Chapter 38

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


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.


\(^{2}\)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 MHz . . . , 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:

<table>
<thead>
<tr>
<th>Maximum addressable primary memory (words)</th>
<th>Processor and memory cost (1970 kilodollars)</th>
<th>Word length (bits)</th>
<th>Processor state (words)</th>
<th>Data types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Micro</td>
<td>8 K</td>
<td>~5</td>
<td>8 - 12</td>
<td>2</td>
</tr>
<tr>
<td>Mini</td>
<td>32 K</td>
<td>5 - 10</td>
<td>12 - 16</td>
<td>2-4</td>
</tr>
<tr>
<td>Midi</td>
<td>65 - 128 K</td>
<td>10 - 20</td>
<td>16 - 24</td>
<td>4-16</td>
</tr>
</tbody>
</table>
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 60 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 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
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**

**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

---

1A 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.
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.](image)
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 \( \mu \)sec., 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 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
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 Pcs (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 providing multiple busses and defining wider path busses. Finally, and most important, the Unibus is straightforward.

The Instruction Set Processor (ISP) Level-Architecture

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, 1965]. 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).

Although the approach to the design was conventional, the

1The 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.”

2A 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:

\[ op \ l_1, l_2, l_3, l_4 \]

where \( l_1 \) specifies the location (address) in which to store the result of the binary operation \((op)\) of the contents of operand locations \( l_2 \) and \( l_3 \), and \( l_4 \) specifies the location of the next instruction.

The action of the instruction is of the form:

\[ l_1 \leftarrow l_2 \ op \ l_3; \ goto \ l_4 \]

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 \( l_1 = l_2 \) = the “accumulator” and \( l_4 \) is the location following that of the current instruction. The two-address machine assumes \( l_1 = l_2; l_4 \) 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). 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
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 many 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>; 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.)

1A 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" ($x \lor y$ or $\neg x \lor y$).

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,
Chapter 38 | A New Architecture for Mini-Computers: The DEC PDP-11

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

The following access modes can be specified:

1. Direct-to-a-register, R[r]
2. Indirect-to-a-register, R[R]
3. Address of data
4. Immediate data - next full word is the data (r=PC)
5. Direct data - next full word is the address of data (r=PC)
6. Direct indexed - use next full word indexed with R[r] as address of data
7. Relative access - next full word plus PC is the address (r=PC)

Fig. 10. Address calculation formats.

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 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 +, −, ×, /,

Note, by convention a stack builds toward register 0, and when the stack crosses 4000, a stack overflow occurs.

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

\[ \text{op}\ D\ S \]

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.

---

**Binary arithmetic and logical operations:**

\[ \text{bop}\ S\ D\ ]

form: \( D \leftarrow S + D \)

example: \( \text{ADD} (\text{bop}\ =\ 0010) \leftarrow (CC, D \leftarrow D+S) \)

**Unary arithmetic and logical operation:**

\[ \text{uop}\ D\ ]

form: \( D \leftarrow u\ D; \)

examples:
- \( \text{NEG} (\text{uop}\ =\ 00000110100) \leftarrow (CC, D \leftarrow D) \) — negate
- \( \text{ASL} (\text{uop}\ =\ 00000110001) \leftarrow (CC, D \leftarrow D \times 2); \) shift left

**Branch (relative) operator:**

\[ \text{bop}\ \text{offset} \]

form: \( \text{if} \) \( \text{bop} \) \( \text{condition} \) \( \text{then} \) \( (\text{PC} \leftarrow \text{PC} + \text{offset}) \)

example: \( \text{BEQ} (\text{bop}\ =\ 01) (Z \leftarrow (\text{PC} \leftarrow \text{PC} + \text{offset})) \)

**Jump:**

\[ 0\ 000\ 000\ 001\ D \]

form: \( \text{PC} \leftarrow D + \text{PC} \)

**Jump to subroutine:**

\[ 0\ 000\ 100\ D \]

save \( R[\text{ar}] \) on stack, enter subroutine at \( D + \text{PC} \)

**Misc. operations:**

\[ \text{op code} \]

form: \( \text{ST} = f \)

example: \( \text{HALT} (\text{op}\ =\ 0) \rightarrow \text{RUN} \leftarrow 0; \)

\[ ^1 \text{Note: these instructions are all 1 word. D and/or S may each require 1 additional immediate data or address word. Thus instructions can be 1, 2, or 3 words long.} \]

---

**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: \( \text{ADD} (\text{bop}\ =\ 0010) \leftarrow (CC, D \leftarrow D+S) \); addition does not exactly describe the changes to the Condition Codes/CC (which means whenever a binary opcode [bop] of 0010 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.
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:

<table>
<thead>
<tr>
<th>Programmed</th>
<th>250 microseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programmed but (special normalize and differencing of exponent instructions)</td>
<td>75 microseconds</td>
</tr>
<tr>
<td>Microprogrammed hardware</td>
<td>25 microseconds</td>
</tr>
<tr>
<td>Hardwired</td>
<td>2 microseconds</td>
</tr>
</tbody>
</table>

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.
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:

<table>
<thead>
<tr>
<th>binary ops</th>
<th>op</th>
<th>floating point</th>
<th>and double word</th>
</tr>
</thead>
<tbody>
<tr>
<td>bop</td>
<td>S D</td>
<td>FMOVE</td>
<td>DMOVE</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>FADD</td>
<td>DADD</td>
</tr>
<tr>
<td></td>
<td>−</td>
<td>FSUB</td>
<td>DSUB</td>
</tr>
<tr>
<td></td>
<td>×</td>
<td>FMUL</td>
<td>DMUL</td>
</tr>
<tr>
<td></td>
<td>/</td>
<td>FDIV</td>
<td>DDIV</td>
</tr>
<tr>
<td></td>
<td>compare</td>
<td>FCMP</td>
<td>DCMP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>unary ops</th>
</tr>
</thead>
<tbody>
<tr>
<td>uop</td>
</tr>
</tbody>
</table>

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.

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(add, 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

APPENDIX 1  PDP-11 ISP

PO511 := begin
! This is a summary description of a PDP-11/70 processor written
! in the ISP5 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, HGRROM[53:0]
! should be initialized by use of an external RLAO 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 := [16777777].
| (256 * 2 bytes)

MM(max byte:0):= MM(max byte:0);[16777777].
| the addressing space
MM(max byte:0):= MM(max byte:0);[16777700].
| the i/o page (4k)
MM(max byte:0):= MM(max byte:0);[16777700].
| the i/o page (4k)
MM(max byte:0):= MM(max byte:0);[16777700].
| the i/o page (4k)

MACRO ADDR, reg(21:0).

MACRO ADDR, reg(21:0).

**MC.State**

R.register[15:0]:= R[15:0].
| the register file including two sets of general
| registers: A0-A5 (address 0000-0101, 1000-1101),
| One program counter (address 0111), and three
| Stack pointers (address 0110, 1110, 1111).

PC[15:0]:= R[0111].
| Only 1 program counter

macro SP := [R[0110] and R[0111]].
| Stack pointer (3)
macro link := [R[0111]].
| Two R's (subroutine link)

PC[15:0]:= MM[17777777:17777777].
| Program status word
ccstatus(1:0):= (PS[15:14].
| Current address space
! kernel/ supervisor/user
macro kernel := [[R[0110] eq '0]].

macro super := [[R[0110] eq '1]].

macro user := [[R[0110] eq '1]].

macro prevmode := [R[12:12]].
| Previous address space
! program priority(2:0)
! (PS[2:0].)

wsregister.set := [R[11:11]].
| \trace()
! (PS[3:3].)

cccondition.codes[3:0] := [R[3:0]].

Nnegative := [cc3].

Zzero := [cc2].

Voverflow := [cc1].

Ccarry := [cc0].

**Implementation.Declarations**

bus.errorcr<0>,
byte.ACCESS<0>,
ccmode<1:0>,
bustreg<15:0>,
destination.reg.addr,
in.register.file<3:0>,
iflags,
cld<10:0>,
p.temp<15:0>,
pmode<1:0>,
ps[0:0],
regflags,
HIPROMregister.mapping,
read,only.memory[53:0],
src[0:9],reg.addr.in.register.file<3:0>,
state<15:0>,
Current state 0 = instruction fetch/decode
1 = execute
2 = service
3 = unused

temp<15:0>,
trace.flags,
trap.instr<0>,
var<21:0>
| Trace trap bit temporary
| Set by event, trap, bpt, and lot to inhibit
| Virtual address register used in read and write
| 84 bits of zeros

**Instruction.Format**

! Instruction format.

! Instruction format.

! Instruction format.

**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 MAR.

source[15:0] := begin
! flag := 0 next
DECIDE sm :=
begin
  0 := source = R[sa].
| Register mode: registers
| are addressed directly.
| Autoincrement mode: use
| the contents of the specified
| register as an address.
| Increment after use.

begin
  H[sa] := R[sa] + [2-(us)byte.address].

begin
  H[sa] := R[sa] + [2-(us)byte.address].
| Decrement contents of the
| begin
| specified register before
| use

begin
  H[sa] := R[sa] + [2-(us)byte.address].

begin
  H[sa] := R[sa] + [2-(us)byte.address].
| Decrement contents of the
| begin
| specified register before
| use
|
APPENDIX 1 (cont'd.)

MAR = [sr]next
read(byte.access *[us] (1 - [us] sd) next source = MBR
end.
3 :=
begin
i = flag = 1; MAR = PC next
PC = PC + 2 next
of the specified register
read(0) next
if flag = 0;
MAR = MBR; [sr]<15:0> next
the register contents
are unmodified.
read(byte.access *[us] (1 - [us] sd) next source = MBR
end
end next

If sd =>
begin
MAR = source next
read(byte.access) next
source = MBR
end.

dest[15:0] :=
begin
if flag = 0; oldval = [dr] next
DECODE dm =>
begin
0 :=
begin
dest = 0; regflag = 1
end;
1 :=
begin
dest = [dr] next
DECODE (dr?) or dd =>
begin
R[dr] = R[dr] + (2 * [us] byte.access),
R[dr] = R[dr] + 2
end next
if flag = (dr eq [us] #7) =>
begin
force 1 space if using PC
end.

2 :=
begin
DECODE (dr?) or dd =>
begin
R[dr] = R[dr] - (2 * [us] byte.access),
R[dr] = R[dr] - 2
end next
dest = R[dr]
end.

3 :=
begin
if flag = 1; MAR = PC next
PC = PC + 2 next
read(0) next
if flag = 0; dest = MBR + [dr]
end
end next
MAR = dest next

If dd =>
begin
read(0) next
if flag = regflag = 0;
MAR = MBR
end.

**Service Facilities**

stack.stack.reference :=
begin
regflag = 0;
SP = SP - 2
end.

odderr|odd.address.error :=
begin
oddaddr = bus.error = 1 next
cstestate()
end.

cstestate(abort():
begin
DECODE state =>
begin
no.op(),
I Fetch
LEV slave, I Execute
LEV slave, I Service
no.op()
end
end.

sd.read(byte.access) :=
begin
source() next
dest() next
read(byte.access)
end.

d.read(byte.access,R[dr]) :=
begin
dest() next
read(byte.access)
end.

read(byte.access) :=
begin
if MAR(0) and not byte.acc => odderr(); var = MAR next
if [var(15:13)] eq [us] #7 => [var(21:10) = #7 next
DECODE var(21:10) eq [us] #17 =>
begin
0 := DECODE regflag =>
begin
0 := DECODE byte.acc =>
begin
MBR = MW[var],
MBR = MBR
end.
end;
1 := DECODE byte.acc =>
begin
MBR = [dr],
MBR = MBR + [dr]</17:0>
end.
end.
end;
1 := IF var(17:13) eq [us] #37 =>
begin
DECODE byte.acc =>
begin
MBR = MW[var],
MBR = MBR + [var]
end
end.
end.
end;
end.
end.

write(byte.acc) :=
begin
IF MAR(0) and not byte.acc => odderr(); var = MAR next
IF [var(15:13)] eq [us] #7 => [var(21:10) = #7 next
DECODE var(21:10) eq [us] #17 =>
begin
0 := DECODE regflag =>
begin
0 := DECODE byte.acc =>
begin
MBR = MW[var],
MBR = MBR + [var]
end.
end;
1 := DECODE byte.acc =>
begin
MBR = [dr],
MBR = MBR + [dr]</17:0>
end.
end.
end.
end.
end.

END.

**Condition Code Setting and Branch Operations**

setcc(n,<15:0>, v,<16:0>, z,<15:0>) :=
begin
DECODE byte.acc =>
begin
0 :=
begin
N = n,<15>
V = v,<16>
Z = z,<15>
end.
end.
end.
end.
end.

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

intv|interrupt trap.vector.setup(vector(6:0)) :=
begin
MAR = busreg = vector;
cmode = byte.access = 0 next
read(byte.access) next
pc.tmp = MAR next
mar = busreg + 2 next
read(byte.access) next

663
APPENDIX 1 (cont'd.)

ps, temp : MTR next
start(); MTR = PS next
MAR = SP next
write(byte access) next
stare(); MTR = PS next
MAR = SP next
write(byte access) next
pm = cm next
PC = PC.temp = PS = ps.temp end,

Instr. traptrap.trap(trap.vector(0:0)) := ! Reserved and illegal
begin
intvec(trap.vector) next
If bus.error => a = 2 ! Malt the processor if bus error occurs here end,

I 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.

grantbus.grant.processing.routine(type.request(15:0)) :=
begin
a = 0 next
intvec(type.request) next
IFAVE service end,

service :=
begin
If bus.error =>
begin
bus.error = 0 next
intvec(op.errors) next
If bus.error =>
begin
a = 2 next
IFAVE service end next
IFAVE service end end

**Instruction.Interpretation**

! Initialization sequence

start(main) :=
begin
zeros = 0;
LOREG = 0;
a = 0 next
! Clear all cpu registers
!
Main run cycle of the ISP

run(instruction, interpretation) :=
begin
If go =>
begin
state = 1; instr = 0;
MAR = PC next
DECOD MAKO(0) =>
begin
D := begin
! Even
comd = cm; reg1g = 0 next
read(0) next
! Instruction fetch
IR = MBR: PC = PC + 2 next
byte.access = byte: trace: flag = 1;
sr = REGROM[comd & rs & srreg];
sr = REGROM[comd & rs & srreg];
state = 1 next
exec();
end
end
end
end
II HALT => STOP() next
state = 2 next
II service() next
II START run end

**Instruction.Execution**

exec(instruction.execution) :=
begin
DECODE bop =>
begin
#0 := reserop(),
#1 := MOV(),
#2 := CMP := no.op(),
#3 := BIT := no.op(),
#4 := OIC := no.op(),
#5 := OIS := no.op(),
#6 := begin
DECODE byte.access =>
begin
D := ADD := no.op().

! Reserved op code
! Move instruction
! Compare instruction
! Bit test instruction
! Bit clear instruction
! Add and subtract

1 := SUB := no.op();
end
#7 := no.op();

! Extended instruction set
end
end
reserop.reserve.op.code :=
begin
DECODE rop =>
begin
0 : = branop(),
1 : = classop(),
end
end
branop|branch.op.codes :=
begin
DECODE (jtop & brap)(3:0) =>
begin
#0 := regop(),
#1 := branch(0),
#2 := NEE := branch(not 0),
#3 := B10 := branch(1),
#4 := B11 := branch(N eq V),
#5 := B11 := branch(N xor V),
#6 := B11 := branch(not (1 or (N xor V))),
#7 := B11 := branch(! or (N xor V),
#8 := B10 := branch(0),
#9 := B10 := branch(N),
#10 := B11 := branch(c or z),
#11 := B11 := branch(c not 0),
#12 := B10 := branch(0),
#13 := B10 := branch(c not 0),
#14 := B10 := branch(0),
#15 := B10 := branch(0),
#16 := B10 := branch(c not 0),
#17 := B10 := branch(c not 0),
end
end
reg|register.operations :=
begin
DECODE rop =>
begin
0 := begin
If cont eq 0 =>
begin
DECOD (jtop & brap) =>
begin
#0 := H10(),
#1 := H11 := no.op(),
#2 := H11 := no.op(),
#3 := H11 := no.op(),
#4 := H11 := no.op(),
#5 := H11 := no.op(),
#6 := H11 := no.op(),
#7 := instr.trap(res.instr)
end
end
end
end
end
end
end
end

class|secondary|decode.into|classes :=
begin
DECODE typeop =>
begin
subrclass(subclass) :=
begin
subrclass(1) :=
begin
DECODE jtopp =>
begin
D := JSR(),
begin
D := JSR(),
begin
D := JSR(),
begin
D := JSR(),
begin
D := JSR(),
begin
D := JSR(),
begin
D := JSR(),
begin
D := JSR(),
begin
D := JSR(),
begin
D := JSR(),
begin
D := JSR(),
begin
D := JSR(),
begin
D := JSR(),
begin
D := JSR(),
begin
D := JSR(),
begin
D := JSR(),
begin
D := JSR(),
begin
D := JSR(),
begin
D := JSR(),
begin
D := JSR(),
begin
D := JSR(),
begin
D := JSR()
APPENDIX 1 (cont'd.)

```
singleloop\singleoperand.instructions :=
  begin
  DECODE wop =>
  begin
    0 := CLR();
    1 := COM := no.op();
    2 := INC := no.op();
    3 := DEC := no.op();
    4 := NEG := no.op();
    5 := ADD := no.op();
    6 := SUB := no.op();
    7 := TEST := no.op();
  end;

  shiftloop\shift operand.instructions :=
  begin
  DECODE wop =>
  begin
    0 := SHR();
    1 := ROR := no.op();
    2 := ASR := no.op();
    3 := CLRB := no.op();
    4 := NOP := no.op();
    5 := HLT := no.op();
    6 := JMP := no.op();
    7 := INT := no.op();
  end;

  movloop\move operand.instructions :=
  begin
  DECODE wop =>
  begin
    0 := ADD();
    1 := CLR();
    2 := AND := no.op();
    3 := OR := no.op();
    4 := XOR := no.op();
    5 := NOT := no.op();
    6 := TEST := no.op();
    7 := MOV := no.op();
  end;

  condloop\condition instructions :=
  begin
  DECODE wop =>
  begin
    0 := JNE();
    1 := JZ := no.op();
    2 := JNS := no.op();
    3 := JN := no.op();
    4 := JNZ := no.op();
    5 := JNZS := no.op();
    6 := JNS := no.op();
    7 := JNZ := no.op();
  end;

  singleloop\single operand instructions :=
  begin
  DECODE wop =>
  begin
    0 := CLR();
    1 := COM := no.op();
    2 := INC := no.op();
    3 := DEC := no.op();
    4 := NEG := no.op();
    5 := ADD := no.op();
    6 := SUB := no.op();
    7 := TEST := no.op();
  end;
```

! Jump, swap execution and register operation decode

```
JMP :=
  begin
  ENCODE (dl @ dd) eq 0 =>
  begin
    0 := (dest() next PC = MAR), false
    1 := instr.trap(1, instr), true
  end;

SWAB :=
  begin
  d.read(byte, access) next
  MDR = bmbr @ MBR<15:0> next
  C = V = 0: N = MBR<7:0> eq 1 0:
  if d eq 07 => write(byte, access)
  end;
```

! Shift instruction execution

```
ROR :=
  begin
  d.read(byte, access) next
  DECODE byte, access =>
  begin
    0 := (temp<15:0> = (c @ MBR) srr 1 next
    1 := (temp<15:0> = temp<15:0>)
    2 := temp<15:0> = temp<15:0>)
  end;

MLT :=
  begin
  Halt. MLT op code #00000
  DECODE kernel
  begin
    0 := (intvec(cpu, errors))
    1 := a = 2
  end;

RTS :=
  begin
  Halt. RTS op code #00020
  DECODE pc
  begin
    0 := (pc @ MBR) next
    1 := (sp @ MBR) next
    2 := sp = z next
    3 := sp = z next
    4 := sp = z next
    5 := sp = z next
  end;
```

! CPU control instruction execution

```
im =
  begin
  emt op code #104000:
  intvec(emt, trap), trap instr = 1
  end,

trap :=
  begin
  trap op code #104400:
  intvec(trap, trap), trap instr = 1
  end,
```

! Single operand instruction execution

```
CLR :=
  begin
  CLR op code #0050, CLR op code #1050
  begin
    cc := '0100 next
    dest() next
    MDR = 0 next
    write(byte, access)
  end,
```

! User instructions execute

```
singleloop\single operand.instructions :=
  begin
  DECODE wop =>
  begin
    0 := CLR();
    1 := COM := no.op();
    2 := INC := no.op();
    3 := DEC := no.op();
    4 := NEG := no.op();
    5 := ADD := no.op();
    6 := SUB := no.op();
    7 := TEST := no.op();
  end;

  shiftloop\shift operand.instructions :=
  begin
  DECODE wop =>
  begin
    0 := SHR();
    1 := ROR := no.op();
    2 := ASR := no.op();
    3 := CLRB := no.op();
    4 := NOP := no.op();
    5 := HLT := no.op();
    6 := JMP := no.op();
    7 := INT := no.op();
  end;

  movloop\move operand.instructions :=
  begin
  DECODE wop =>
  begin
    0 := ADD();
    1 := CLR();
    2 := AND := no.op();
    3 := OR := no.op();
    4 := XOR := no.op();
    5 := NOT := no.op();
    6 := TEST := no.op();
    7 := MOV := no.op();
  end;

  condloop\condition instructions :=
  begin
  DECODE wop =>
  begin
    0 := JNE();
    1 := JZ := no.op();
    2 := JNS := no.op();
    3 := JN := no.op();
    4 := JNZ := no.op();
    5 := JNZS := no.op();
    6 := JNS := no.op();
    7 := JNZ := no.op();
  end;

  singleloop\single operand instructions :=
  begin
  DECODE wop =>
  begin
    0 := CLR();
    1 := COM := no.op();
    2 := INC := no.op();
    3 := DEC := no.op();
    4 := NEG := no.op();
    5 := ADD := no.op();
    6 := SUB := no.op();
    7 := TEST := no.op();
  end;
```
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 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.
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).

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.

### 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 CPU. 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 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

¹N.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.
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. 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-non modifying, 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 Strecker1 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.

---

1Private communication.
Table 1  Average PDP-11 Instruction Execution Times in Microseconds

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

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:

\[ t_i = k_1 c_{i1} + k_2 c_{2i} \]

where \( t_i \) = the average instruction execution time of machine \( i \) from Table 1
\( c_{i1} \) = 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, \( k_1 = 11.550, k_2 = 1.162, \) 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

<table>
<thead>
<tr>
<th></th>
<th>Independent variables</th>
<th>Dependent variable</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Microcycle time</td>
<td>Memory-read-pause-time</td>
</tr>
<tr>
<td>LSI-11</td>
<td>0.400</td>
<td>0.400</td>
</tr>
<tr>
<td>PDP-11/04</td>
<td>0.260</td>
<td>0.940</td>
</tr>
<tr>
<td>PDP-11/10</td>
<td>0.300</td>
<td>0.600</td>
</tr>
<tr>
<td>PDP-11/20</td>
<td>0.280</td>
<td>0.370</td>
</tr>
<tr>
<td>PDP-11/34</td>
<td>0.180</td>
<td>0.940</td>
</tr>
<tr>
<td>PDP-11/40</td>
<td>0.140</td>
<td>0.500</td>
</tr>
<tr>
<td>PDP-11/45</td>
<td>0.150</td>
<td>0.000</td>
</tr>
<tr>
<td>(bipolar memory)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PDP-11/60</td>
<td>(87% cache hit ratio)</td>
<td>0.170</td>
</tr>
</tbody>
</table>
average instruction time is more sensitive to microcycle time than to memory-read-pause time by a factor of $k_2/k_1$ 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 affected, 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 $BA\leftarrow PC; DATI; PC\leftarrow PC+2$ 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

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

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:

\[ t = (k_1 + k_3k_4) - k_5(k_1 - k_3)a \]

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.

<table>
<thead>
<tr>
<th>Frequency</th>
<th>Execute Instruction</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>1.0000</td>
<td></td>
</tr>
<tr>
<td>Source</td>
<td>0.4069</td>
<td></td>
</tr>
<tr>
<td>Mode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 R</td>
<td>0.1377</td>
<td></td>
</tr>
<tr>
<td>1 @R or (R)</td>
<td>0.0338</td>
<td></td>
</tr>
<tr>
<td>2 (R)+</td>
<td>0.1587</td>
<td></td>
</tr>
<tr>
<td>3 @R(R)+</td>
<td>0.0122</td>
<td></td>
</tr>
<tr>
<td>4 -(R)</td>
<td>0.0352</td>
<td></td>
</tr>
<tr>
<td>5 @-(R)</td>
<td>0.0000</td>
<td></td>
</tr>
<tr>
<td>6 X(R)</td>
<td>0.0271</td>
<td></td>
</tr>
<tr>
<td>7 @X(R)</td>
<td>0.0022</td>
<td></td>
</tr>
</tbody>
</table>

**Destination Mode** 0.6872

<table>
<thead>
<tr>
<th>Frequency</th>
<th>Execute Instruction</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Manipulation</td>
<td>0.6355</td>
<td></td>
</tr>
<tr>
<td>Mode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 R</td>
<td>0.3146</td>
<td></td>
</tr>
<tr>
<td>1 @R or R</td>
<td>0.0599</td>
<td></td>
</tr>
<tr>
<td>2 (R)+</td>
<td>0.0854</td>
<td></td>
</tr>
<tr>
<td>3 @R(R)+</td>
<td>0.0307</td>
<td></td>
</tr>
<tr>
<td>4 -(R)</td>
<td>0.0823</td>
<td></td>
</tr>
<tr>
<td>5 @-(R)</td>
<td>0.0000</td>
<td></td>
</tr>
<tr>
<td>6 X(R)</td>
<td>0.0547</td>
<td></td>
</tr>
<tr>
<td>7 @X(R)</td>
<td>0.0080</td>
<td></td>
</tr>
</tbody>
</table>

**Jump (JMP/JSR)** 0.0517

<table>
<thead>
<tr>
<th>Frequency</th>
<th>Execute Instruction</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operand Mode</td>
<td>0.0000 (ILLEGAL)</td>
<td></td>
</tr>
<tr>
<td>0 R</td>
<td>0.0000</td>
<td></td>
</tr>
<tr>
<td>1 @R or (R)</td>
<td>0.0000</td>
<td></td>
</tr>
<tr>
<td>2 (R)+</td>
<td>0.0000</td>
<td></td>
</tr>
<tr>
<td>3 @R(R)+</td>
<td>0.0079</td>
<td></td>
</tr>
<tr>
<td>4 -(R)</td>
<td>0.0000</td>
<td></td>
</tr>
<tr>
<td>5 @-(R)</td>
<td>0.0000</td>
<td></td>
</tr>
<tr>
<td>6 X(R)</td>
<td>0.0438</td>
<td></td>
</tr>
<tr>
<td>7 @X(R)</td>
<td>0.0000</td>
<td></td>
</tr>
</tbody>
</table>

**No Destination** 0.3128

<table>
<thead>
<tr>
<th>Frequency</th>
<th>Execute Instruction</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double operand</td>
<td>0.4069</td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td>0.0524</td>
<td></td>
</tr>
<tr>
<td>SUB</td>
<td>0.0274</td>
<td></td>
</tr>
<tr>
<td>BIC</td>
<td>0.0309</td>
<td></td>
</tr>
<tr>
<td>BICB</td>
<td>0.0000</td>
<td></td>
</tr>
<tr>
<td>BIS</td>
<td>0.0012</td>
<td></td>
</tr>
<tr>
<td>BISB</td>
<td>0.0013</td>
<td></td>
</tr>
<tr>
<td>CMP</td>
<td>0.0626</td>
<td></td>
</tr>
<tr>
<td>CMPB</td>
<td>0.0212</td>
<td></td>
</tr>
<tr>
<td>BIT</td>
<td>0.0041</td>
<td></td>
</tr>
<tr>
<td>BITB</td>
<td>0.0014</td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>0.1517</td>
<td></td>
</tr>
<tr>
<td>MOVB</td>
<td>0.0524</td>
<td></td>
</tr>
<tr>
<td>XOR</td>
<td>0.0000</td>
<td></td>
</tr>
<tr>
<td>Single operand</td>
<td>0.2286</td>
<td></td>
</tr>
<tr>
<td>CLR</td>
<td>0.0186</td>
<td></td>
</tr>
<tr>
<td>CLRB</td>
<td>0.0018</td>
<td></td>
</tr>
<tr>
<td>COM</td>
<td>0.0000</td>
<td></td>
</tr>
<tr>
<td>COMB</td>
<td>0.0000</td>
<td></td>
</tr>
<tr>
<td>INC</td>
<td>0.0224</td>
<td></td>
</tr>
<tr>
<td>INCB</td>
<td>0.0000</td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>0.0809</td>
<td></td>
</tr>
<tr>
<td>DECB</td>
<td>0.0000</td>
<td></td>
</tr>
<tr>
<td>NEG</td>
<td>0.0038</td>
<td></td>
</tr>
<tr>
<td>NEGB</td>
<td>0.0000</td>
<td></td>
</tr>
<tr>
<td>ADC</td>
<td>0.0070</td>
<td></td>
</tr>
<tr>
<td>ADCB</td>
<td>0.0000</td>
<td></td>
</tr>
<tr>
<td>SBC</td>
<td>0.0000</td>
<td></td>
</tr>
<tr>
<td>SBCB</td>
<td>0.0000</td>
<td></td>
</tr>
<tr>
<td>ROR</td>
<td>0.0036</td>
<td></td>
</tr>
<tr>
<td>RORB</td>
<td>0.0000</td>
<td></td>
</tr>
<tr>
<td>ROL</td>
<td>0.0059</td>
<td></td>
</tr>
<tr>
<td>ROLB</td>
<td>0.0000</td>
<td></td>
</tr>
<tr>
<td>ASR</td>
<td>0.0069</td>
<td></td>
</tr>
<tr>
<td>ASRB</td>
<td>0.0000</td>
<td></td>
</tr>
<tr>
<td>ASL</td>
<td>0.0298</td>
<td></td>
</tr>
<tr>
<td>ASLB</td>
<td>0.0000</td>
<td></td>
</tr>
<tr>
<td>TST</td>
<td>0.0329</td>
<td></td>
</tr>
<tr>
<td>TSTB</td>
<td>0.0079</td>
<td></td>
</tr>
<tr>
<td>SWAB</td>
<td>0.0038</td>
<td></td>
</tr>
<tr>
<td>SXT</td>
<td>0.0000</td>
<td></td>
</tr>
</tbody>
</table>

**Branch** 0.2853

<table>
<thead>
<tr>
<th>Frequency</th>
<th>Execute Instruction</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>All branches (true)</td>
<td>0.1744</td>
<td></td>
</tr>
<tr>
<td>All branches (false)</td>
<td>0.1109</td>
<td></td>
</tr>
<tr>
<td>SOB (true)</td>
<td>0.0000</td>
<td></td>
</tr>
<tr>
<td>SOB (false)</td>
<td>0.0000</td>
<td></td>
</tr>
</tbody>
</table>
### Chapter 39 | Implementation and Performance Evaluation of the PDP-11 Family

#### Table: Frequency of Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jump</td>
<td>0.0517</td>
</tr>
<tr>
<td>JMP</td>
<td>0.0272</td>
</tr>
<tr>
<td>JSR</td>
<td>0.0245</td>
</tr>
<tr>
<td>Control, trap and miscellaneous</td>
<td>0.0270</td>
</tr>
<tr>
<td>Set/clear condition codes</td>
<td>0.0017</td>
</tr>
<tr>
<td>MARK</td>
<td>0.0000</td>
</tr>
<tr>
<td>RTS</td>
<td>0.0236</td>
</tr>
<tr>
<td>RTI</td>
<td>0.0000</td>
</tr>
<tr>
<td>RTT</td>
<td>0.0000</td>
</tr>
<tr>
<td>IOT</td>
<td>0.0000</td>
</tr>
<tr>
<td>EMT</td>
<td>0.0017</td>
</tr>
<tr>
<td>TRAP</td>
<td>0.0000</td>
</tr>
<tr>
<td>BPT</td>
<td>0.0000</td>
</tr>
</tbody>
</table>

**NOTES:** Frequency of destination odd-byte addressing (DM1-7) = 0.0213.

Execution frequencies indicated as 0.0000 have an aggregate frequency < 0.0050.