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.

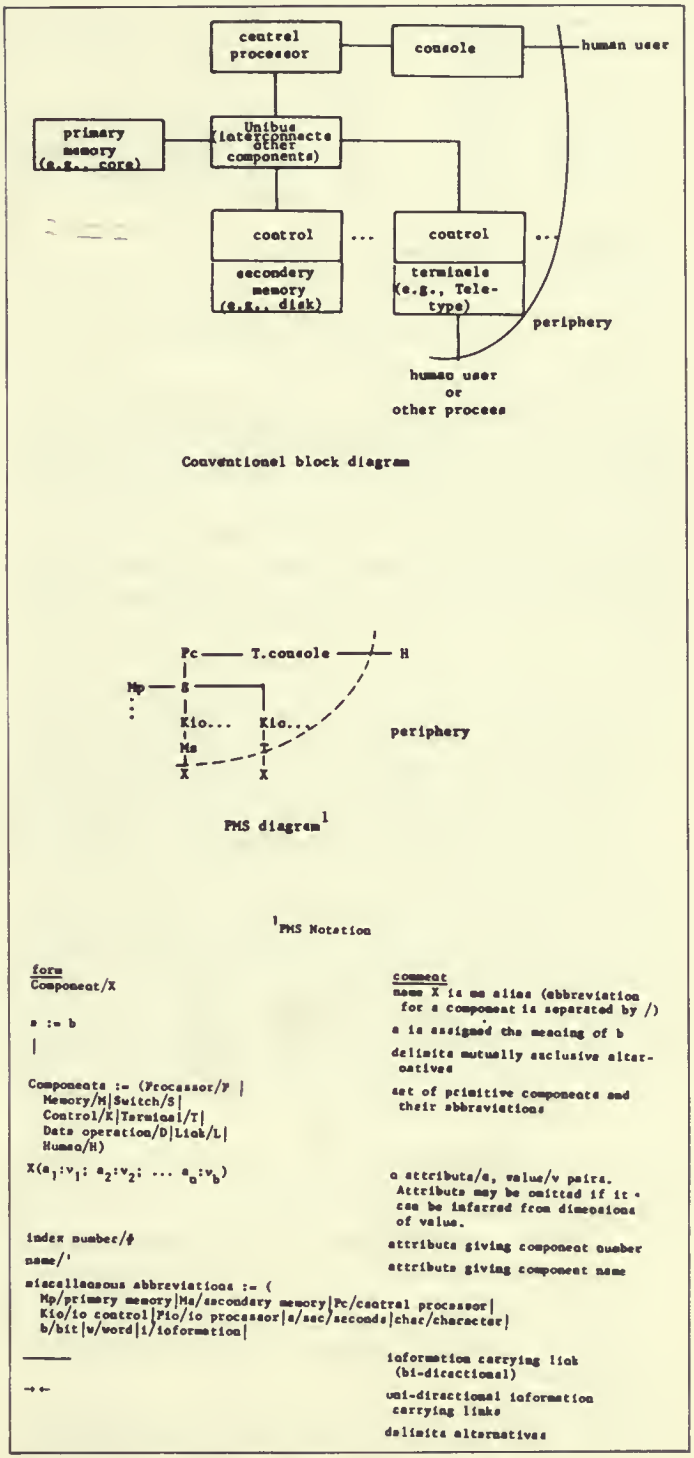

Fig. 1. Conventional block diagram and PMS diagram of PDP-11.

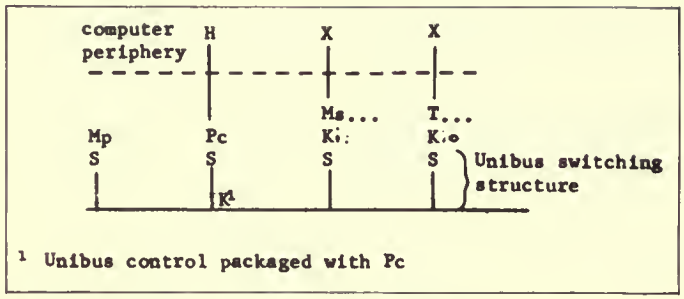

Fig. 2. PDP-11 physical structure PMS diagram.

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.)

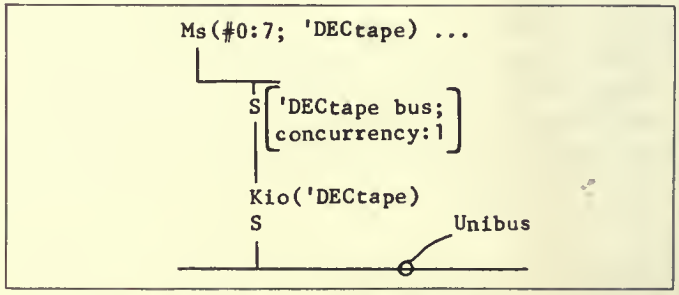

Fig. 3. DECtape control switching PMS diagram.

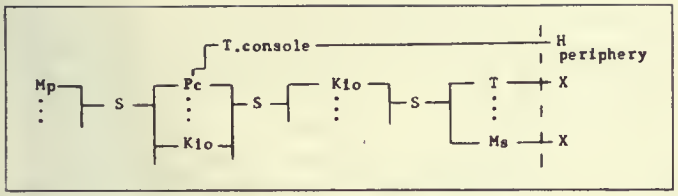

Fig. 4. Conventional hierarchy computer structure.

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

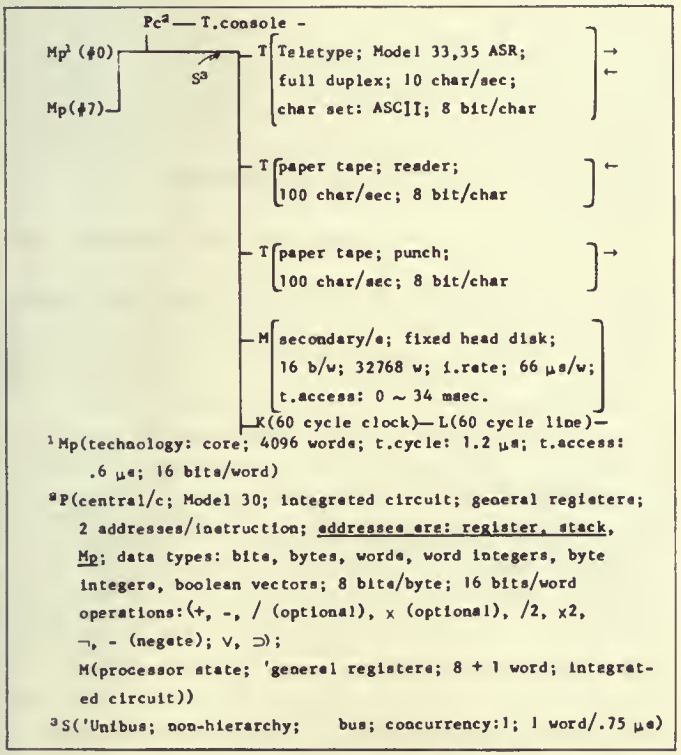

Fig. 5. PDP-11 structure and characteristics PMS diagram.

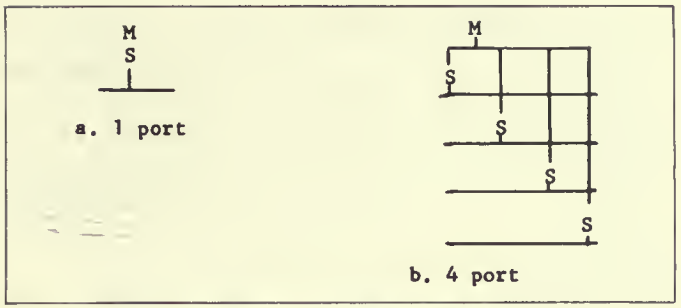

Fig. 6. 1 and 4 port memory modules PMS diagram.

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

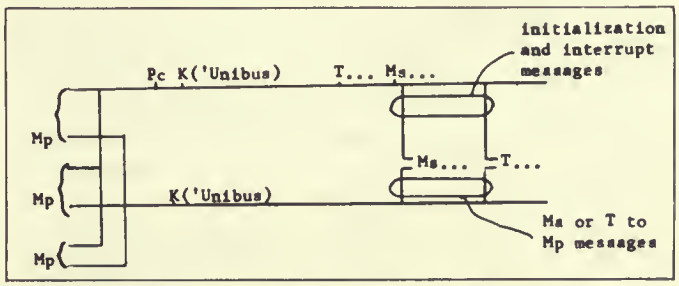

Fig. 7. Three Mp, 2 S('Unibus) structure PMS diagram.

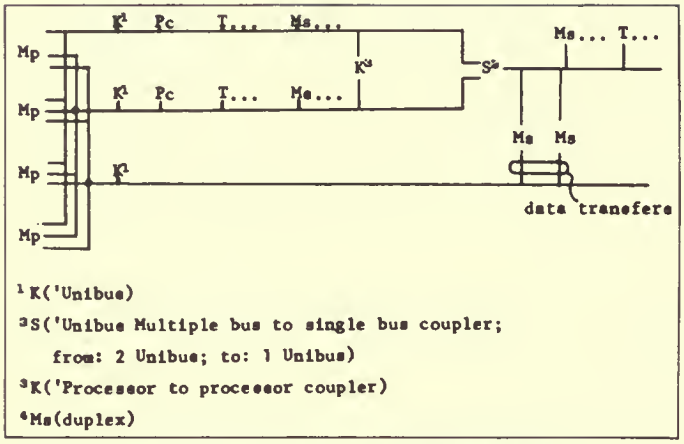

Fig. 8. Dual Pc multiprocessor system PMS diagram.

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

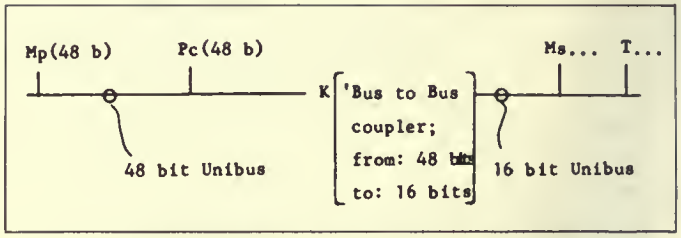

Fig. 9. Computer with 48 bit Pc, Mp with 16 bit Ms, T PMS diagram.

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,

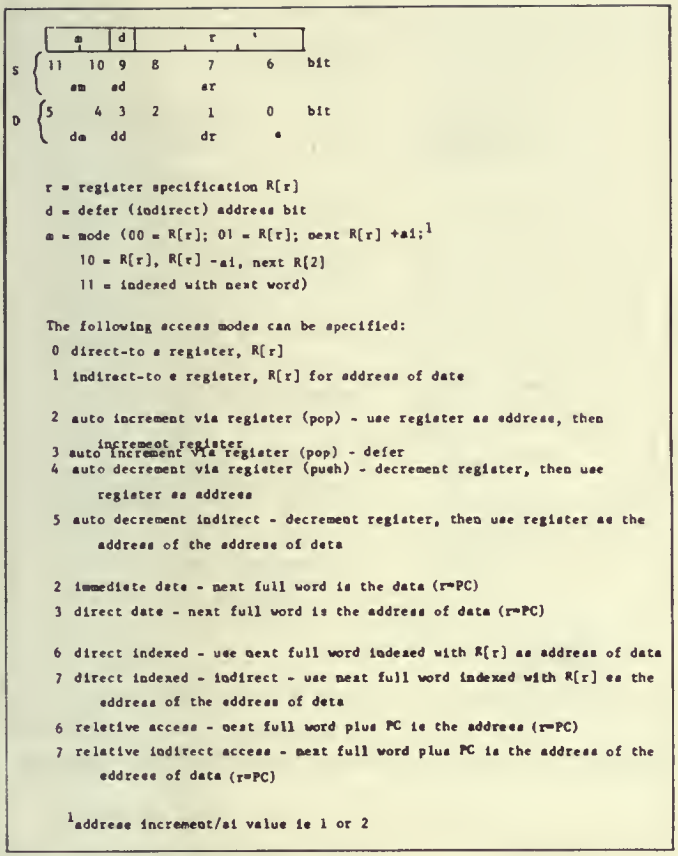

Fig. 10. Address calculation formats.

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.

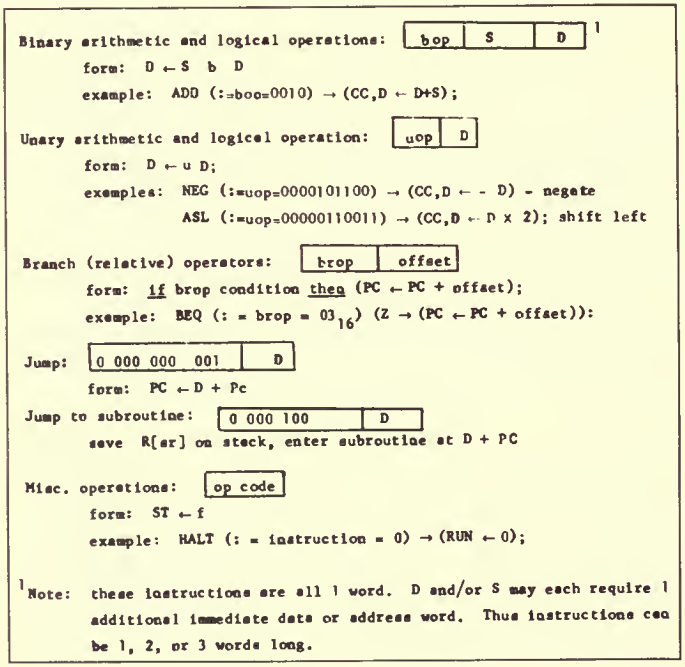

Fig. 12. PDP-11 instruction formats (simplified).

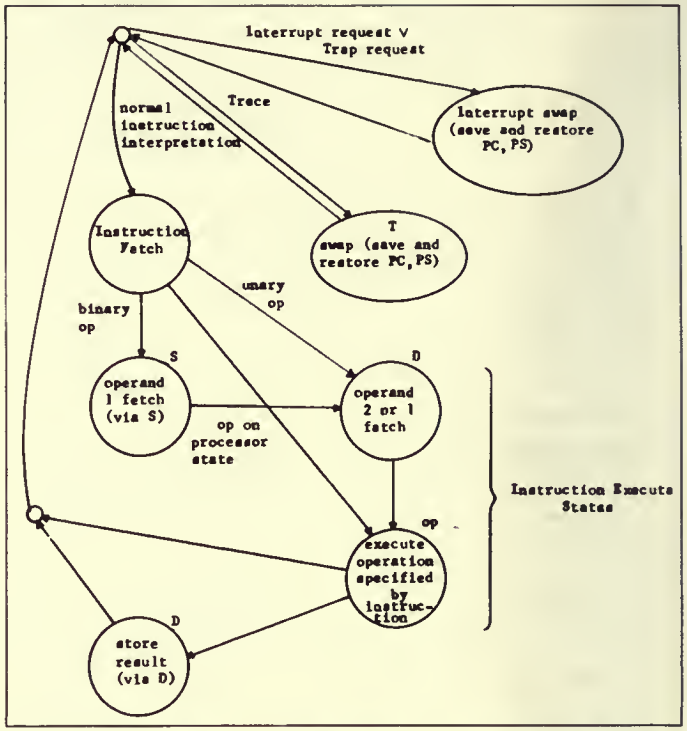

Fig. 13. PDP-11 instruction interpretation process state diagram.

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.

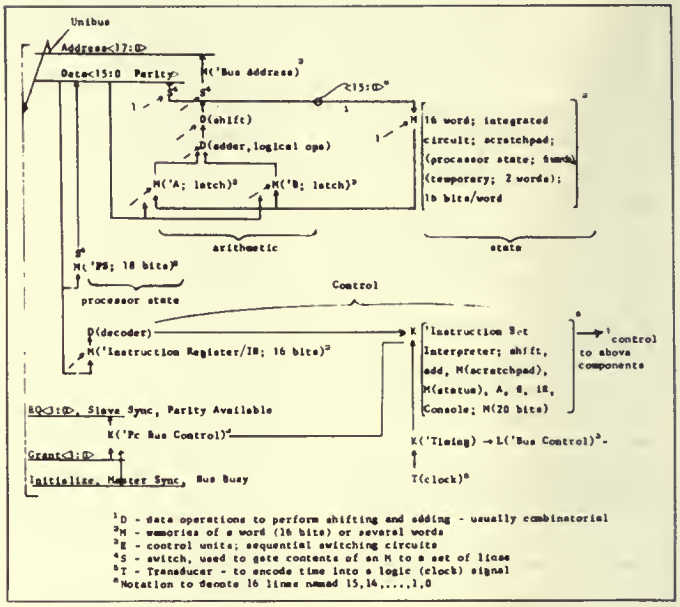

Fig. 16. PDP-11 Pc structure.

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
```