/75	36	671	64-2048	32-bit wide DMA bus; large memory		Cache; multiple buses, RAMP, FP11-C mid-life kicker; remote diagnosis
70mP							Multiprocessor architectural extensions; on-line maintainability; performance; availability
		range: 41-1	range: 56-1	range: 256-1			

PDP-11/20, gate density has not improved markedly. Speed improvement has taken place in the Schottky TTL, and a speed/power improvement has occurred in the low power Schottky (LS) series. Departures from medium-scale integrated transistor-transistor logic, in terms of gate density, have been few, but effective. Examples are the bit-slice in the PDP-11/34 Floating-Point Processor, the use of programmable logic arrays in the PDP-11/04 and PDP-11/34 control units, and the use of emitter-coupled logic in some clock circuitry.

Memory densities and costs have improved rapidly since 1969 and have thus had the most impact. Read-write memory chips have gone from 16 bits to 4,096 bits in density and read-only memories from 16 bits to the 8 or 16 Kbits widely available in 1978.

The memory technology of 1969 imposed several constraints. First, core memory was cost-effective for the primary (program) memory, but a clear trend toward semiconductor primary memory was visible. Second, since the largest high speed read-write memories available were just 16 words, the number of processor registers had to be kept small. Third, there were no large high speed read-only memories that would have permitted a microprogrammed approach to the processor design.

These constraints established four design attitudes toward the PDP-11's architecture. First, it should be asynchronous, and thereby capable of accepting different configurations of memory that operate at different speeds. Second, it should be expandable to take eventual advantage of a larger number of registers, both user registers for new data-types and internal registers for improved context switching, memory mapping, and protected multiprogramming. Third, it could be relatively complex, so that a microcode approach could eventually be used to advantage: new data-types could be added to the instruction set to increase performance, even though they might add complexity. Fourth, the Unibus width should be relatively large, to get as much performance as possible, since the amount of computation possible per memory cycle was relatively small.

As semiconductor memory of varying price and performance became available, it was used to trade cost for performance across a reasonably wide range of PDP-11 models. Different techniques

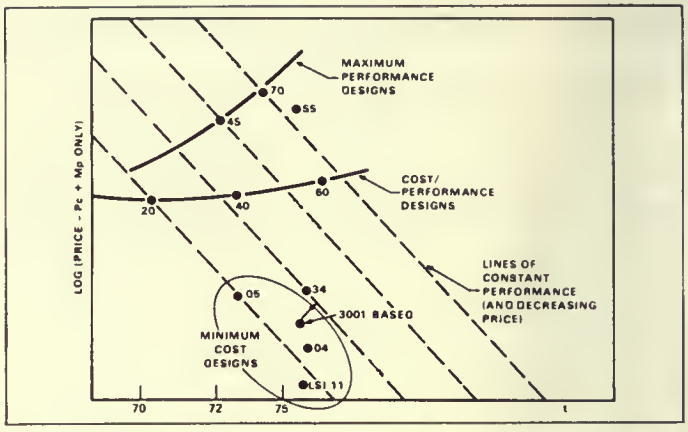

Fig. 2. PDP-11 models price versus time with lines of constant performance.

were used on different models to provide the range. These techniques include: microprogramming for all models except the 11/20 to lower cost and enhance performance with more data-types (for example, faster floating point); use of faster program memories for brute-force speed improvements (e.g., 11/45 with MOS primary memory, 11/55 with bipolar primary memory, and the 11/60 with a large writable control store); use of caches (11/70, 11/60, and 11/34C); and expanded use of fast registers inside the processor (the 11/45 and above). The use of semiconductors versus cores for primary memory is a purely economic consideration.

Table 2 shows characteristics of each of the PDP-11 models along with the techniques used to span a range of cost and performance. (Chapter 39 gives a detailed comparison of the processors.) Figure 2 gives the cost/performance mapping for the various PDP-11 implementations.

References

Bell et al. [1970]; Eckhouse [1975]; Gear [1974]; Shannon [1948]; Stone and Siewiorek [1975].