Chapter 5

The DEC PDP-8

Introduction[1]

The PDP-8 is a single-address, 12-bit-word computer of the second generation. It is designed for task environments with minimum arithmetic computing and small Mp requirements. For example, it can be used to control laboratory devices, such as gas chromotographs or sampling oscilloscopes. Together with special T's, it is programmed to be a laboratory instrument, such as a pulse height analyzer or a spectrum analyzer. These applications are typical of the laboratory and process control requirements for which the machine was designed. As another example, it can serve as a message concentrator by controlling telephone lines to which typewriters and Teletypes are attached. The computer occasionally stands alone as a small-scale general-purpose computer. Most recently it was introduced as a small-scale general-purpose time-sharing system, based on work at Carnegie-Mellon University and DEC. It is used as a KT(display) when it has a P(display; '338); this C is discussed in Chap. 25. The PDP-8 has achieved a production status formerly reserved for IBM computers; about 5,000 have been constructed.

PDP-8 differs from the character-oriented 8-bit computer in Chap. I0; it is not unlike the 16-bit computers, such as the IBM 1800 in Chap. 33. The PDP-8 is typical of several 12-bit computers: the early CDC-160 series (1960), CDC-6600 Peripheral and Control Processor (Chap. 39), the SDS-92, M.I.T. Lincoln Laboratory's Laboratory Instrument Computer LINC (1963), Washington University's Programmed Console (1967), and the SCC 650 (1966).

The PDP-5 (transistor, 1963), PDP-8 (1965), PDP-8/S (serial, 1966) and PDP-8/I (integrated circuit, 1968), PDP-8/L (integrated circuit, 1968) constitute a series of computers based on evolving technology. All of these have identical ISP's. Their PMS structures are nearly identical, and all components other than Pc and Mp are compatible throughout the series. The LINC-8-338 PMS structure is presented in Fig. 1. A cost performance tradeoff took place in the PDP-8 (parallel-by-word arithmetic) and PDP-8/S (serial-by-bit arithmetic) implementations. A PDP-8/S is one-fifteenth of a PDP-8 at one-half the cost. The performance factors can be attributed to 8/1.5 or 5.3 for Mp speed and a factor of about 3 for logical organization, even though the same 2-megahertz logic clock is used in both cases. The PDP-8 is about 6.7 times a PDP-5.

[1]The initials in the title stand for Digital Equipment Corporation Programmed Data Processor.

The ISP of the PDP-8 Pc is about the most trivial in the book. It has only a few data operators, namely, $\leftarrow$, +, − (negate), $\neg$, $\wedge$, / 2, $\times$ 2, (optional) $\times$, /, and normalize. It operates on words, integers, and boolean vectors. However, there are microcoded instructions, which allow compound instructions to be formed in a single instruction.

The computer is straightforward and illustrates the levels discussed in Chap. I. We can easily look at it from the "top down." The C in PMS notation is

```
C('PDP-8; technology:transistors; 12 b/w;
   descendants:'PDP-8/S, 'PDP-8/I, 'PDP-8/L;
   antecedents: 'PDP-5;
   Mp(core; #0:7; 4096 w; tc:1.5 μs/w);
   Pc(Mps(2 ~ 4 w);
      instruction length:1|2 w
      address/instruction:1;
      operations on data/od:(←, +, ¬, ∧, −(negate), × 2,
      / 2, +1)
      optional operations:(×, /, normalize);
      data-types:word, integer, boolean vector;
      operations for data access:4);
   P(display; '338);
   P(c; 'LINC);
   S('I/O Bus; 1 Pc; 64 K);
   Ms(disk, 'DECtape, magnetic tape);
   T(paper tape, card, analog, cathode-ray tube))
```

ISP

The ISP is presented in Appendix I of this chapter (including the optional Extended Arithmetic Element/EAE). The 2^{12}-word Mp is divided into 32 fixed-length pages of 128 words each. Address calculation is based on references to the first page, Page_0, or to the current page of the Program Counter/PC. The effective-address calculation procedure provides for both direct and indirect reference to either the current page or the first page. This scheme allows a 7-bit address to specify local page addresses.

A 2^{15}-word Mp is available on the PDP-8, but addressing greater than 2^{12} words is comparatively inefficient. In the extended range, two 3-bit registers, the Program Field and Data Field Registers, select which of the eight 2^{12}-word blocks are being actively addressed as program and data.

There is an array of eight registers, called the Auto_index registers, which resides in Page_0. This array (Auto_index[0:11]⟨0:7⟩:= M[10_8:17_8]⟨0:11⟩) possesses the useful property that whenever an indirect reference is made to it, a I is first added

[1] Mp(core; 1.5 μs/w; 4096 w: (12 + 1)b)

[2] S('Memory Bus)

[3] Pc(1 ~ 2 w/instruction: data: w, i,bv; 12 b/w: M.processor state(2$\frac{1}{2}$ ~ 3$\frac{1}{2}$) w: technology: transistors; antecedents: PDP-5; descendants; PDP-8S, PDP-8I, PDP-L)

[4] S('I/O Bus; from; Pc; to; 64 K)

[5] K(1 ~ 4 instructions; M.buffer(1 char ~ 2 w))

Fig. 1. DEC LINC-8-338 PMS diagram.

to its contents. (That is, there is a side effect to referencing.) Thus, address integers in the register can select the next member of a vector or string for accessing.

The instruction-set-execution definition can also be presented as a decoding diagram or tree (Fig. 2). Here, each block represents an encoding of bits in the instruction word. A decoding diagram allows one more descriptive dimension than the conventional, linear ISP description, revealing the assignment of bits to the instruction. Figure 2 still requires ISP descriptions for Mp, Mps, the instruction execution, the effective-address calculation, and the interpreter. Diagrams such as Fig. 2 are useful in the ISP design to determine which instruction numbers are to be assigned to names and operations and instructions which are free to be assigned (or encoded).

There are eight basic instructions encoded by 3 bits, that is op⟨0:2⟩ := i⟨0:2⟩, where instruction/i⟨0:11⟩. Each of the first six instructions (where $0 \leq op < 6$) have the 4 address operand determination modes (thus yielding essentially 24 instructions). The first six instructions are:

data transmission: deposit and clear-accumulator/dca
two's complement add to the accumulator/tad

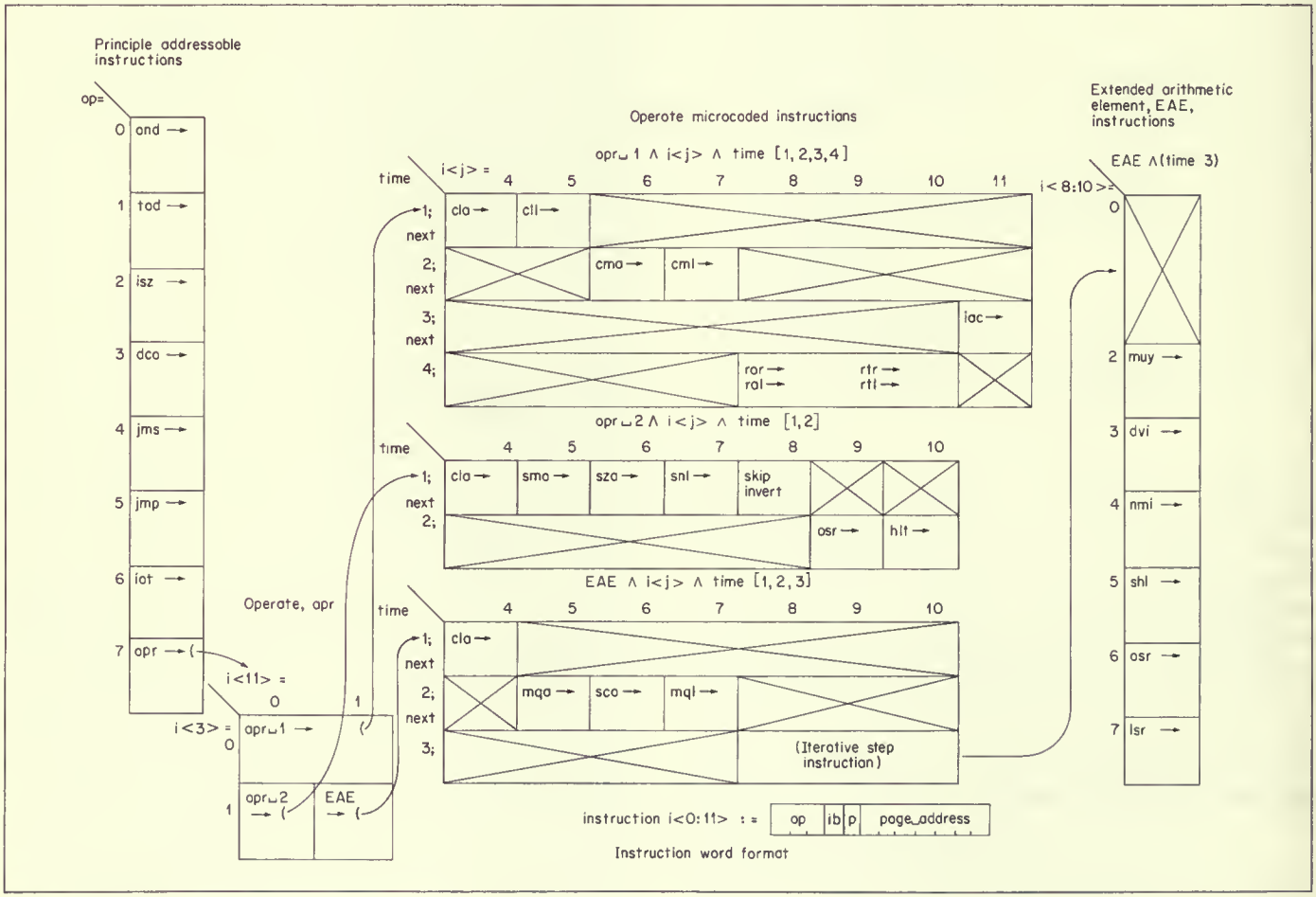


Fig. 2. DEC PDP-8 instruction-decoding diagram.

binary arithmetic:	two's complement add to the accumulator/tad
binary boolean:	and to the accumulator/and
program control:	jump/set program counter/jmp jump to subroutine/jms index memory and skip if results are zero/isz

Note that the add instruction, tad, is used for both data transmission and arithmetic.

The subroutine-calling instruction, jms, provides a method for transferring a link to the beginning (or head) of the subroutine. In this way arguments can be accessed indirectly, and a return is executed by a jump indirect instruction to the location storing the returned address. This straightforward subroutine-call mechanism, although inexpensive to implement, requires reentrant and recursive subroutine calls to be interpreted by software, rather than by hardware. A stack, as in the DEC 338 (Chap. 25), would be nicer.

The input_output instruction/iot (:= op = 6) uses the remaining 9 bits of the instruction to specify instructions to input/output devices. The 6 io_select bits select 1 of 64 devices. The 3 bits, io_p1_bit, io_p2_bit, io_p4_bit, command the selected device by conditionally providing three pulses in sequence. The instructions to a typical io device are:

io_p1_bit → (IO_skip_flag[io select] → (PC ← PC + 1))
testing a condition of an IO device output to a device input from a device

io_p4_bit → (Output_data[io select] ← AC)

io_p2_bit → (AC ← Input_data[io select])

There are three microcoded instruction groups selected by op = 7. The instruction decoding diagram (Fig. 2) and the ISP description (Appendix 1 of this chapter) show the microinstructions which can be combined in a single instruction. These instructions are: operate group 1 (:= (op = 7) ∧ ¬ i⟨3⟩) for operating on the processor state; operate group 2 (:= (op = 7) ∧ (i⟨3,11⟩ = 10_2)) for testing the processor state; and the extended arithmetic element group (:= ((op = 7) ∧ (i⟨3,11⟩ = 11_2))) for multiply, divide, etc. Within each instruction the remaining bits, ⟨4:10⟩ or ⟨4:11⟩, are extended instruction (or opcode) bits; that is, the bits are microcoded to select instructions. In this way an instruction is actually programmed (or microcoded). For example, the instruction set_link →L ←1 is formed by coding the two microinstructions, clear link, next, complement link.

opr_1 → (i⟨5⟩ → L ← 0; next
i⟨7⟩ →L ← ¬L)

Thus, in operate group 1, the instructions clear link, complement link, and set link are formed by coding instruction⟨5,7⟩ = 10, 01, and 11, respectively. The operate group 2 instruction is used for testing the condition of the Pc state. This instruction uses bits 5, 6, and 8 to code tests for the accumulator. The AC skip conditions are coded (0 ~ 7) as never, always, =0, ≠0, <0, ≥0, ≤0, and >0. If all the nonredundant and useful variations in the two operate groups were available as separate instructions in the manner of the first seven (dca, tad, etc.), there would be approximately 7 + 12(opr_1) + 10(opr_2) + 6(EAE) = 35 instructions in the PDP-8.

The optional Extended Arithmetic Element/EAE includes additional Multiplier Quotient/MQ and Shift Counter/SC registers and provides the hardwired operations multiply, divide, logical shift left, arithmetic shift, and normalize. The EAE is defined on the last page of Appendix 1.

The interrupt scheme

External conditions in the input/output devices can request that Pc be interrupted. Interrupts are allowed if (Interrupt_state = 1). A request to interrupt clears Interrupt_state (Interrupt_state ← 0), and Pc behaves as though a jump to subroutine 0 instruction, jms 0, had been given. A special iot instruction (instruction = 6001_8) followed by a jump to subroutine indirect to 0 instruction (instruction = 5200_8) returns Pc to the interruptable state with Interrupt_state = 1. The program time to save M(processor state/ps) is 6 Mp accesses (9 microseconds), and the time to restore Mps is 9 Mp accesses (13.5 microseconds).

Only one interrupt level is provided in the hardware. If multiple priority levels are desired, programmed polling is required. Most io devices have to interrupt because they do not have a program-controlled enable switch for the interrupt. For multiple devices approximately 3 cycles (4.5 μs) are required to poll each interrupter.

PMS structure

The PMS structure of the LINC-8-338 consisting of a Pc('LINC), Pc('PDP-8), and P.display('338) is shown in Fig. 1. The PDP-8 is just a single Pc. The Pc('LINC) is a very capable Pc with more

instructions than the main Pc. It is available in the structure to interpret programs written for the C('LINC), a computer developed by M.I.T.'s Lincoln Laboratory as a laboratory instrument computer for biomedical and laboratory applications. Because of the rather limited ISP in Pc, one would hardly expect to find all the components present in Fig. 1 in an actual configuration.

The S between the Mp and the Pc allows eight Mp's. This S is actually S('Memory Bus; 8 Mp; 1 Pc; (*P requests*); time-multiplexed; 1.5 μs/w). Thus the switch makes Mp logically equivalent to a single Mp(32768 w). There are two other L's which are connected to the Pc, excluding the T.console. They are L('I/O Bus) and L('Data Break; *Direct Memory Access*). These links become switches when we consider the physical structure. Associated with each device is a switch, and the bus links all the devices; the L('I/O Bus) is really an S('I/O Bus). Each time a K connects to it, the S is included in the K. A simplified PMS diagram (Fig. 3) shows the structure and the logical-physical transformation. Thus, the I/O Bus is

S('I/O Bus; duplex; bus; time-multiplexed, 1 Pc; 64 K; *Pc controlled, K requests;* t:4.5 μs/w)

The S('I/O Bus) is the same for the PDP-5, 8, 8/S, 8/I, and 8/L. Hence, any K can be used on any of the above C's. The I/O Bus is the link to the K's for Pc-controlled data transfers. Each word transferred is designated by a Pc instruction. However, the I/O Bus allows a K to request Pc's attention via the interrupt request signal. The Pc polls the K's to find the requesting K if multiple interrupt requests occur. A detailed structure of the Pc-Mp (Fig. 4) shows these L('I/O Bus, 'Data Break) connections to the registers and control in the notation used by DEC. This diagram is essentially a functional block diagram.

The S('I/O Bus) in Fig. 1 is only an abstract representation of

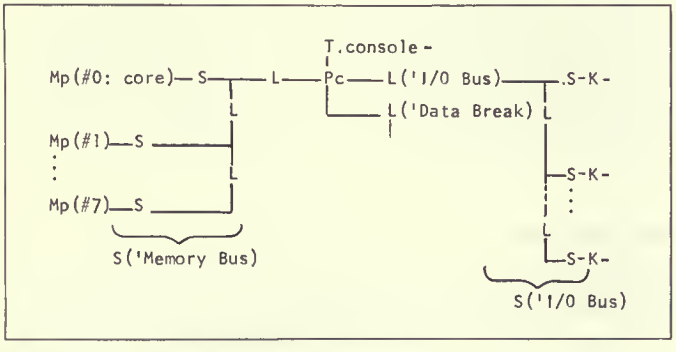


Fig. 3. DEC PDP-8 PMS diagram (simplified).

the structure. Since it is a bus structure, the S can be expanded into L's and simple S's as shown in Fig. 3. The termination of the L in Pc is given in Fig. 3. The corresponding logic at a K is given in Fig. 5 in terms of logic design elements (AND's and OR's). (Fig. 5 also shows the S('I/O Bus) structure of Figs. 1 and 3). The operation of S('I/O Bus) shown in Fig. 5 starts when Pc sends a signal to select (or address) a particular K, using the IO_select ⟨0:5⟩ signals to form a 6-bit code to which K responds. Each K is hardwired to respond to a unique code. The local control, K[j], select signal is then used to form three local commands when ANDed with the three iot command lines from Pc, io_p1_bit, io_p2_bit, and io_p4_bit. Twelve data bits are transmitted either to or from Pc, indirectly under K's control. This is accomplished by using the AND-OR gates in K for data input to Pc, and the AND gate for data input to K. The data lines are connected to AC as shown in Fig. 4. A single *skip* input is used so that Pc can test a status bit in K. A K communicates to Pc via the interrupt request line. Any K wanting attention simply ORs its request signal into the interrupt request signal. Program polling in Pc then selects the specific interrupter. Normally, the K signal causing an interrupt is also connected to the skip input.

The L('Data Break; *Direct Memory Access*) provides a direct access path for a P or K to Mp via Pc. The number of access ports to memory can be expanded to eight by using the S('DM01 Data Multiplexer). The S is requested from a P or K. The P or K supplies an Mp address, a read or write access request, and then either accepts or supplies data for the Mp accessed word. In the configuration (Fig. 1), P('LINC) and P('338) are connected to S('DM01) and make requests to Mp for both their instructions and data in the same way as the Pc. The global control of these processor programs is via the S('I/O Bus). The Pc issues start and stop commands, initializes their state, and examines their final state when a program in the other P halts or requires assistance.

When a K is connected to L('Data Break) or to S('DM01 Data Multiplexer), the K only accesses Mp for data. The most complex function these K's carry out is the transfer of a complete block of data between the Mp and an Ms or a T, for example, K('DECtape, disk). A special mode, the three-cycle data break, is controlled by Pc so that a K may request the next word from a queue in Mp. In this mode the next word is taken from the queue (block) in Mp, and a counter is reduced each time K makes a request. With this scheme, a word transfer takes three Mp cycles: one to add one to the block count, one to add one to the address pointer, and one to transmit the word.

The DECtape was derived from M.I.T.'s Lincoln Laboratory LINCtape unit. Data are explicitly addressed by blocks (variable

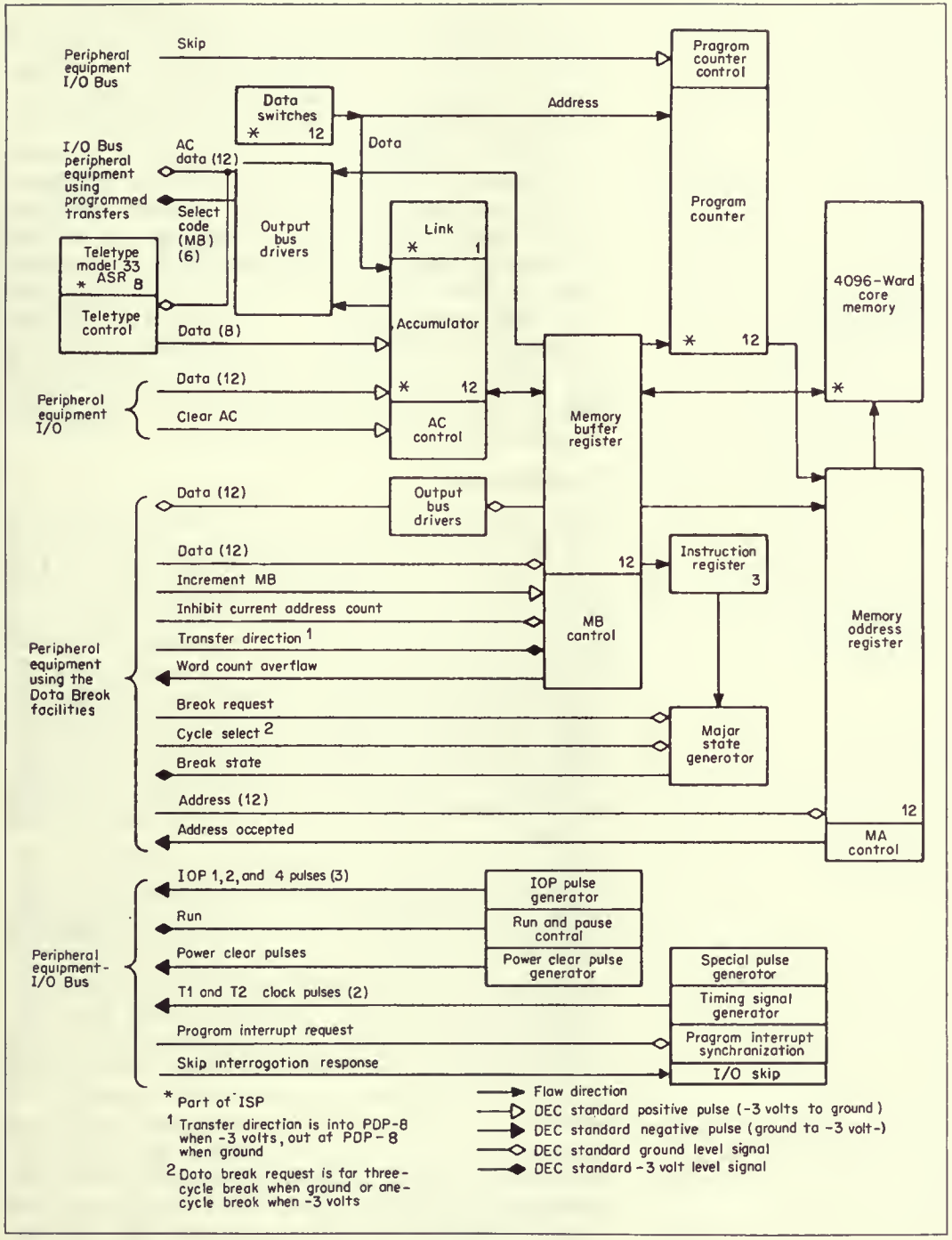


Fig. 4. DEC PDP-8 timing and control-element block diagram.
(*Courtesy of Digital Equipment Corporation.*)

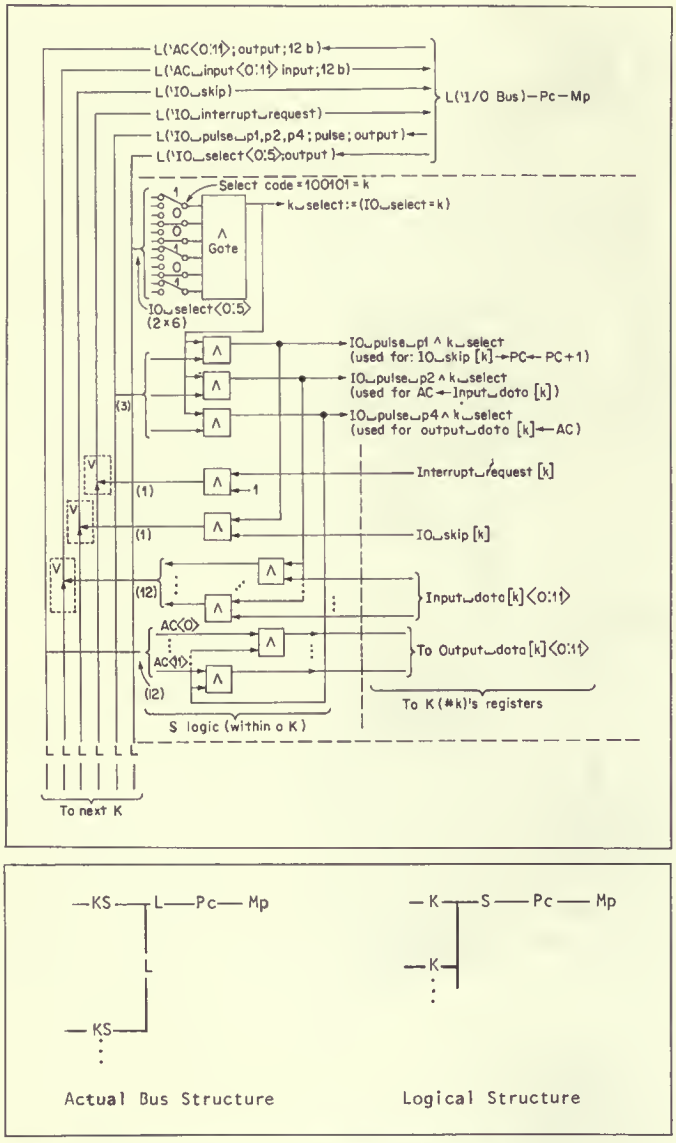


Fig. 5. DEC PDP-8 S('I/O Bus) logic and PMS diagrams.

but by convention 128 w). Thus information in a block can be replaced or rewritten at random. This operation is unlike queue-accessed tape (conventional IBM format magnetic tape) in which data can be appended only to the end of a file.

The control for the T(telephone) links 64 Teletypes or typewriters to the Pc. The final K which connects to a line is on a bit-serial basis. Since a telephone line sends and receives information serially by bit, there are special input/output instructions in the Pc to sample the line and to convert the sampled bits to coded characters. There are 11 bits transmitted per character (although other codings use 7, 7.42, 7.5, and 10 bits per character). Of the 11 bits, there are 3 control, 1 parity, and 7 information bits. The action of the Pc instruction, which is issued 5×11 (55) times for every character, is to control the line by forming the 7-bit characters. The instruction is a good example of tradeoff in the hardware/software domain toward almost pure software; the only hardware state associated with a telephone line is a 1-bit register to hold the state of the outgoing line, and a single AND gate to sample the incoming line state. This sampling process requires about 0.3 per cent of Pc-Mp capacity per active line (each of $10 \sim 15$ char/s). In general, the PDP-8 hardware controls are minimal—in turn fairly elaborate control programs must be used as part of them.

Computer levels

In this section we describe all the systems levels in the PDP-8 computer from the top down. The reader should already have a sketchy knowledge of the PDP-8 because the registers and ISP have been exposed. Here, we wish to clarify how it operates. A map of the hierarchy is given in Fig. 6, starting from PMS to ISP and down through logic design to circuit electronics. These description levels are subdivided to provide more organizational detail. For example, the register-transfer level has the more detailed registers, data operators, functional units, and macro logic of the processor, whereas the next logic level below has sequential and combinational networks, and the sequential and combinatorial elements.

It should be apparent that the relationship of the various description levels constitutes a tree structure where the organizationally complex computer is the top node and each descending description level represents increasing detail (or smaller component size), until the final circuit element level is reached. For simplicity, only a few of the many possible paths through the structural description tree are illustrated. For example, the path showing mechanical parts is missing. The path shown proceeds from the PDP-8 computer to the processor and from there to the arithmetic unit or, more specifically, to the AC register of the arithmetic unit. Next, the macro logic implementing the register-transfer operations and functions for the jth bit of the AC is given; the flip-flops and gates needed for this particular implementation are shown. Finally, on the last segment of the path, come the electronic circuits and components of which flip-flops and NAND gates are constructed.

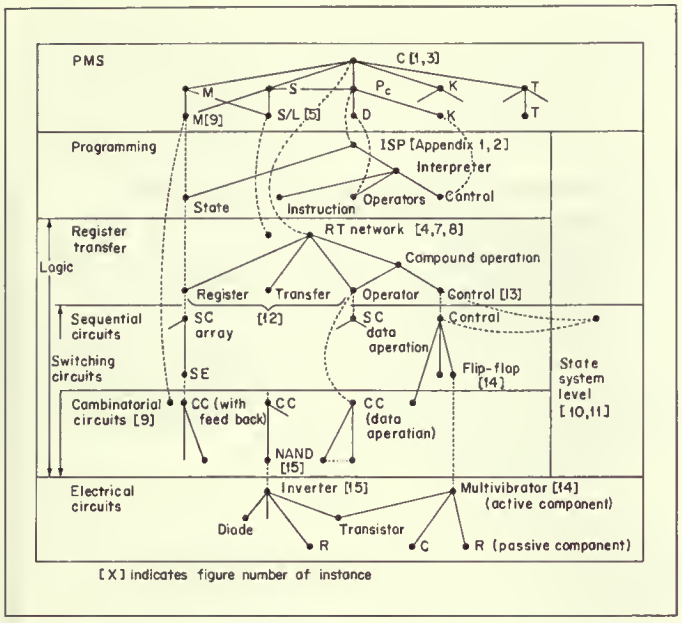


Fig. 6. DEC PDP-8 hierarchy of descriptions.

Abstract representations

Figure 6 also lists some of the methods used to represent the physical computer abstractly at the different description levels. As mentioned previously, only a small part of the PDP-8 description tree is represented here. The many documents, schematics, diagrams, etc., which constitute the complete representation of even this small computer include logic diagrams, wiring lists, circuit schematics and printed-circuit board layout masks, production description diagrams, production parts lists, testing specifications, programs for testing and diagnosing faults, and manuals for modification, production, maintenance, and use. As the discussion continues down the abstract description tree, the reader will observe that the tree conveniently represents the constituent objects of each level and their interconnection at the next highest level. Each level in the abstract-description tree will be described in order.

The PMS level

The simplified PMS structure in Fig. 3 has been reduced from Fig. 1. The computer is small enough so that the physical delineation of the PMS components, such as K's and S's, is less pronounced than in larger systems. In fact, in the case of the S('Memory Bus, 'I/O Bus), the S's are actually within the K and Mp, as shown in Fig. 5. The implementation of these switches within the K and Mp was shown in Fig. 5. In Fig. 7 we present a more conventional functional diagram and the equivalent PMS diagram of the computer, with Pc decomposed into K, processor state (Mps), and D. The functional diagram has the same components of the characteristic elementary computer model, namely, K, D, M, and T(input, output). These figures give a somewhat general idea of what processes can occur in the computer, and how information flows, but it is apparent that at least another level is needed to describe the internal structure and behavior of the Mp and Pc. We should look at these primitives (although still together as a C) at the register-transfer level.

Programming level (ISP)

The ISP interpretation is given in Appendix I of this chapter and is the specification of the programming machine. In addition, it constrains the physical machine's behavior to have a particular ISP. The ISP has been discussed earlier in the chapter.

Register-transfer level

The C can also be represented at the register-transfer level by using PMS. Figure 4 (by DEC) shows the register-transfer level;

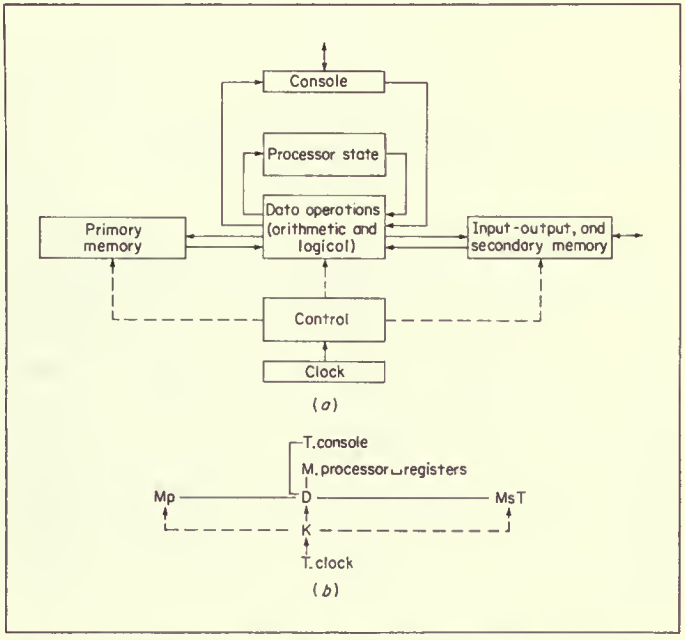


Fig. 7. DEC PDP-8 function block and PMS diagrams. (*a*) Processor functional block diagram. (*b*) Pc PMS diagram.

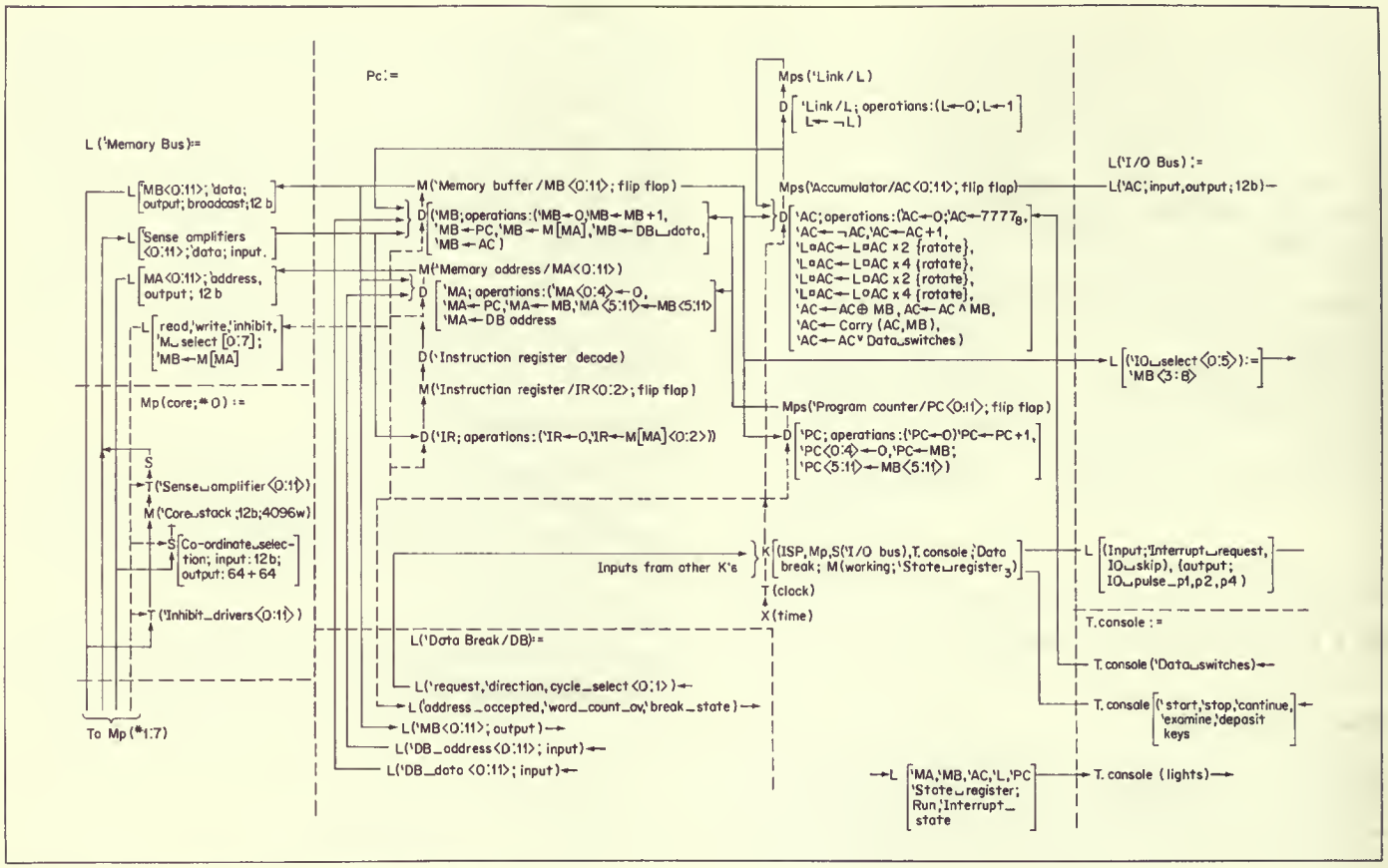


Fig. 8. DEC PDP-8 register-transfer-level PMS diagram.

only registers, operations, and L's are important at this level. We still lack information about the conditions under which operations are evoked. Figure 8 is a PMS diagram of Pc-Mp registers. Here we show considerably more detail (although we do not bother with electrical pulse voltages and polarities) than in Fig. 4. We declare the Pc state (including the temporary register) within Pc. The figure also gives the permissible data operations, D, which are permitted on the registers. It should be clear from this that the logical design level for the registers and the operators can easily be reached. The K logic design cannot be reached until we use the programming level constraints (ISP), thus defining the conditions for evoking the data operators.

The core memory. The Mp structure is given in Fig. 8. A more detailed block diagram which shows the core stack with its twelve 64×64 1-bit core planes is needed. Such a diagram, though still a functional block diagram, takes on some of the aspects of a circuit diagram because a core memory is largely circuit-level details. The Mp (Fig. 9) consists of the component units: the two address decoders (which select 1 each of 64 outputs in the X and Y axis directions of the coincident current memory); selection switches (which transform a coincident logic address into a high-current path to switch the magnetic cores); the 12 inhibit drivers (which switch a high current or no current into a plane when either a 0 or 1 is rewritten); 12 sense amplifiers (which take the induced low sense voltage from a selected core from a plane being switched or not switched and transform it into a 1 or 0); and the core stack, an array $M[0{:}7777_8]\langle 0{:}11\rangle$. Since this is the only time the Mp is mentioned, Fig. 9 also includes the associated circuit-level hardware needed in the core-memory operation, such as

power supplies, timing, and logic signal level conversion amplifiers. The timing signals are generated within Pc(K) and are shown together with Pc's clock in Fig. 10.

The process of reading a word from memory is:

1 A 12-bit selection address is established on the MA⟨0:11⟩ address lines, which is 1 of 10000_8 (or 4096_{10}) unique numbers. The upper 6 bits, ⟨0:5⟩, select 1 of 64 groups of Y addresses and the lower 6 bits, ⟨6:11⟩, select 1 of 64 groups of X addresses.

2 The read logic signal is made a 1.

3 A high-current path flows via the X and Y selection switches. In each of the X and Y directions 64 × 12 cores have selection current. Only one core in each plane is selected since Ix = Iy = Iswitching/2, and the current at the selected intersection = Ix + Iy = Iswitching.

4 If a core is switched to 0 (by having Iswitching amperes through it), then a 1 was present and is read at the output of the plane (bit) sense amplifiers. A sense amplifier receives an input from a winding that threads every core of every bit within a core plane [$0:7777_8$]. All 12 cores of the selected word are reset to 0. The sense time at which the sense amplifier is observed is tms (memory strobe), and the strobe in effect creates MB ← M[MA].

5 The read current is turned off.

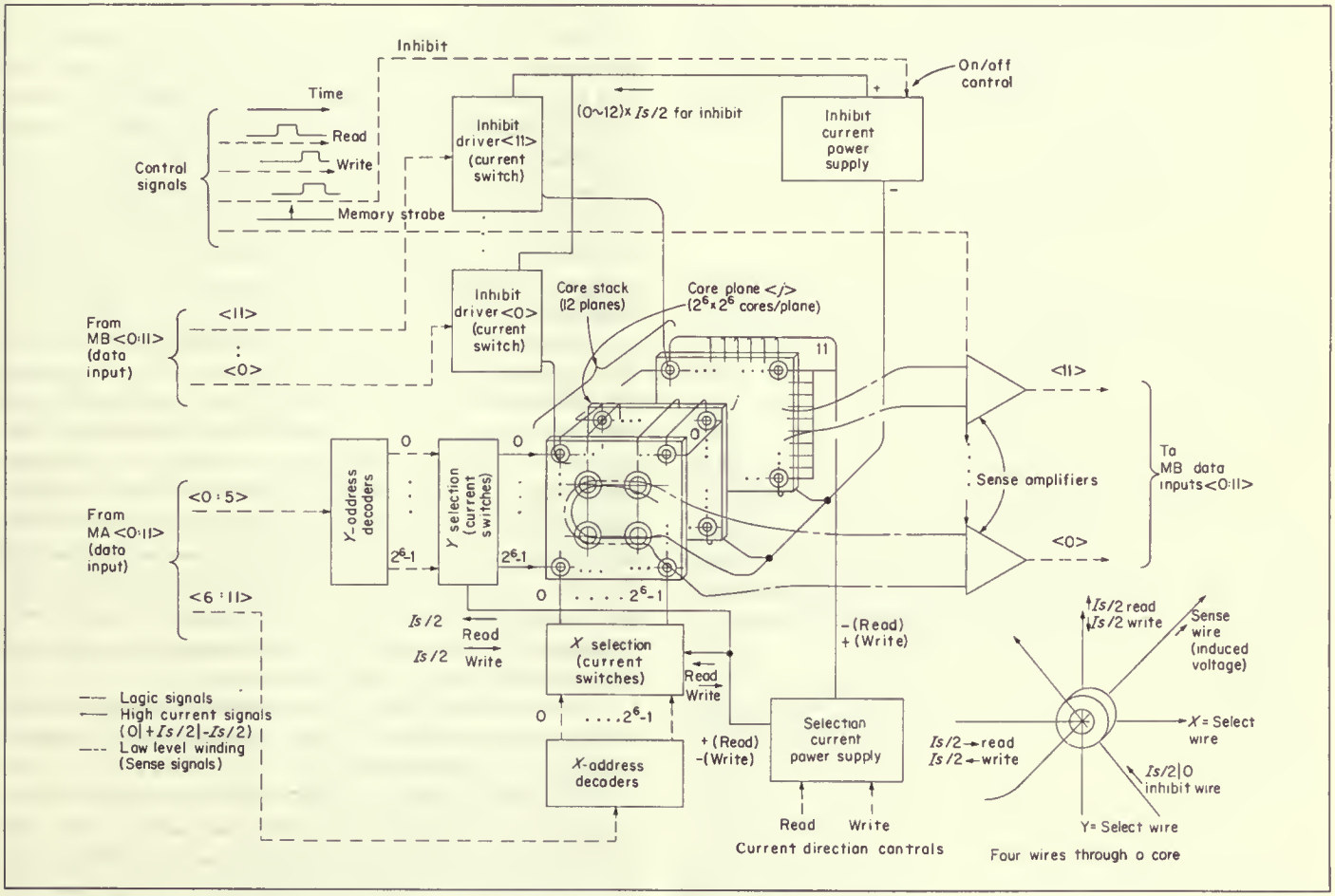


Fig. 9. DEC PDP-8 four-wire coincident current (three dimensions) core-memory-logic block diagram.

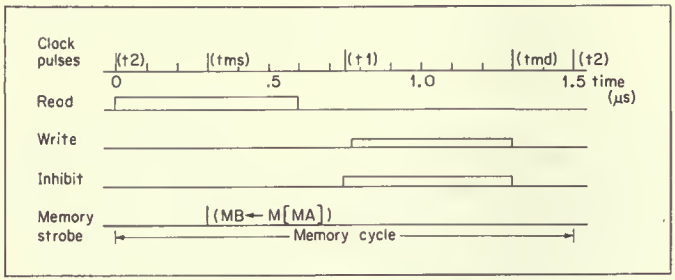


Fig. 10. DEC PDP-8 clock and memory timing diagram.

6 The write and inhibit logic signals are turned on. The bit inhibit signal is present or not, depending on whether a 0 or I, respectively, is written into a bit.

7 A high-current path flows via the X and Y selection switches, but in an opposite direction to the read case (2 above). If a I is written, no inhibit current is present, and the net current in the selected core is −Iswitching. If a 0 is written, the current is −Iswitching +(Iswitching/2) and the core remains reset.

8 The inhibit and write logic signals are turned off, and the memory cycle is completed.

Registers and operations. As Fig. 8 shows, the registers in the Pc cannot be uniquely assigned to a single function. In a minimal machine such as the PDP-8, functional separation is not economical. Thus there are not completely distinct registers and transfer paths for memory, arithmetic, and program and instruction flow. (This sharing complicates understanding of the machine.) However, Fig. 8 clarifies the structure considerably by defining all the registers in Pc (including temporaries). For example, the Memory Buffer/MB is used to hold the word being read from or written to Mp. MB also holds one of the operands for binary operations (for example, AC ← AC ∧ MB). MB is also used as an extension of the Instruction Register/IR during the instruction interpretation.

The additional registers, not in the ISP, are:

Memory Buffer/MB⟨0:11⟩	holds memory data, instruction, and operands
Memory Address/MA⟨0:11⟩	holds address of word in Mp being accessed
Instruction Register/IR⟨0:2⟩	holds the value of current instruction being performed
$State\_register_3$	a ternary state register holding the major state of memory cycle being performed
Fetch/F := (State_register = 0)	memory cycle to fetch instruction
Defer/D/Indirect := (State_register = 1)	memory cycle to get address of operand
Execute/E := (State_register = 2)	memory cycle to fetch (store) operand and execute the instruction

Figure 8 has been concerned with the static definition (or declaration) of the information paths, the operations, and state. The ISP interpretation (Appendix 1) is the specification for the physical machine's behavior. As the temporary hardware registers are added, a more detailed ISP definition could be given in terms of time and temporary registers. Instead, we give a state diagram (Fig. 11) to define the actual Pc which is constrained by both the ISP registers, the temporary registers implied by the implementation, and time. The relationship among the state diagram, the ISP description, and the logic is shown in the hierarchy of Fig. 6. In the relationships of the figures, we observe that the ISP definition does not have all the necessary detail for fully defining a physical Pc. The physical Pc is constrained by actual hardware logic and lower-level details even at the circuit level. For example, a core memory is read by a destructive process and requires a temporary register (MB) to hold the value being rewritten. This is not representable within a single ISP language statement since we define only the nondestructive transfer ←, but it can be considered as the two parallel operations MB ← M[MA]; M[MA] ← 0. The problem of explaining rewriting of core using ISP is also difficult, because explicit time is not in the ISP language (although we can define clock events, or at least relative time).

The state diagram (Fig. 11) describes the implementation behavior using the registers and register operations (Fig. 8) and the temporary registers declared above.

The implementation is fundamentally Mp-timing-based, as we see from both the state diagram and the times when the four clock signals are generated (Fig. 10). Thus there are three (State_register = 0,1,2) × 4 (clock), that is, 12 major states, in the implementation. We use the IR to obtain two more states, F2b and F3b,

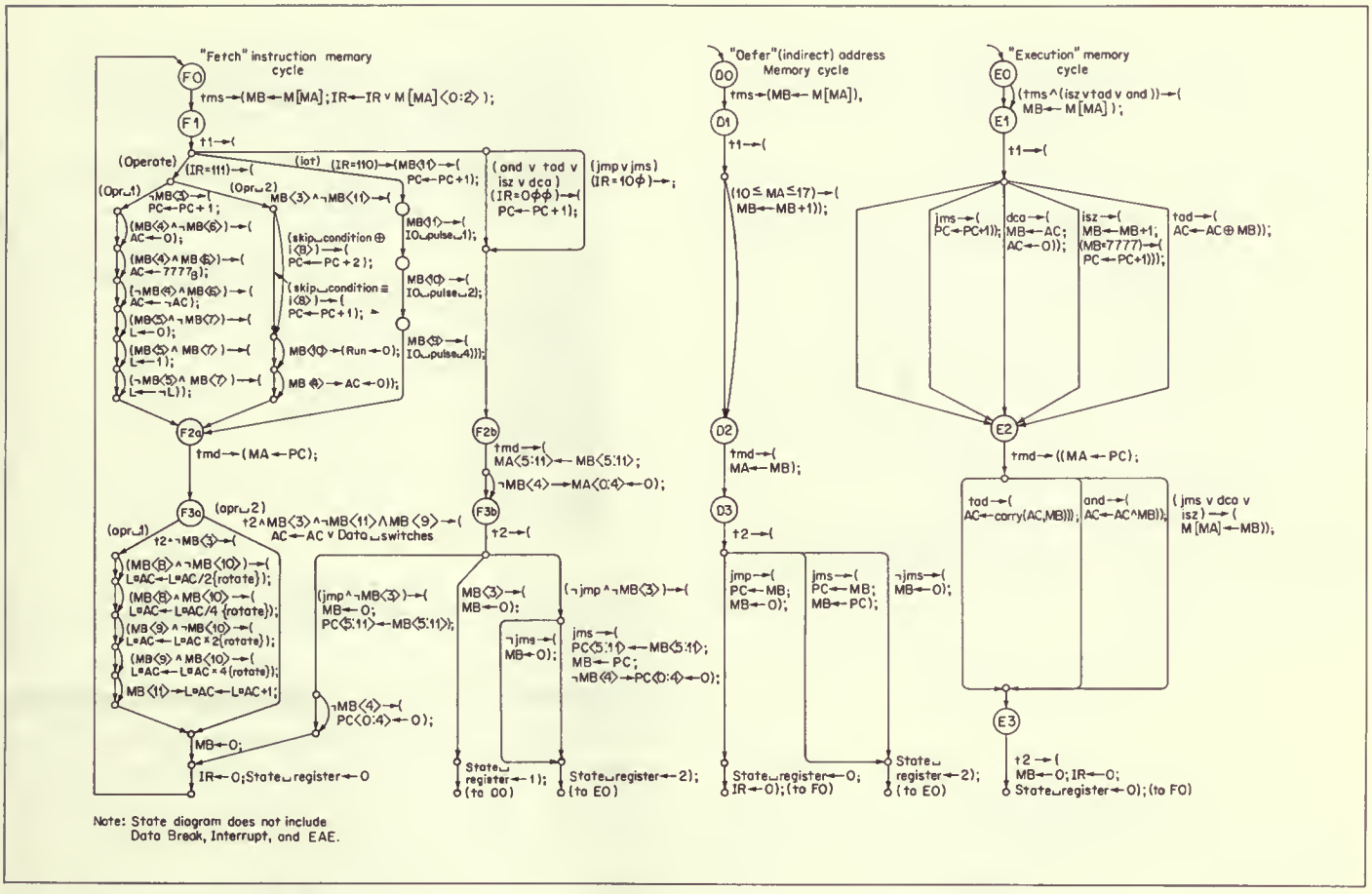


Fig. 11. DEC PDP-8 Pc state diagram.

for the description. The State_register values 0, 1, and 2 correspond to fetching, deferring (indirect addressing, i.e., fetching an operand address), and executing (fetching or storing data, then executing) the instruction. The state diagram does not describe the Extended Arithmetic Element/EAE operation, the interrupt state, and the data break states (these add 12 more states). The initialization procedure, including the T.console state diagram, is also not given. One should observe that when t2 occurs at the beginning of the memory cycle, a new State_register value is selected. The State_register value is always held for the remainder of the cycle; i.e., only the sequences (F0 → F1 → F2 → F3 or D0 → D1 → D2 → D3 or E0 → E1 → E2 → E3) are permitted.

Figure 8 alludes to Pc(K), that is, the sequential network used for controlling Pc. The inputs and the present state (including clocks) determine the operations to be issued on the registers.

Logic design level (registers and data operations)

Proceeding from the register-transfer and ISP descriptions, the next level of detail is the logic module. Typical of the level is the 1-bit logic module for an accumulator bit, AC⟨j⟩, illustrated in Fig. 12. The horizontal data inputs in the figure are to the logic module from AC⟨j⟩, MB⟨j⟩, IO Bus⟨j⟩, and Data_switch⟨j⟩. The vertical control signal inputs command the register operations (i.e., the transfers); they are labeled by their respective ISP operations (for example, AC ← MB ∧ AC, AC ← AC × 2 {rotate}). The sequential network Pc(K) (Fig. 8) generates these control signal inputs.

Logic design level (Pc control, Pc(K) sequential network)

The output signals from the Pc(K) (Fig. 8) can be generated in a straightforward fashion by formulating the boolean expressions

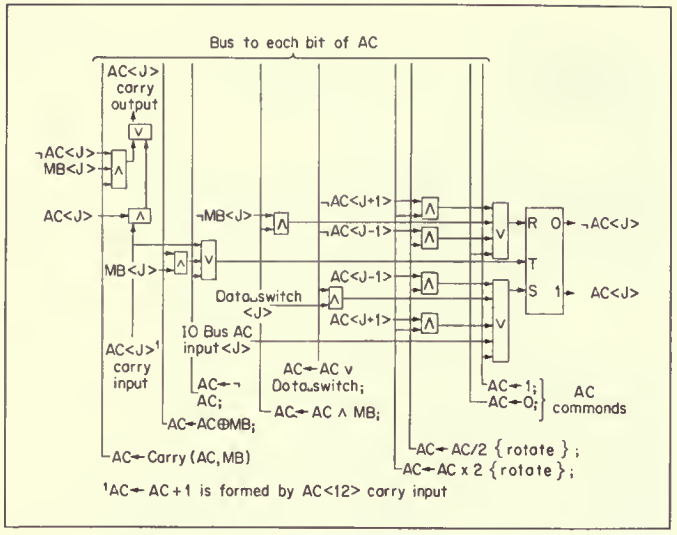


Fig. 12. DEC PDP-8 AC⟨J⟩ bit register-transfer logic diagram.

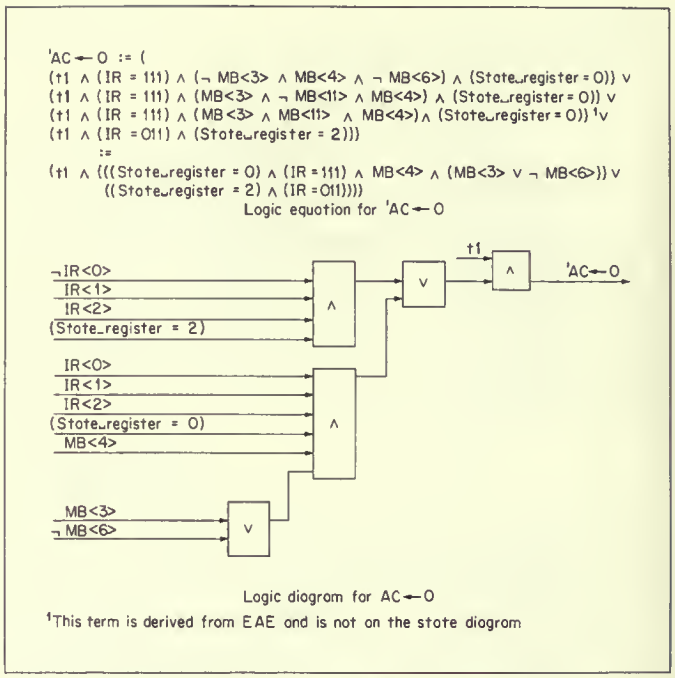
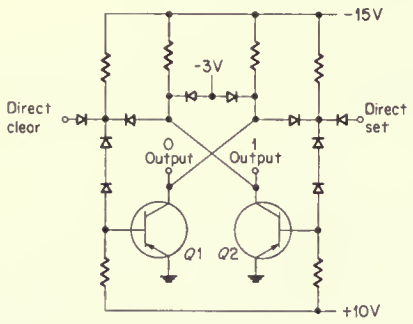
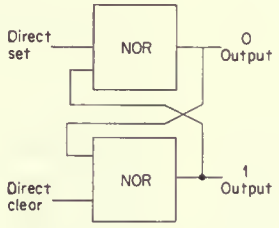


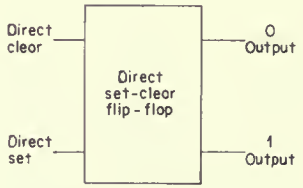
Fig. 13. DEC PDP-8 Pc(K) 'AC ← 0 signal-logic equations and diagram.



Flip-flop circuit



Combinatorial logic equivalent of flip-flop



Direct set-cleor flip-flop sequential logic element

Table of circuit input-output

Outputs (at t)		Inputs		Outputs (at t^+)[1]	
1	0	Direct set	Direct cleor	1	0
0	-3	-3	-3	0	-3
-3	0	-3	-3	-3	0
-3	0	-3	0	-3	0
0	-3	-3	0	-3	0
-3	0	0	-3	0	-3
0	-3	0	-3	0	-3

Table of flip-flop input-output

Outputs (at t)		Inputs		Outputs (at t^+)[1]	
1	0	Direct set	Direct cleor	1	0
1	0	0	0	1	0
0	1	0	0	0	1
0	1	0	1	0	1
1	0	0	1	0	1
0	1	1	0	1	0
1	0	1	0	1	0

[1]*Note*: This is not an "ideal" sequential circuit element, because there is no delay in the output.

Fig. 14. DEC PDP-8 sequential-element circuit and logic diagrams.

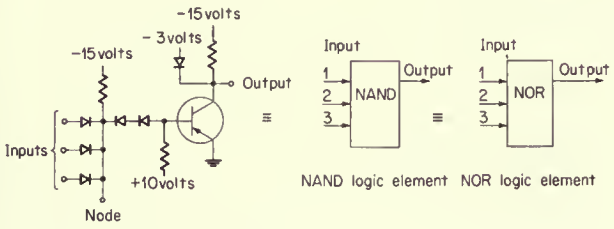


Table of circuit behavior

Input 1	2	3	Output
0	0	0	-3
0	0	-3	-3
0	-3	0	-3
0	-3	-3	-3
-3	0	0	-3
-3	0	-3	-3
-3	-3	0	-3
-3	-3	-3	0

Table of NAND behavior

Input 1	2	3	Output
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Table of NOR behavior

Input 1	2	3	Output
1	1	1	0
1	1	0	0
1	0	1	0
1	0	0	0
0	1	1	0
0	1	0	0
0	0	1	0
0	0	0	1

Fig. 15. DEC PDP-8 combinational element circuit and logic diagrams.

directly from the state diagram in Fig. 11. For example, the AC ← 0 control signal is expressed algebraically and with a combinatorial network in Fig. 13. Obviously these boolean output control signals are functions which include the clock, the State_register, and the states of the arithmetic registers (for example, A = 0, L = 0, etc.). The expressions should be factored and minimized so as to reduce the hardware cost of the control for the interpreter. Although we are rather cavalier about Pc(K), it constitutes about one-half the logic within Pc.

Circuit level

The final level of description is the circuits which form the logic functions of storage (flip-flops) and gating (NAND gates). Figures 14 and 15 illustrate some of these logic devices in detail.

In Fig. 14 a direct set and direct clear flip-flop, a sequential-logic element, is described in terms of circuit implementation, combinational logic equivalent, a table of its behavior, and its algebraic behavior. Note that this is not an ideal element, because it has no delay and responds directly and immediately to an input. Some idealized sequential logic elements are used in the PDP-8 (but not illustrated), including the RS (Reset-Set), T(Trigger), JK, and D(Delay). A delay in the flip-flops makes them behave in the same way as the ideal primitives in sequential-circuit theory. The outputs require a series delay, Δt, such that, if the inputs change at time t, the outputs will not change until $t + \Delta t$. In fact, the PDP-8 uses capacitor-diode gates at the flip-flop inputs to delay the inputs.

Figure 15 illustrates the combinatorial logic elements used in the PDP-8. The circuit selection is limited to the inverter circuit with single or multiple inputs. These are more familiarly called NAND gates or NOR gates, depending on whether one uses positive and/or negative logic-level definitions.

Conclusion

We could continue to discuss the behavior of the transistor as it is used in these switching-circuit primitives but will leave that to books on semiconductor electronics and physics. It is hoped that the student has gained a grasp of how to think about the hierarchical decomposition of computers into particular levels of analysis (and synthesis).

APPENDIX 1 DEC PDP-8 ISP DESCRIPTION

```
Appendix 1

DEC PDP-8 ISP Description

Pc State
   AC<0:11>                                   Accumulator
   L                                          Link bit/AC extension for overflow and carry
   PC<0:11>                                   Program Counter
   Run                                        1 when Pc is interpreting instructions or "running"
   Interrupt␣state                            1 when Pc can be interrupted; under programmed control
   IO␣pulse␣1; IO␣pulse␣2; IO␣pulse␣4         IO pulses to IO devices

Mp State
Extended memory is not included.
   M[0:7777₈]<0:11>
      Page␣0[0:177₈]<0:11> := M[0:177₈]<0:11>     special array of directly addressed memory registers
      Auto␣index[0:7]<0:11> := Page␣0[10₈:17₈]<0:11>   special array when addressed indirectly,is incremented by 1

Pc Console State
Keys for start, stop, continue, examine (load from memory), and deposit (store in memory) are not included.
   Data switches<0:11>                        data entered via console

Instruction Format
   instruction/i<0:11>
      op<0:2>              := i<0:2>          op code
      indirect␣bit/ib      := i<3>            0, direct; 1 indirect memory reference
      page␣0␣bit/p         := i<4>            0 selects page 0; 1 selects this page
      page␣address<0:6>    := i<5:11>
      this␣page<0:4>       := PC'<0:4>
      PC'<0:11>            := (PC<0:11> -1)
      IO␣select<0:5>       := i<3:8>          selects a T or Ms device
      io␣p1␣bit            := i<11>           these 3 bits control the selective generation of -3 volts,
      io␣p2␣bit            := i<10>            0.4 μs pulses to I/O devices
      io␣p4␣bit            := i<9>
      sma                  := i<5>            μ bit for skip on minus AC, operate 2 group
      sza                  := i<6>            μ bit for skip on zero AC
      snl                  := i<7>            μ bit for skip on non zero Link

Effective Address Calculation Process
   z<0:11> := (                               effective
      ¬ib → z'';
      ib ∧ (10₈ ≤ z'' ≤ 17₈) → (M[z''] ← M[z''] + 1; next);     auto indexing
      ib → M[z''])
   z'<0:11> := (¬ ib → z''; ib → M[z''])
   z''<0:11> := (page␣0␣bit → this␣page□page␣address;        direct address
                 ¬page␣0␣bit → 0□page␣address)

μ microcoded instruction or instruction bit(s) within an instruction
```

APPENDIX 1 DEC PDP-8 ISP DESCRIPTION (Continued)

```
Instruction Interpretation Process
   Run ∧ ¬ (Interrupt␣request ∧ Interrupt␣state) → (          no interrupt interpreter
      instruction ← M[PC]; PC ← PC + 1; next                  fetch
      instruction␣execution);                                 execute
   Run ∧ Interrupt␣request ∧ Interrupt␣state → (              interrupt interpreter
      M[0] ← PC; Interrupt␣state ← 0; PC ← 1)

Instruction Set and Instruction Execution Process
   Instruction␣execution := (
      and (:= op = 0) → (AC ← AC ∧ M[z]);                     logical and
      tad (:= op = 1) → (L□AC ← L□AC + M[z]);                 two's complement add
      isz (:= op = 2) → (M[z'] ← M[z] + 1; next               index and skip if zero
                         (M[z'] = 0) → (PC ← PC + 1));
      dca (:= op = 3) → (M[z] ← AC; AC ← 0);                  deposit and clear AC
      jms (:= op = 4) → (M[z] ← PC; next PC ← z + 1);         jump to subroutine
      jmp (:= op = 5) → (PC ← z);                             jump
      iot (:= op = 6) → (                                     μ in out transfer, microprogrammed to generate up to 3 pulses
         io␣p1␣bit → IO␣pulse␣1 ← 1; next                       to an io device addressed by IO␣select
         io␣p2␣bit → IO␣pulse␣2 ← 1; next
         io␣p4␣bit → IO␣pulse␣4 ← 1);
      opr (:= op = 7) → Operate␣execution                     the operate instruction is defined below
                  )                                           end Instruction execution

Operate Instruction Set
The microprogrammed operate instructions: operate group 1, operate group 2, and extended arithmetic are defined as a separate
 instruction set.
   Operate␣execution := (
      cla (:= i<4> = 1) → (AC ← 0);                           clear AC. Common to all operate instructions.
      opr␣1 (:= i<3> = 0) → (                                 operate group 1
         cll (:= i<5> = 1) → (L ← 0); next                    μ clear link
         cma (:= i<6> = 1) → (AC ← ¬ AC);                     μ complement AC
         cml (:= i<7> = 1) → (L ← ¬ L); next                  μ complement L
         iac (:= i<11> = 1) → (L□AC ← L□AC + 1); next         μ increment AC
         ral (:= i<8:10> = 2) → (L□AC ← L□AC x 2 {rotate});   μ rotate left
         rtl (:= i<8:10> = 3) → (L□AC ← L□AC x 2^2 {rotate}); μ rotate twice left
         rar (:= i<8:10> = 4) → (L□AC ← L□AC / 2 {rotate});   μ rotate right
         rtr (:= i<8:10> = 5) → (L□AC ← L□AC / 2^2 {rotate})); μ rotate twice right
      opr␣2 (:= i<3,11> = 10) → (                             operate group 2
         skip condition ⊕ (i<8> = 1) → (PC ← PC + 1); next    μ AC,L skip test
            skip condition := ((sma ∧ (AC < 0)) ∨ (sza ∧ (AC = 0)) ∨ (snl ∧ L))
         osr (:= i<9> = 1) → (AC ← AC ∨ Data switches);       μ "or" switches
         hlt (:= i<10> = 1) → (Run ← 0));                     μ halt or stop
      FAE (:= i<3,11> = 11) → EAF␣instruction␣execution)      optional EAE description
```

APPENDIX 1 DEC PDP-8 ISP DESCRIPTION (Continued)

```
KT and KMs State
 Each K may have any or all of the following registers.  There can be up to 64 optional K's.
   Input_data[0:77₈]<0:11>                                  64 input buffers
   Output_data[0:77₈]<0:11>                                 64 output buffers
   IO_skip_flag[0:77₈]                                      64 test conditions
   IO_interrupt_request[0:77₈]                              1 signifies a request.  If interrupt_state = 1, then an
                                                             interrupt occurs.
      Interrupt_request := (                                "or" of all requests from each IO device
        max(IO_interrupt_request[0:77₈]))

Extended Arithmetic Element, EAE (optional)
 Provides additional arithmetic instructions (or operators) including x, /, normalize, logical shift and arithmetic shift.
   EAE State
      MQ<0:11>                                              Multiplier Quotient
      SC<0:4>                                               Shift Counter
   Instruction Format and Data
      mds<0:11>                                             multiplier divisor shift data
      s<0:4> := mds<7:11>                                   shift count parameter

Instruction Set for EAE
   EAE_instruction_execution := (next
      mqa (:= i<5>) → (AC ← AC ∨ MQ);                       MQ into AC
      sca (:= i<6>) → (AC ← AC ∨ SC):                       SC into AC
      mql (:= i<7>) → (MQ ← AC; AC ← 0); next               AC into MQ, clear AC
   Note only one of nmi, shl, asr, lsr, muy, or dvi can be given at a time.
      i<8:10> = 00₈ → ;                                     IO operation
      ¬ nmi →(mds ← M[PC]; PC ← PC + 1); next
      muy (:= i<8:10> = 2) → (L□AC□MQ ← MQ x mds; SC ← 0)   multiply
      dvi (:= i<8:10> = 3) → (MQ ← L□AC□MQ/mds;             divide
                              L□AC ← L□AC□MQ mod mds: SC ← 0):
      nmi (:= i<8:10> = 4) → ( AC□MQ ← normalize(AC□MQ);    normalize(AC,MQ) into SC
                               SC ← normalize_exponent(AC□MQ));
      shl (:= i<8:10> = 5) → (L□AC□MQ ← L□AC□MQ x 2^(s+1); SC ← 0);   shift left
      asr (:= i<8:10> = 6) → (L□AC□MQ ← L□AC□MQ / 2^(s+1); SC ← 0);   shift right
      lsr (:= i<8:10> = 7) → (L□AC□MQ ← L□AC□MQ / 2^(s+1){logical};   logical shift
                              SC ← 0)
                              )                             end EAE instruction execution
```

Chapter 8

Structural Levels of the PDP-8[1]

C. Gordon Bell / Allen Newell / Daniel P. Siewiorek

A map of the PDP-8 design hierarchy, based on the Structural Levels View of Chap. 2, is given in Fig. 1, starting from the PMS structure, to the ISP, and down through logic design to circuit electronics. These description levels are subdivided to provide more organizational details such as registers, data operators, and functional units at the register transfer level.

The relationship of the various description levels constitutes a tree structure, where the organizationally complex computer is the top node and each descending description level represents increasing detail (or smaller component size) until the final circuit element level is reached. For simplicity, only a few of the many possible paths through the structural description tree are illustrated. For example, the path showing mechanical parts is missing. The descriptive path shown proceeds from the PDP-8 computer to the processor and from there to the arithmetic unit, or more specifically, to the Accumulator (AC) register of the arithmetic unit. Next, the logic implementing the register transfer operations and functions for the *j*th bit of the Accumulator is given, followed by the flip-flops and gates needed for this particular implementation. Finally, on the last segment of the path, there are the electronic circuits and components from which flip-flops and gates are constructed.

[1]Originally printed in C. G. Bell, J. C. Mudge, and J. E. McNamara, *Computer Engineering: A DEC View of Hardware System Design*, Digital Press, 1978, pp. 209–228.

Abstract Representations

Figure 1 also lists some of the methods used to represent the physical computer abstractly at the different description levels. As mentioned previously, only a small part of the PDP-8 description tree is represented here. The many documents which constitute the complete representation of even this small computer include logic diagrams, wiring lists, circuit schematics, printed circuit board photo etching masks, production description diagrams, production parts lists, testing specifications, programs for testing and diagnosing faults, and manuals for modification, production, maintenance, and use. As the discussion continues down the abstract description tree, the reader will observe that the tree

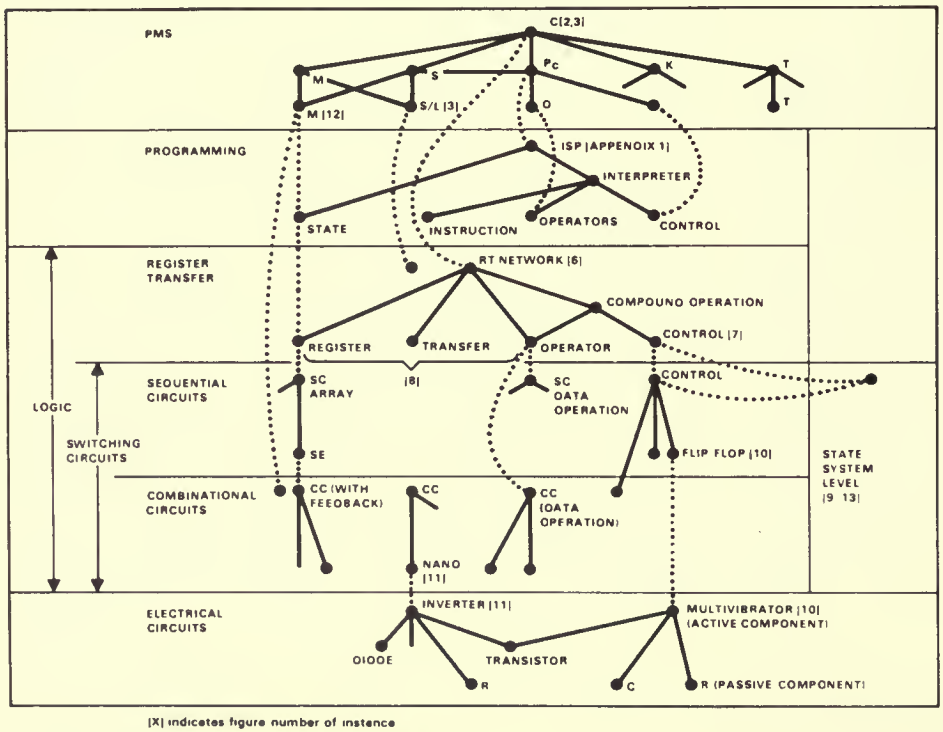


Fig. 1. PDP-8 hierarchy of descriptions.

conveniently represents the constitutent objects of each level and their interconnection at the next highest level.

The PMS Level

The PDP - 8 computer in PMS notation is:

```
C[PDP-8 technology:transistors: 12 b/w:
  descendants:'PDP-8/S, 'PDP-8/I, 'PDP-8/L,
  '8/E, '8/F, '8/M, '8/A, 'CMOS-8;
  antecedents: 'PDP-5;
  Mp[core; #0:7; 4096 words; tc:1.5 μs/word];]
  Pc(Mps(2 to 4 words);
    instruction length:1|2 words;
    address/instruction:1;
    operations on data:(=, +, Not, And, Minus
    (negate), Srr 1(/2), Slr 1 (×2), +)
    optional operations:(×,/,normalize);
    data-types:word,integer,Boolean vector;
    operations for data access:4);
  P(display; '338);
  P(c;'LINC);
  S('I/O Bus; 1 Pc; 64K);
  Ms(disk,'DECtape, magnetic tape);
  T(paper tape, card, analog, cathode-ray tube)
```

As an example of PMS structure, the LINC-8-338 is shown in Fig. 2; it consists of three processors (designated P): Pc('LINC), Pc('PDP-8), and P.display('338). The LINC processor is a very capable processor with more instructions than the PDP-8 and is available in the structure to interpret programs written for the LINC. Because of the rather limited instruction set being interpreted, one would hardly expect to find all the components present in Fig. 2 in an actual configuration.

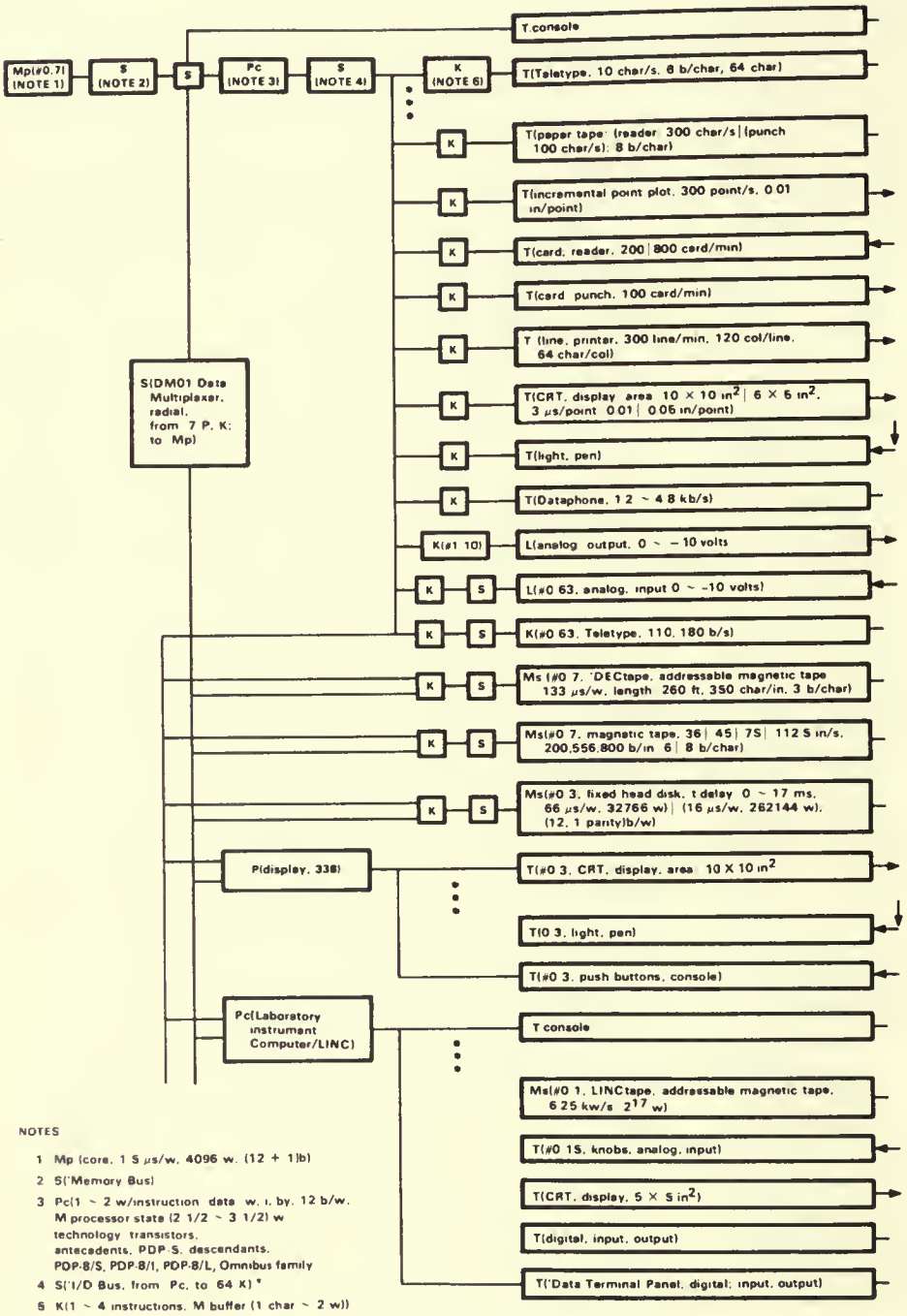
The switches (S) between the memory and the processor allow eight primary memories (Mp) to be connected. This switch, in PMS called S('memory Bus; 8 Mp; 1 Pc; time-multiplexed; 1.5 μs/word), is actually a bus with a transfer rate of 1.5 microseconds per word. The switch makes the eight memory modules logically equivalent to a single 32,768-word memory module. There are two other connections (a switch and a link) to the processor excluding the console. They are the S('I/O Bus) and L('Data Break; Direct Memory Access) for interconnection with peripheral devices. Associated with each device is a switch, and the I/O Bus links all the devices. A simplified PMS diagram (Fig. 3) shows the structure and the logical-physical transformation for the I/O Bus, Memory Bus, and Direct Memory Access link. Thus, the I/O Bus is:

```
S('I/O Bus duplex; time-multiplexed; 1 Pc; 64K;Pc controlled,
K requests; t:4.5 μs/w)
```

The I/O Bus is nearly the same for the PDP-5, 8, 8/S, 8/I, and 8/L. Hence, any controller can be used on any of the above computers provided there is an appropriate logic level converter (PDP-5, 8, and 8/S use negative polarity logic; the 8/I and 8/L, positive logic). The I/O Bus is the link to the controllers for processor-controlled data transfers. Each word transferred is designated by a processor in-out transfer (IOT) instruction. Due to the high cost of hardware in 1965, the PDP-8 I/O Bus protocol was designed to minimize the amount of hardware to interface a peripheral device. As a result, only a minimal number of control signals were defined with the largest portion of I/O control performed by software.

A detailed structure of the processor and memory (Fig. 4) shows the I/O Bus and Data Break connections to the registers and control in the notation used in the initial PDP-8 reference manual. This diagram is essentially a functional block diagram. The corresponding logic for a controller is given in Fig. 3 in terms of logic design elements (ANDs and ORs). The operation of the I/O Bus starts when the processor sends a control signal and sets the six I/O selection lines (IO.SELECT <0:5>) to specify a particular controller. Each controller is hardwired to respond to its unique 6-bit code. The local control, K[k], select signal is then used to form three local commands when ANDed with the three IOT command lines from the processor. These command lines are called IO.PULSE.1, IO.PULSE.2, and IO.PULSE.4. Twelve data bits are transmitted either to or from the processor, indirectly under the controller's control. This is accomplished by using the AND/OR gates in the controller for data input to the processor, and the AND gate for data input to the controller. A single skip input is used so that the processor can test a status bit in the controller. A controller communicates back to the processor via the interrupt request line. Any controller wanting attention simply ORs its request signal into the interrupt request signal. Normally, the controller signal causing an interrupt is also connected to the skip input, and skip instructions are used in the software polling that determines the specific interrupting device.

The Data Break input for Direct Memory Access provides a direct access path for a processor or a controller to memory via the processor. The number of access ports to memory can be expanded to eight by using the DM01 Data Multiplexer, a switch. The DM01 port is requested from a processor (e.g., LINC or Model 338 Display Processor) or a controller (e.g., magnetic tape). A processor or controller supplies a memory address, a read or write access request, and then accepts or supplies data for the accessed word. In the configuration (Fig. 1), Pc['LINC] and P['338] are connected to the multiplexer and make requests to memory for both their instructions and data in the same way as the PDP-8 processor. The global control of these processor programs is via the processor over the I/O Bus. The processor issues start and stop commands, initializes their state, and examines their



NOTES

1 Mp (core, 1.5 μs/w, 4096 w, (12 + 1)b)
2 S('Memory Bus)
3 Pc(1 ~ 2 w/instruction, data w, i, by, 12 b/w, M processor state (2 1/2 ~ 3 1/2) w technology transistors, antecedents, PDP-5, descendants, PDP-8/S, PDP-8/I, PDP-8/L, Omnibus family
4 S('I/O Bus, from Pc, to 64 K)*
5 K(1 ~ 4 instructions, M buffer (1 char ~ 2 w))

Fig. 2. LINC-8-338 PMS diagram.

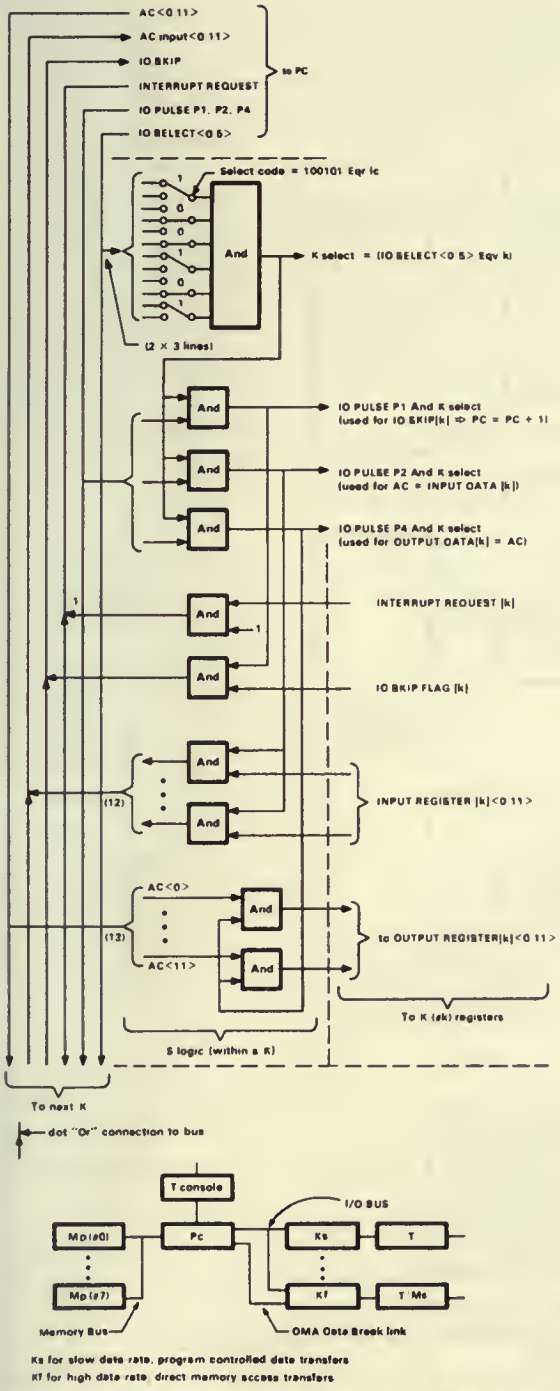


Fig. 3. PDP-8 S('I/O Bus) logic and PMS diagrams.

final state when a program in the other processor halts or requires assistance.

When a controller is connected to the Data Break or to the DM01 Data Multiplexer, it only accesses memory for data. The most complex function these controllers carry out is the transfer of a complete block of data between the memory and a high speed transducer or a secondary memory (e.g., DECtape or disk). A special mode, the Three Cycle Data Break, allows a controller to request the next word from a block in memory.

The DECtape was derived from M.I.T.'s Lincoln Laboratory LINCtape unit. Data were explicitly addressed by blocks (variable but by convention 128 words). Thus, information in a block could be replaced or rewritten at random. This operation was unlike the early standard IBM format magnetic tape in which data could be appended only to the end of a file.

Programming Level (ISP)

The ISP of the PDP-8 processor is probably the simplest for a general purpose stored program computer. It operates on 12-bit words, 12-bit integers, and 12-bit Boolean vectors. It has only a few data operators, namely, =, +, minus (negative of), not, and slr 1 (rotate bits left), srr 1 (rotate bits right), (optional) ×, /, and normalize. However, there are microcoded instructions, which allow compound instructions to be formed in a single instruction.

The ISP of the basic PDP-8 is presented in Appendix 1. The 2^{12}-word memory (declared MP[0:4095]<0:11>) is divided into 32 fixed-length pages of 128 words each (not shown in the ISPS description). Address calculation is based on references to the first page, page.zero, or to the current page of the Program Counter (PC\Program.Counter). The effective address calculation procedure, called MA in Appendix 1, provides for both direct and indirect reference to either the current page or the first page. This scheme allows a 7-bit address to specify a local page address.

A 2^{15}-word memory is available on the PDP-8, but addressing more than 2^{12} words is comparatively inefficient. In the extended range, two 3-bit registers, the Program Field and Data Field registers, select which of the eight 2^{12}-word blocks are being actively addressed as program and data. These are not given in the ISPS description.

There is an array of eight 12-bit registers, called the auto.index registers, which resides in page.zero. This array (auto.index [0:7]<0:11>:=MP[#10: #17]<0:11>) possesses a useful property: whenever an indirect reference is made to it, a 1 is first added to its contents. (That is, there is a side effect to referencing.) Thus, address integers in the register can select the next member of a vector or string for accessing.

The processor state is minimal, consisting of a 12-bit accumulator (AC\Accumulator<0:11>), an accumulator extension bit called the Link (L\Link), the 12-bit Program Counter, the GO flip-flop,

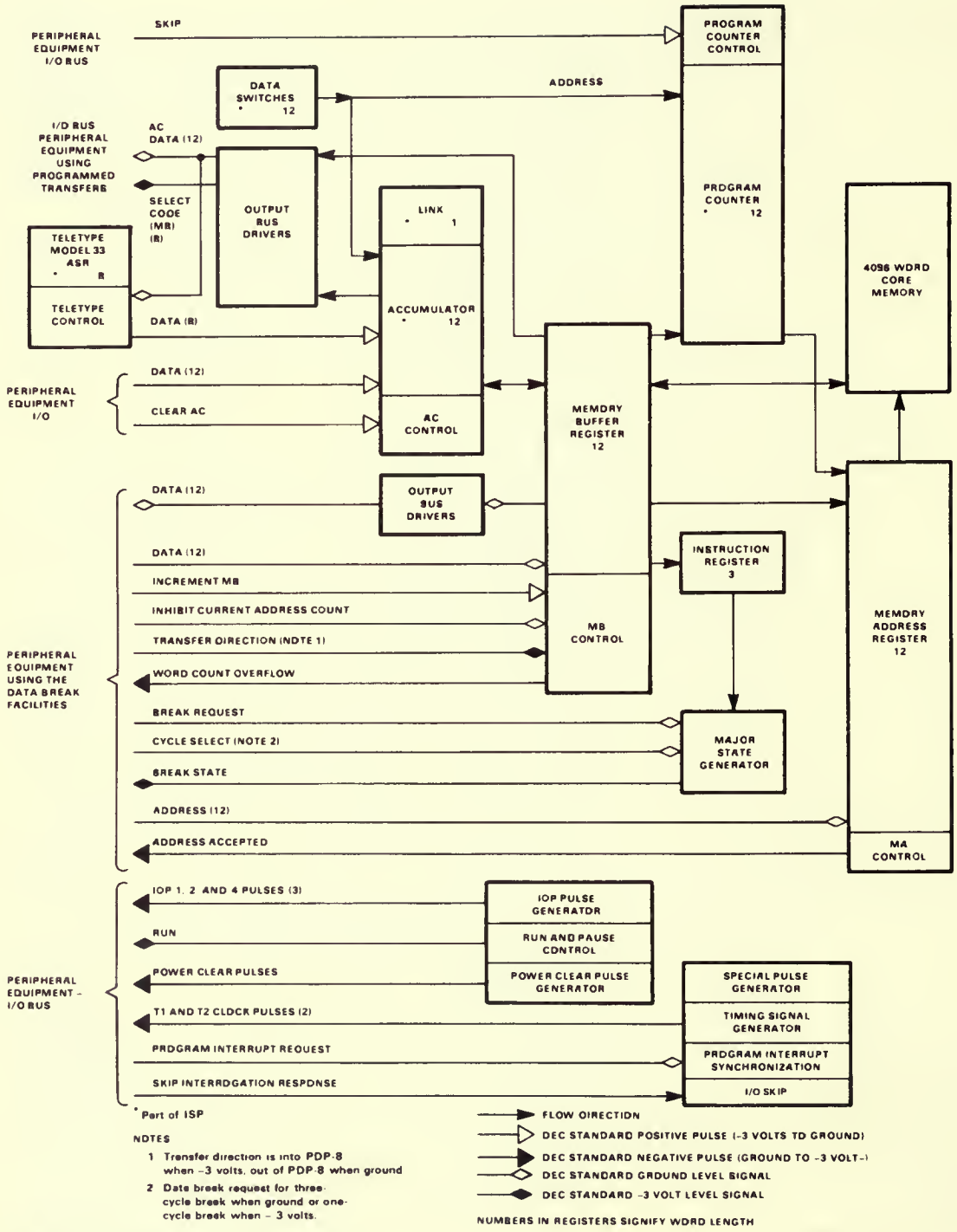


Fig. 4. PDP-8 processor block diagram.

and the INTERRUPT.ENABLE bit. The external processor state is composed of console switches and an interrupt request.

The instruction format can also be presented as a decoding diagram or tree (Fig. 5). Here, each block represents an encoding of bits in the instruction word. A decoding diagram allows one more descriptive dimension than the conventional, linear ISPS description, revealing the assignment of bits to the instruction. Figure 5 still requires ISPS descriptions for the memory, the processor state, the effective address calculation, the instruction interpreter, and the execution for each instruction. Diagrams such as Fig. 5 are useful in the ISP design to determine which instruction operation codes are to be assigned to names and operations, and which instructions are free to be assigned (or encoded).

There are eight basic instructions encoded by 3 opcode bits of the instruction.register, that is, IR<0:2>. Each of the first memory reference six instructions, where the opcode is less than or equal to 5, has four addressing modes (direct page.zero, direct current.page, indirect page.zero, and indirect current.page). The first six instructions in the following four categories are:

1 *Data transmission*
"deposit and clear Accumulator" (DCA). (Note that the add instruction, TAD, is used for both data transmission and arithmetic.)

2 *Binary arithmetic*
"two's complement add to the Accumulator" (TAD).

3 *Binary Boolean*
"and to the Accumulator" (AND).

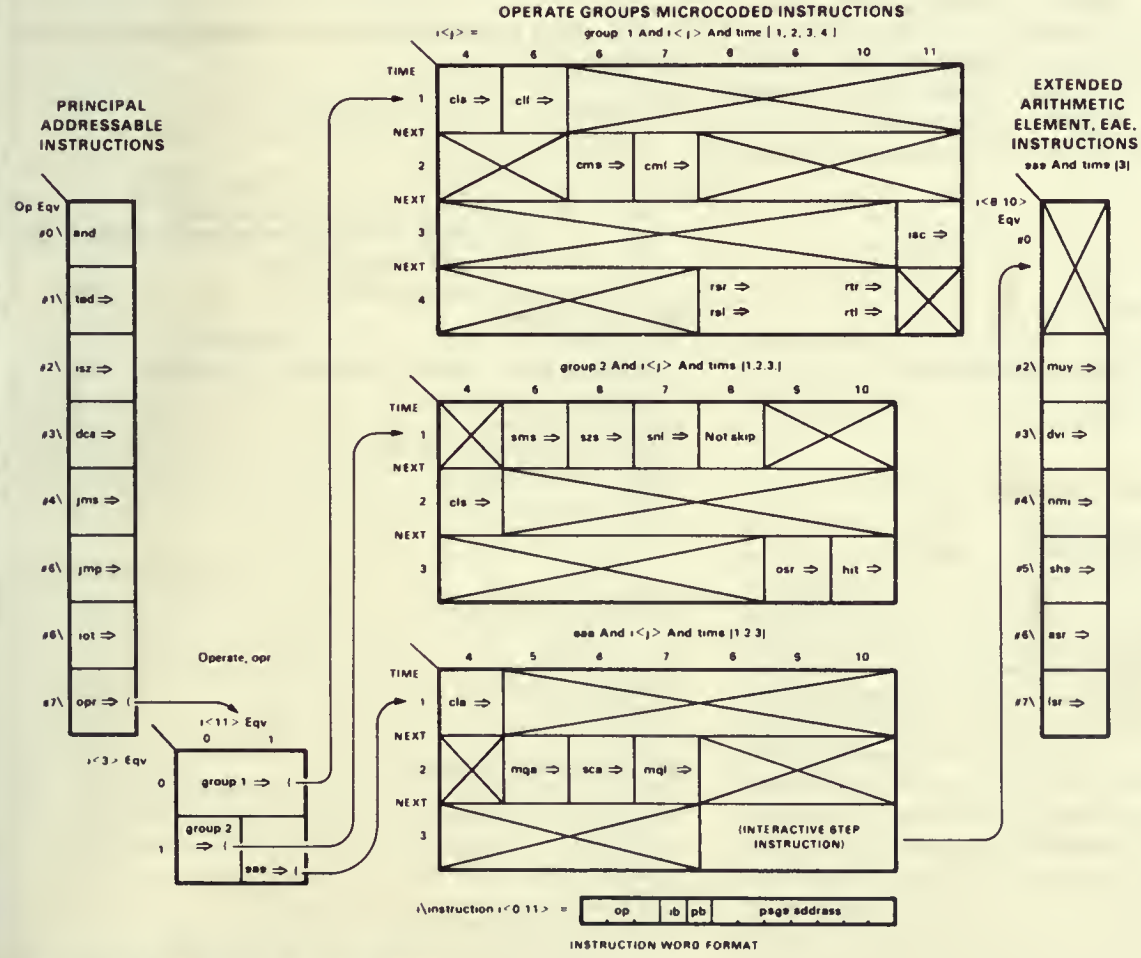


Fig. 5. PDP-8 instruction decoding diagram.

4 *Program control*
"jump/set Program Counter" (JMP); "jump to subroutine" (JMS); "index memory and skip if results are zero" (ISZ).

The subroutine calling instruction, JMS, provides a method for transferring a link to the beginning (or head) of the subroutine. In this way arguments can be accessed indirectly, and a return is executed by a "jump indirect" instruction to the location storing the returned address. This straightforward subroutine call mechanism, although inexpensive to implement, requires reentrant and recursive subroutine calls to be interpreted by software rather than by hardware. A stack for subroutine linkage, as in the PDP-11, would allow the use of read-only memory program segments consisting of pure code. This scheme was adopted in the CMOS-8.

The "in-out transfer" instruction, opcode 6, IOT uses the remaining nine bits of the instruction to specify instructions to input/output devices. The six io.select bits select 1 of 64 devices. Three conditional pulse commands to the selected device, io.pulse.1, io.pulse.2, and io.pulse.4, are controlled by the IOT, io.control<0:2> operation code bits. The instructions to a typical I/O device are:

1 Testing a Boolean Condition of an IO device.
IF io.pulse.1⇒
(IF io.skip.flag[io.select]⇒
PC = PC + 1)

2 Output data to a device from Accumulator.
IF io.pulse.4⇒
(output.register[io.select] =
AC)

3 Input data from a device to Accumulator.
IF io.pulse.2⇒
(AC = input.register[io.select])

There are three microcoded instruction groups selected by (IR<0:2> eqv #7), called the operate instructions. The instruction decoding diagram (Fig. 5) and the ISP description show the microinstructions which can be combined in a single instruction. These instructions are: operate group 1 ((IR<0:2> eqv #7) and not ib) for operating on the processor state; operate group 2 ((IR<0:2> eqv #7) and NOT ib<> and MB<11>) for testing the processor state; and the Extended Arithmetic Element group (not included in the ISP description) (IR<0:2> eqv #7 and ib<> and MB<11>) for multiply, divide, etc. Within each instruction the remaining bits, <4:10> or <4:11>, are extended instruction (or opcode) bits: that is, the bits are microcoded to select additional instructions. In this way, an instruction is actually programmed (or microcoded, as it was originally named before "microprogramming" was used extensively). For example, the instruction, "set link to 1," is formed by coding the two microinstructions, "clear link" following by "complement link."

If (IR <0:2> eqv #7) and (group eqv 0)) ⇒
If MB<5> ⇒ L = 0; next
If MB<7> ⇒ L = not L)

Thus, in operate group 1, the instructions "clear link, complement link, and set link" are formed by coding MB<5,7> = 10,01, and 11, respectively. The operate group 2 instructions are used for testing the condition of the processor state. These instructions use bits 5, 6, and 8 to code tests for the Accumulator. The AC skip conditions are coded as never, always, AC eql 0, AC neg 0, AC lss 0, AC leq 0, AC geq 0 and AC gtr 0. The optional Extended Arithmetic Element (EAE) includes additional Multiplier Quo-

Table 1 PDP-8 Register Transfer Control Signals and Data Break Interface

```
AC/Accumulator, L/Link and combined L, AC LAC
   AC = 0; AC = #7777; AC = not AC; LAC = LAC + 1
   L = 0; L = 1; L = not L;
   LAC = LAC srr 1; LAC = LAC srr 2;  !rotates right
   LAC = LAC slr 1; LAC = LAC slr 2;  !rotates left
   AC = AC or SWITCHES; AC = AC and MB; AC = IO.BUS
   AC = AC xor MB; LAC = carry (AC.MB);
   (note that previous two commands form: LAC = AC + MB).
MB/Memory.Buffer
   MB = MB + 1;                              !Increment
   MB = PC; MB = AC; MB = M[MA]; MB = DB.DATA.  !Set
   MB = 0;
MA/Memory.Address
   MA<0:4> = 0; MA = PC; MA = MB; MA<5:11> = MA<5:11>;
   MA = DB.ADDRESS.
PC/Program.Counter
   PC = 0; PC<0:4> = 0;                      !Clear
   PC = MB; PC<5:11> = MB<5:11>              !Set
   PC = PC + 1                               !Increment
IR/Instruction.Register
   IR = 0;                                   !Clear
   IR = M[MA]<0:2>                           !Load
M/Memory[0:4095]<0:11>
   M[MA] = MB                                !write
   MB = M[MA]                                !read
DB/DATA.BREAK interface
   DB.DATA<0:11>                             ! Input to MB
   DB.ADDRESS<0:11>                          ! Input to MA
   MB<0:11>
   DB.REQUEST                                ! Control inputs to Pc
   DB.DIRECTION
   DB.CYCLE.SELECT<0:11>
   ADDRESS.ACCEPTED                          ! Control outputs from Pc
   WORD.COUNT.OK
   BREAK.STATE
```

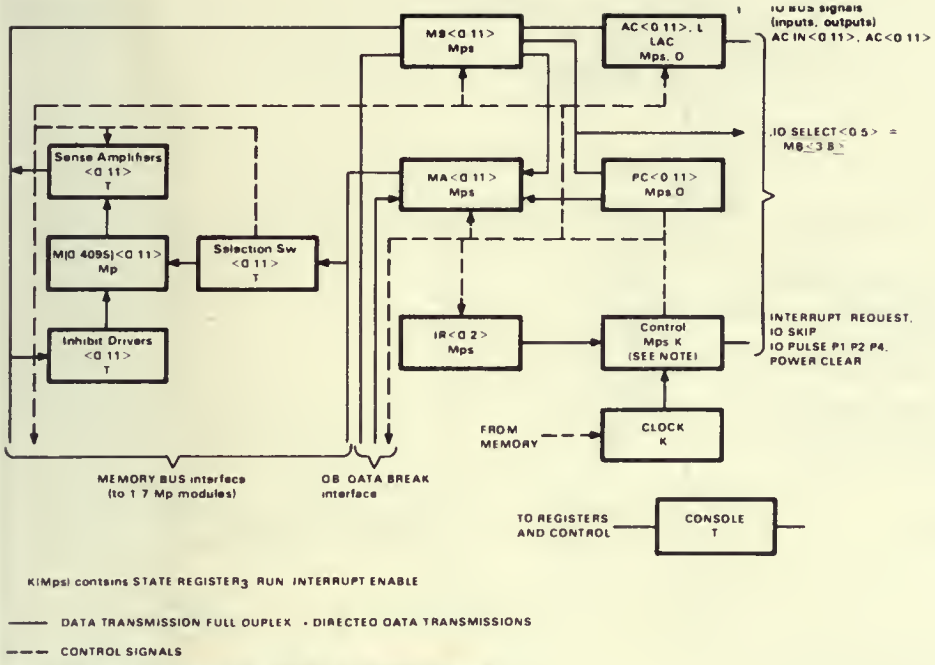


Fig. 6. PDP-8 register transfer level PMS diagram.

tient (MQ) and Shift Counter (SC) registers and provides the hardwired operations, "multiply," "divide," "logical shift left," "arithmetic shift," and "normalize." If all the nonredundant and useful variations in the two operate groups were available as separate instructions in the manner of the first seven (DCA, TAD, etc.), there would be approximately 7 + 12 (group 1) + 10 (group 2) + 6 (EAE) = 35 instructions in the PDP-8.

The Interrupt Scheme

External conditions in the input/output devices can request that the processor be interrupted. Interrupts are allowed if the processor's interrupt enable flip-flop is set (if INTERRUPT. ENABLE eqv 1). A request to interrupt (i.e., INTERRUPT. REQUEST = 1) clears the interrupt enable bit (INTERRUPT. ENABLE = 0), and the processor behaves as though a "jump to subroutine" 0 instruction (JMS 0) has been executed. A special IOT instruction (MB<0:11> eql #6001) followed by a "jump to subroutine indirect" to 0, and instruction (MB<0:11> eql #5220) returns to the processor to the interruptable state with INTERRUPT.ENABLE a 1. The program time to save the processor state is six memory accesses (9 microseconds), and the time to restore the state is nine memory accesses (13.5 microseconds).

Only one interrupt level is provided in the hardware. If multiple priority levels are desired, programmed polling is required. Most I/O devices have to interrupt because they do not have a program-controlled device interrupt-enable switch. For multiple devices, approximately three cycles (4.5 microseconds) are required to poll each interrupter.

Register Transfer Level

More detail is required than is provided by either the PMS or ISP levels to describe the internal structure and behavior of the processor and memory. Figure 4 shows the registers and controllers at a block diagram level, and Fig. 6 gives a more detailed version using PMS notation. Table 1 gives the permissible register transfer operations that the processor's sequential control circuit can give to the PDP-8 registers.

Although electrical pulse voltages and polarities are not shown in Table 1, the operations are presented in considerably more detail than shown in Fig. 4. As Fig. 6 shows, the registers in the processor cannot be uniquely assigned to a single function. In a minimal machine such as the PDP-8, functional separation is not economical. Thus, there are not completely distinct registers and transfer paths for memory, arithmetic, program, and instruction flow. (This sharing complicates understanding of the machine.) However, Fig. 6 clarifies the structure considerably by defining all the registers in the processor (including temporaries and controls). For example, the Memory Buffer (MB\Memory.Buffer<0:11>)

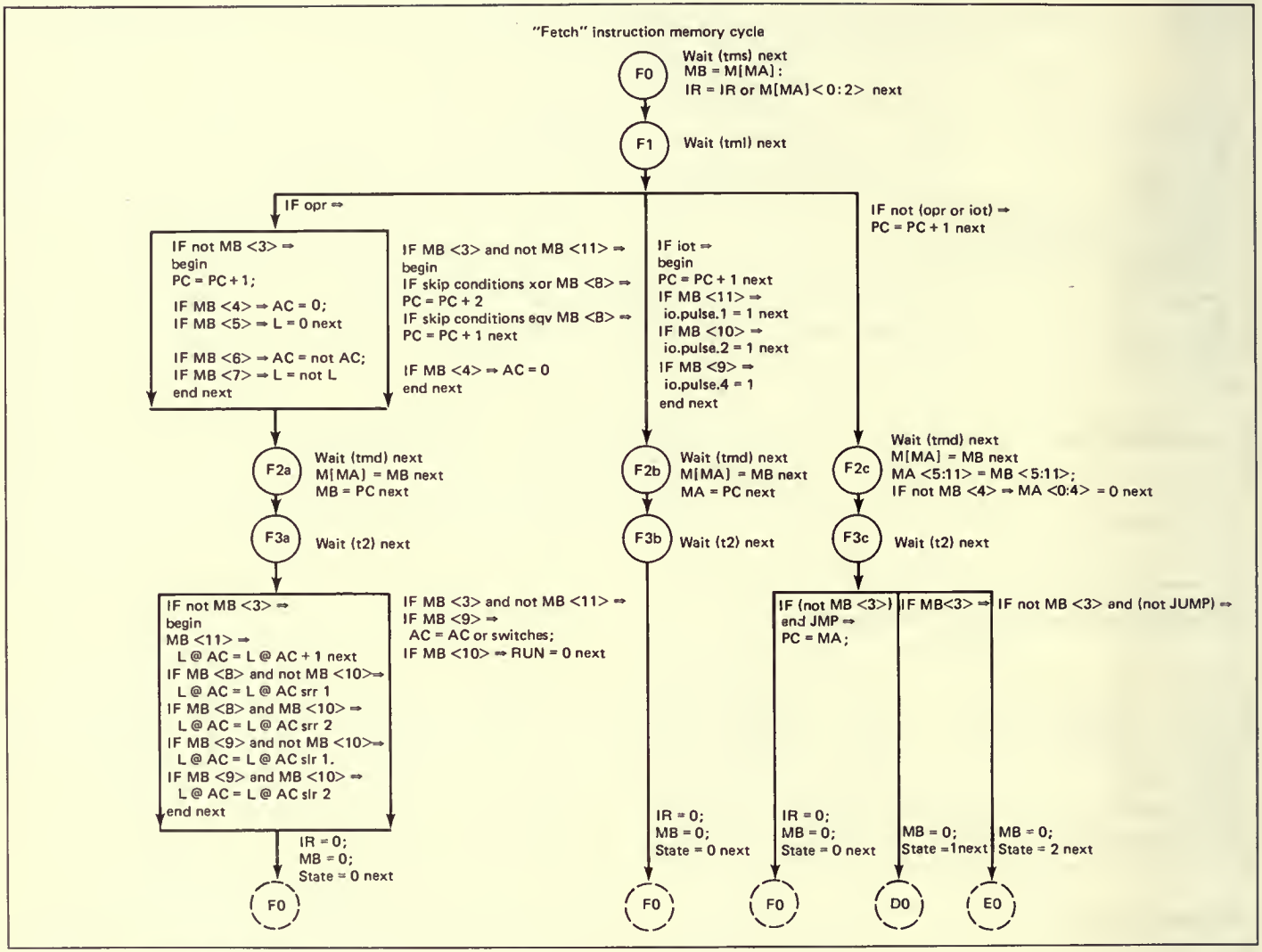


Fig. 7. PDP-8 Pc state diagram.

is used to hold the word being read from or written to memory. The Memory Buffer also holds one of the operands for binary operations (for example, AC = AC and MB). The Memory Buffer is also used as an extension of the Instruction.Register during the instruction interpretation. The additional physical registers, not part of the ISP, are:

MB\Mempry.Buffer<0:11>
Holds memory data, instruction, and operands.
MA\Memory.Address<0:11>
Holds address of word in memory being accessed.
IR\Instruction.Register<0:2>
Holds the value of current instruction being performed.
state.register<0:1>
A ternary state register holding the major state of memory cycle being performed—declared as 2 bits.
F\Fetch:=(IF state.register eqv 0)
Memory cycle to fetch instruction.
D\Deferred:=(IF state.register eqv 1)
Memory cycle to get address of operand.
E\Execute:=(IF state.register eqv 2)
Memory cycle to fetch (store) operand and execute the instruction.

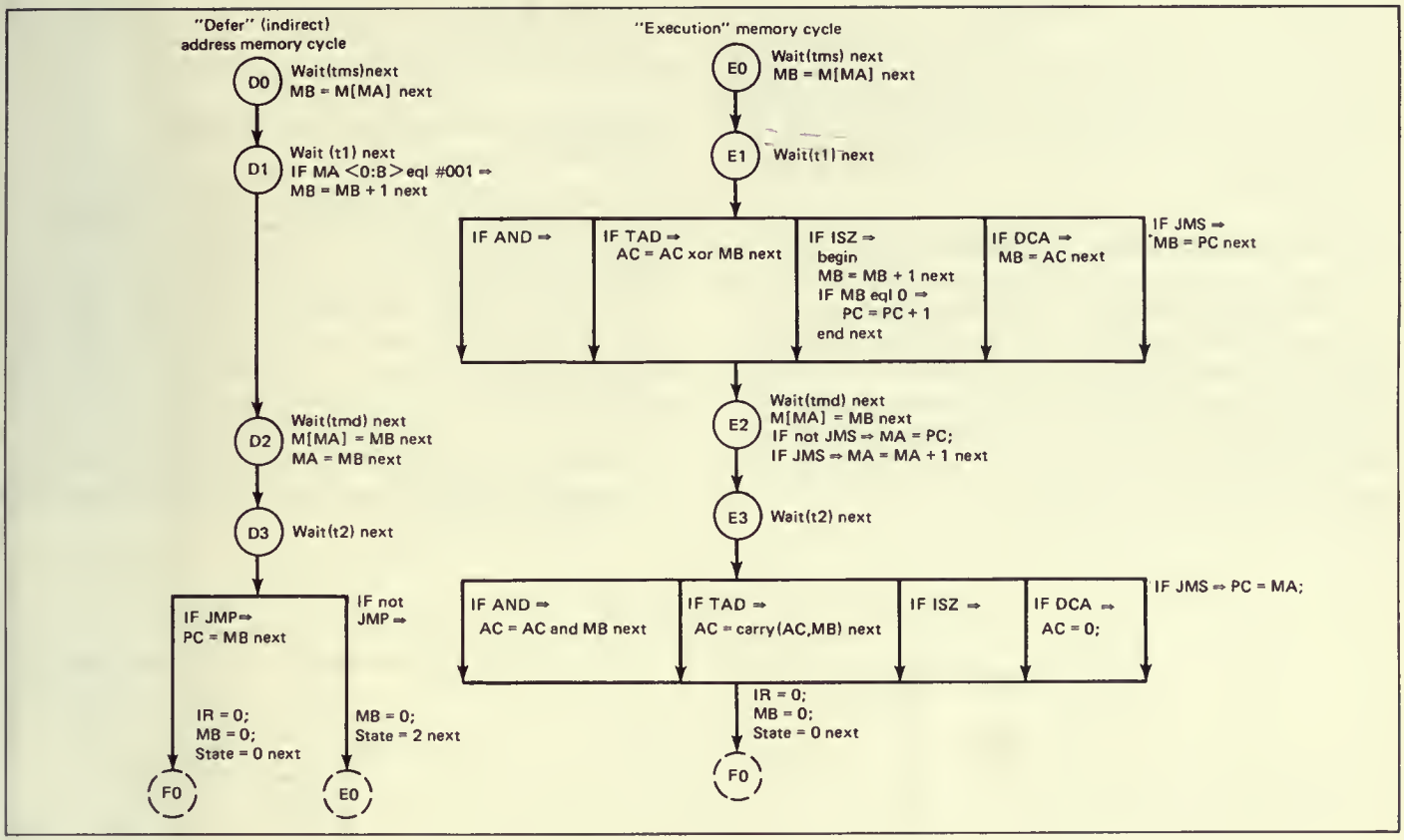


Fig. 7. (*Continued*)

The emphasis in Fig. 6 is on the static definition (or declaration) of the information paths, the operations, and state. The ISP interpretation (Appendix I) is the specification for the machine's behavior as seen by a program.

As the temporary hardware registers are added, a more detailed ISPS definition must be given in terms of time and in terms of temporary and control registers. Instead, a state of diagram (Fig. 7) is given to define the actual processor which is constrained by both the ISP registers, the temporary registers implied by the implementation, and time. The relationship among the state diagram, the ISP description, and the logic is shown in the hierarchy of Fig. 1. In the relationships shown in the figures, one can observe that the ISP definition does not have all the necessary detail for fully defining a physical processor. The physical processor is constrained by actual hardware logic and lower level details even at the circuit level. For example, a core memory is read by a destructive process and requires a temporary register (MB) to hold the value being rewritten. This is not represented within a single ISPS language statement because ISPS defines only the nondestructive transfer; however, it can be considered as the two parallel operations MB = MP[MA]; MP[MA] = 0. The explanation of the physical machine, including the rewriting of core using ISPS, is somewhat more tedious than the highest level description shown in Appendix I. For this reason, the state diagram is used (Fig. 7), and the description of the physical machine (in ISPS) is left as an exercise for the reader.

The state diagram (Fig. 7) is fundamentally driven by minor clock cycles as seen from both the state diagram and the times when the four clock signals are generated. Thus, there are 3 (state.register eqv #0,#1,#2) × 4 (clock) or 12 major states in the implementation. The Instruction.Register is used to obtain two more F2b and F3b, for the description. The state.register values 0, 1, and 2 correspond to fetching, deferred or indirect addressing (i.e., fetching an operand address), and executing. The state diagram does not describe the Extended Arithmetic Element operation, the interrupt state, or the data break states

(which add 12 more states). The initialization procedure, including the console state diagram, is also not given. One should observe that, at the beginning of the memory cycle, a new state.register value is selected. The state.register value is always held for the remainder of the cycle; i.e., only the sequences F0, F1, F2, F3, or D0, D1, D2, D3, or E0, E1, E2, E3 are permitted.

Logic Design Level (Registers and Data Operations)

Proceeding from the register transfer and ISP descriptions, the next level of detail is the logic module. Typical of the level is the 1-bit logic module for an accumulator bit, AC <j>, illustrated in Fig. 8. The horizontal data inputs in the figure are to the logic module from AC<j>, MB<j>, AC<j> input from the IO.Bus. In, and switches <j>. The control signal inputs whose names are identified using the vertical bar (e.g., | AC =0 |) command the register operations (i.e., the transfers). They are labeled by their respective ISP operations (for example, AC = AC and MB, AC = AC slr 1, for rotate once left). The sequential state machine, for the processor Pc(K), generates these control signal inputs using a combinational circuit as the one shown in Fig. 9.

Logic Design Level (Pc Control, Pc(K) Sequential State Machine Network)

The output signals from the processor sequential machine (Fig. 9) can be generated in a straightforward fashion by formulating the Boolean expressions directly from the state diagram in Fig. 7. For example, the AC = 0 control signal is expressed algebraically and with a combinational network in Fig. 9. Obviously, these Boolean output control signals are functions which include the clock, the state.register, and the states of the arithmetic registers (for example, AC = 0, L = 0, etc.). The expressions should be factored and minimized so as to reduce the hardware cost of the control for the interpreter. Although the sequential controller for the processor is mentioned here only briefly, it constitutes about half the logic within the processor.

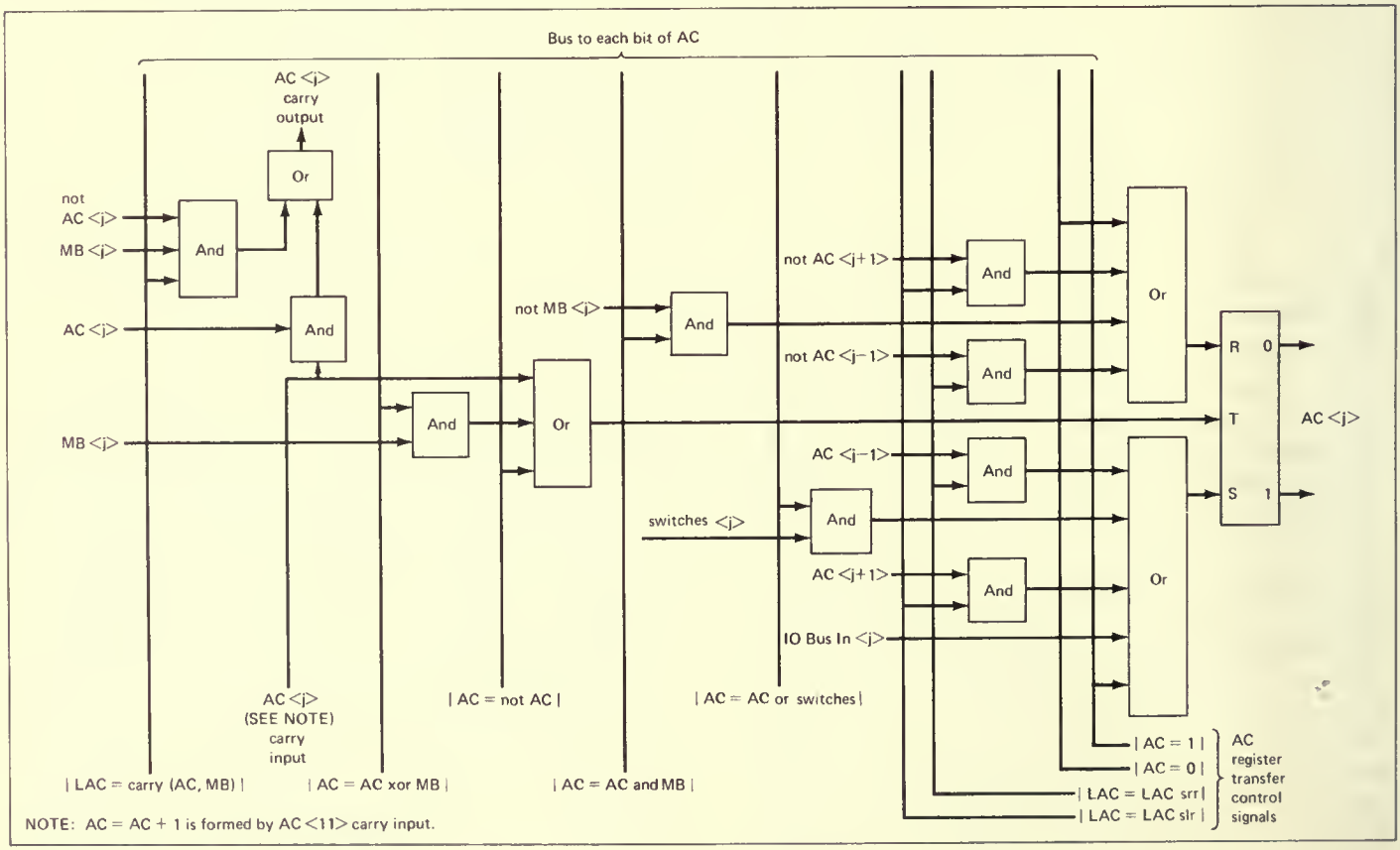


Fig. 8. PDP-8 AC<j> bit logic diagram.

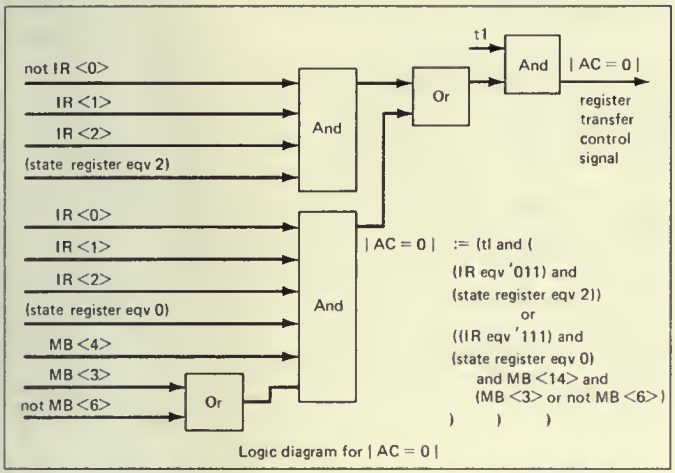
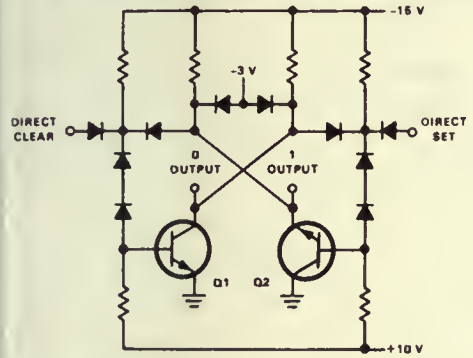


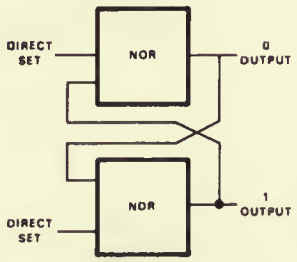
Fig. 9. PDP-8 Pc(K) AC=0 signal logic equation and diagram.

Circuit Level

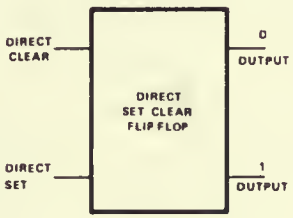
The final level of description is the circuits that form the logic functions of storage (flip-flops) and gating (NAND gates). Figures 10 and 11 illustrate some of these logic devices in detail. In Fig. 10 a direct set/direct clear flip-flop (a sequential logic element) is described in terms of circuit implementation, combinational logic equivalent, a state table, and its algebraic behavior. Note that this is not a conventional textbook circuit because it has no output delay and responds directly and immediately to an input. Some conventional sequential logic elements are used in the PDP-8, including RS(Reset-Set), T(Trigger), D(Delay), and JK. A delay in the flip-flops makes them behave in the same way as the "textbook" primitives in sequential circuit theory. The outputs require a series delay, Δt, such that, if the inputs change at time, t, the outputs will not change until $t + \Delta t$. In actuality, the PDP-8 uses capacitor-diode gates at the flip-flop inputs so that input



(a) Flip-flop circuit.



(b) Combinational logic equivalent of flip-flop.



(c) Direct set-clear flip-flop sequential logic element.

Table of Circuit Input-Output

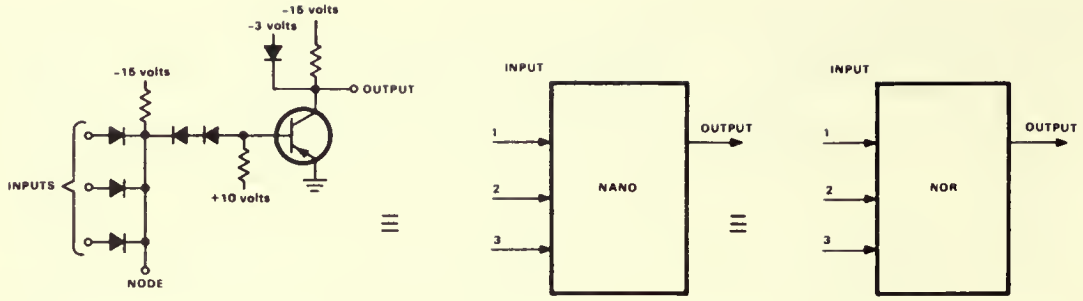
Outputs (At t) 1	Outputs (At t) 0	Inputs Direct Set	Inputs Direct Clear	Outputs (At t+) (See Note) 1	Outputs (At t+) (See Note) 0
0	-3	-3	-3	0	-3
-3	0	-3	-3	-3	0
-3	0	-3	0	-3	0
0	-3	-3	0	-3	0
-3	0	0	-3	0	-3
0	-3	0	-3	0	-3

Table of Flip-Flop Input-Output

Outputs (At t) 1	Outputs (At t) 0	Inputs Direct Set	Inputs Direct Clear	Outputs (et t+) (See Note) 1	Outputs (et t+) (See Note) 0
1	0	0	0	1	0
0	1	0	0	0	1
0	1	0	1	0	1
1	0	0	1	0	1
0	1	1	0	1	0
1	0	1	0	1	0

Note this is not an "ideal" sequential circuit element because there is no delay in the output.

Fig. 10. PDP-8 direct-coupled flip-flop and logic diagram.



(a) Multiple input inverter circuit. (b) NAND logic element. (c) NOR logic element.

Table of Circuit Behavior

Input			
1	2	3	Output
0	0	0	-3
0	0	-3	-3
0	-3	0	-3
0	-3	-3	-3
-3	0	0	-3
-3	0	-3	-3
-3	-3	0	-3
-3	-3	-3	0

Table of NAND Behavior

Input			
1	2	3	Output
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Table of NOR Behavior

Input			
1	2	3	Output
1	1	1	0
1	1	0	0
1	0	1	0
1	0	0	0
0	1	1	0
0	1	0	0
0	0	1	0
0	0	0	1

Fig. 11. PDP-8 combinational circuit and logic diagram.

changes will not be noticed until after the clock passes. This achieves the same effect.

Figure 11 illustrates the combinational logic elements used in the PDP-8. The circuit selection is limited to the inverter circuit with single or multiple inputs. These are more familiarly called NAND gates or NOR gates, depending on whether one uses positive and/or negative logic level definitions.

The core memory structure is given in Fig. 6. A more detailed block diagram showing the core stack with its twelve 64 × 64 1-bit core planes is needed. Such a diagram, though still a functional block diagram, takes on some of the aspects of a circuit diagram because a core memory is largely circuit level details. The memory (Fig. 12) consists of the component units: the two address decoders (which select 1 each of 64 outputs in the X and Y axis directions of the coincident current memory); selection switches (which transform a coincident logic address into a high current path to switch the magnetic cores); the 12 inhibit drivers (which switch a high current or no current into a plane when either a 1 or 0 is rewritten); 12 sense amplifiers (which take the induced low sense voltage from a selected core from a plane being switched or not switched and transform it into a 1 or 0); and the core stack, an array M[#0:#7777]<0:11>. Figure 12 also includes the associated circuit level hardware needed in the core memory operation (e.g., power supplies, timing, and logic signal level conversion amplifiers).

The timing signals are generated within the control portion of the processor and are shown together with processor clock in Fig. 13. The process of reading a word from memory is:

1 A 12-bit selection address is established on the MA<0:11> address lines, which is 1 of #10000 (or 4096) unique

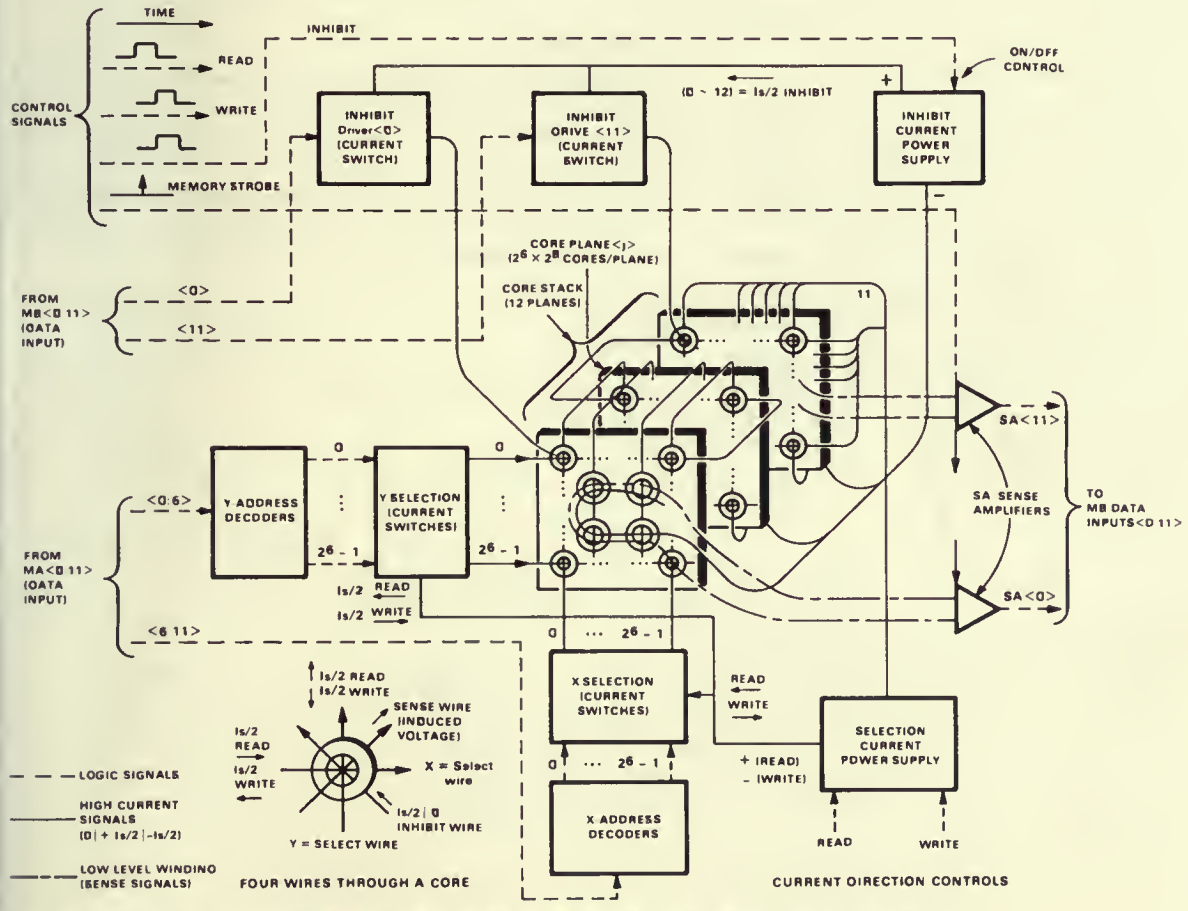


Fig. 12. PDP-8 four-wire coincident current (three dimensions) core memory logic diagram.

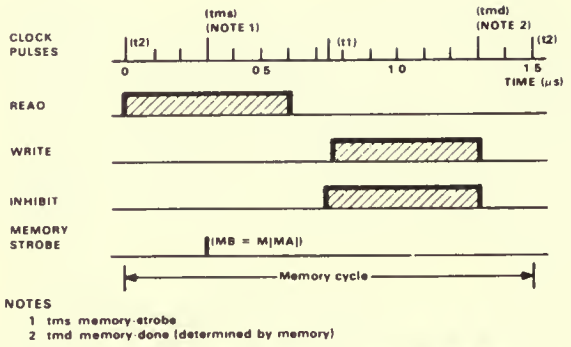


Fig. 13. PDP-8 clock and memory timing diagram.

numbers. The upper 6 bits <0:5> select 1 of 64 groups of Y addresses, and the lower 6 bits <6:11> select 1 of 64 groups of X addresses.

2 The read logic signal is made a 1 at time t2.

3 A high current path flows via the X and Y selection switches. In each of the X and Y directions, 64 × 12 cores have selection current (Ix and Iy). Only one core in each plane is selected since Ix = Iy = Iswitching/2, and the current at the selected intersection = Ix + Iy = Iswitching.

4 If a core is switched to 0 (by having Iswitching amperes through it), then a 1 is present and is read at the output of the plane bit sense amplifiers. A sense amplifier receives an input from a winding that threads every core of every bit within a core plane [#0:#7777]. All 12 cores of the selected word are reset to 0. The time at which the sense amplifier is observed is tms (the memory strobe), which also causes the transfer MB = M[MA].

5 The read current is turned off by timing in the memory module.

6 The inhibit and write (slightly delayed) logic signals are turned on at time t1. The bit inhibit signal is present or not, depending on whether a 0 or 1, respectively, is written into a bit.

7 A high current path flows via the X and Y selection switches, but in an opposite direction to the read case (see item 2). If a 1 is written, no inhibit current is present and the net current in the selected core is −Iswitching. If a 0 is written, the current is −Iswitching +(Iswitching/2) and the core remains reset.

8 The inhibit and write logic signals are turned off at time tmd specified by timing in the memory module, and the memory cycle is completed.

Device Level

For a discussion of the behavior of the transistor as it is used in these switching circuit primitives, the reader should consult semiconductor electronics and physics textbooks. It is hoped that the reader has gained a sense of how to think about the hierarchical decomposition of computers into particular levels of analysis (and synthesis) and that the hierarchical approach will be of aid in the reading of Parts 2, 3, and 4.

References

Bell, Mudge, and McNamara [1978].

APPENDIX 1 PDP-8 ISP DESCRIPTION

```
PDP8 :=
    begin

! The basic PDP-8 instruction set (without extended arithmetic element)
! is implemented.  No I/O devices are included in the description.
! I/O instruction execution is limited to the instructions that
! deal with the internal interrupt enable flags and status.

! Reference: "The DIGITAL Small Computer Handbook", 1967 Edition,
!            Digital Equipment Corporation.

**MP.State**

    MP[#0:#7777]<0:11>,                     ! Main memory (4k words)
        page.zero[#0:#177]<0:11> := MP[#0:#177]<0:11>,
        auto.index[#0:#7]<0:11>  := MP[#10:#17]<0:11>,

    MB<0:11>                                ! Memory buffer

**PC.State**

    L<>,                                    ! Link bit
    AC<0:11>,                               ! Accumulator

    PC<0:11>                                ! Program counter

**External.State**

    switches<0:11>,                         ! Console data switches
    interrupt.request<>                     ! Any device requesting interrupt

**Implementation.Declarations**

    go<>,                                   ! 1 when running
    interrupt.enable<>,                     ! 1 when Pc can be interrupted
    last.pc<0:11>,
    skip<>                                  ! Skip flag

**Instruction.Format**

    IR\instruction.register<0:2>,           ! Operation code
    ib\indirect.bit<>     := MB<3>,         ! Memory reference:
                                            ! 0 = direct; 1 = indirect
    pb\page.bit<>         := MB<4>,         ! 0 = zero page; 1 = current page
    pa\page.address<0:6> := MB<5:11>,
    io.select<0:5>        := MB<3:8>,       ! I/O device select
    io.pulse<0:2>         := MB<9:11>,      ! I/O pulse control bits
       io.pulse.1<>       := io.pulse<0>,
       io.pulse.2<>       := io.pulse<1>,
       io.pulse.4<>       := io.pulse<2>,
```

APPENDIX 1 (*Cont'd.*)

```
! Instruction format (continued)

    group<>              := MB<3>,        ! Microinstruction group
    CLA<>                := MB<4>,        ! Clear AC
    CLL<>                := MB<5>,        ! Clear Link
    CMA<>                := MB<6>,        ! Complement AC
    CML<>                := MB<7>,        ! Complement Link
    RAR<>                := MB<8>,        ! Rotate right
    RAL<>                := MB<9>,        ! Rotate left
    RTx<>                := MB<10>,       ! Rotate twice
    IAC<>                := MB<11>,       ! Increment AC
    SMA<>                := MB<5>,        ! Skip on minus AC
    SPA<>                := MB<5>,        ! Skip on positive AC
    SZA<>                := MB<6>,        ! Skip on zero AC
    SNA<>                := MB<6>,        ! Skip on AC not zero
    SNL<>                := MB<7>,        ! Skip on Link not zero
    SZL<>                := MB<7>,        ! Skip on Link zero
    is<>                 := MB<8>,        ! Invert skip sense
    OSR<>                := MB<9>,        ! Logical or AC with switches
    HLT<>                := MB<10>        ! Halt the processor

**Address.Calculation**

    MA\effective.memory.address<0:11> :=
        begin
        MA = '00000 @ pa next                       ! Zero page
        IF pb => MA<0:4> = last.pc<0:4> next        ! Current page
        IF ib =>                                    ! Indirect bit
            begin
            IF MA<0:8> eql #001 =>                  ! Auto index
                MP[MA] = MP[MA] + 1 next
            MA = MP[MA]                             ! Indirect address
            end
        end

**Instruction.Interpretation**

    start{main} :=
        begin
        go = 1 next
        run()
        end,

    run\instruction.interpretation :=
        begin
        IF go =>
            begin
            MB = MP[PC]; last.pc = PC next
            PC = PC + 1 next
            exec() next
            IF interrupt.enable and interrupt.request =>
                begin
                MP[0] = PC next
                PC = 1
                end next
            RESTART run
            end
        end
```

APPENDIX 1 (*Cont'd.*)

```
**Instruction.Execution**{us}

    exec\instruction.execution :=
        begin
        IR = MB<0:2> next
        IF (IR geq #3) and (IR leq #5) => MA() next
        IF IR leq #2 => MB = MP[MA()] next
        DECODE IR =>
            begin
            #0 := AND.:= AC = AC and MB,                    ! And
            #1 := TAD := L@AC = L@AC + MB,                  ! Two's complement add
            #2 := ISZ := begin                              ! Increment and
                                                            ! skip if zero
                         MB = MB + 1 next
                         IF MB eql 0 => PC = PC + 1
                         end,
            #3 := DCA := begin                              ! Deposit and
                         MB = AC next                       ! clear accumulator
                         AC = 0
                         end,
            #4 := JMS := begin                              ! Jump to subroutine
                         MB = PC next
                         PC = MA + 1
                         end,
            #5 := JMP := PC = MA,                           ! Jump
            #6 := IOT(),                                    ! I/O execution
            #7 := OPR()                                     ! Operate
                                                            ! microinstructions
            end next
        IF (IR geq #2) and (IR leq #4) => MP[MA] = MB
        end,

    IOT :=                                                  ! I/O Transmission
        begin
        DECODE MB<3:11> =>
            begin
            #001 := ION := begin                            ! Turn interrupt on
                           interrupt.enable = 1 next
                           RESTART run
                           end,
            #002 := IOF := interrupt.enable = 0,            ! Turn interrupt off
            otherwise :=   no.op()                          ! Not implemented
            end
        end,
```

APPENDIX 1 (*Cont'd.*)

```
skip.group :=
    begin
    skip = 0 next
    DECODE is =>
        begin
        0 := begin
             IF SNL and (L  eqv 1) => skip = 1;
             IF SZA and (AC eql 0) => skip = 1;
             IF SMA and (AC LSS 0) => skip = 1
             end,
        1 := begin
             IF SZL@SNA@SPA eqv 0  => skip = 1;
             IF SZL and (L  eqv 0) => skip = 1;
             IF SNA and (AC neq 0) => skip = 1;
             IF SPA and (AC geq 0) => skip = 1
             end
        end next
    IF skip => PC = PC + 1                    ! Skip
    end,

opr :=                                        ! Operate Instructions
    begin
    DECODE group =>
        begin
        0 := begin                            ! Group 1 microinstructions
             IF CLA => AC = 0;                ! Clear accumulator
             IF CLL => L = 0 next             ! Clear link
             IF CMA => AC = not AC;           ! Complement accumulator
             IF CML => L = not L next         ! Complement link
             IF IAC => L@AC = L@AC + 1 next ! Increment accumulator
             DECODE RTx =>
                 begin
                 0 := begin                   ! Rotate
                      IF RAL => L@AC = AC@L;
                      IF RAR => AC@L = L@AC
                      end,
                 1 := begin                   ! Rotate two places
                      IF RAL => L@AC = L@AC slr 2;
                      IF RAR => L@AC = L@AC srr 2
                      end
                 end
             end,
         1 := DECODE MB<11> =>                ! Group 2 and 3
                 begin
                 0 := begin                   ! Group 2 microinstructions
                      IF HLT => go = 0;
                      skip.group() next
                      IF CLA => AC = 0 next
                      IF OSR => AC = AC or switches
                      end,
                 1 := no.op()                 ! EAE group
                 end
        end
    end
end          ! End of description
```

Chapter 15

A PDP-8 Implemented from AMD Bit-Sliced Microprocessors

Michael Tsao

An example of a microprogrammable system based on the Am2910 sequencer and the Am2901 ALU will illustrate design with bit slices. The target machine is the PDP-8 ISP (see Appendix 1 of Chap. 8). This register-transfer (RT) level design of the micromachine is thus optimized toward the basic PDP-8. However, the general principles involved in microprogramming bit slices are illustrated by this example. A major goal of this design is the clarity of implementation, rather than the economy of design.

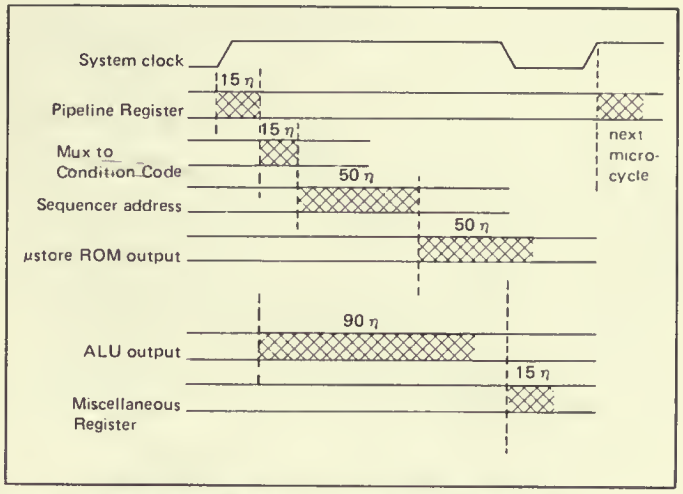


Fig. 1. One-stage pipeline microcycle timing waveform.

Overview

The basic implementation is a *one-stage pipeline* as shown in Fig. 1 in Chap. 13. In this micromachine, the pipeline register stores the current microinstruction, which is being executed by the Am2910 Sequencer and the Am2901 ALU. The status information (zero, overflow, etc.) of the ALU operations is stored in the Status Register. In a one-stage pipeline design, conditional branches can be executed only by the microinstruction following the microcycle that has generated the branching status. The Am2910 sequencer is used instead of the Am2909 to simplify the design and to aid understandability. A more cost-effective design might actually result from using the Am2909 sequencer, since the number of microinstruction types used to emulate the PDP-8 is small. The Am2901 ALU is used because it more closely reflects the ISP of the PDP-8.

A timing diagram for a typical microcycle is shown in Fig. 1. The indicated delays are typical values, illustrating the timing requirements rather than actual component performances. On the rising edge of the system clock, the Pipeline Register latches the microinstruction to be executed during this microcycle. The output of the Pipeline Register is valid 15 ns later. After another 15-ns delay, the Condition Code input to the Am2910 is valid. The microsequencer generates the next microaddress based on the current microinstruction and the Condition Code input. When the microprogram memory output is valid (approximately 130 ns after the rising clock edge), the microcycle can be restarted. Concurrently with the sequencer operation and microword fetch, the Am2901 ALU executes the operations specified by the microword in the Pipeline Register. The output of the ALU is valid prior to the falling edge of the system clock. External registers, such as the Memory Address Register (MAR) and the Status Register, use the falling clock edge to latch results from the ALU output port. In this design, the duty cycle of the system clock does not need to be symmetrical at 50 percent.

RT-Level Implementation and the Microword Format

The RT-level implementation of the Am2900/PDP-8 is shown in Fig. 2 for the control part, and in Fig. 3 for the data part. The design can best be explained in conjunction with the microword format shown in Table 1. The ISPS description of the RT-level design is listed in Appendix 2. The following subsections discuss the meaning of each microword field and the associated RT-level components. For each microword field, there are three possible bit sizes: the number of bits *normally* required for the associated components, the *minimum* required for this PDP-8 application, and the *actual* field size used. The position of each field in the microword is defined in the ISPS description. The reason for inserting extra bits is to align the fields on octal boundaries, thus aiding the reading of the encoded microprogram.

Sequencer Instruction and Address Field

The Am2910 sequencer normally requires a 4-bit-wide instruction and a 12-bit-wide "next address" direct input. The microprogram occupies less than 128 words, requiring only 7 bits of address. Two extra instruction bits and two extra address bits are inserted as 0s in this design example for octal boundary alignment.

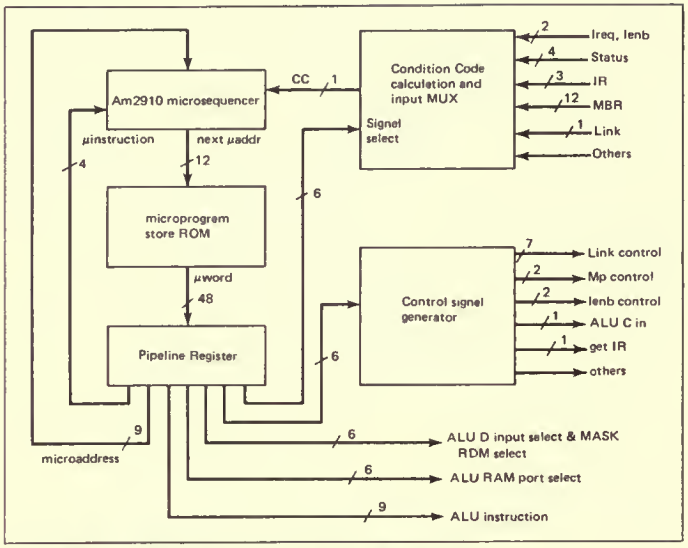


Fig. 2. Micromachine—control and sequencer.

Out of sixteen Am2910 instructions, only 4 are used in this example: Conditional Jump Subroutine (CJS, #01), Conditional Jump (CJP, #03), Conditional Return from Subroutine (CRTN, #12), and Continue (CONT, #16). Therefore, it is theoretically possible to use only 2 bits of information to specify these four actions.

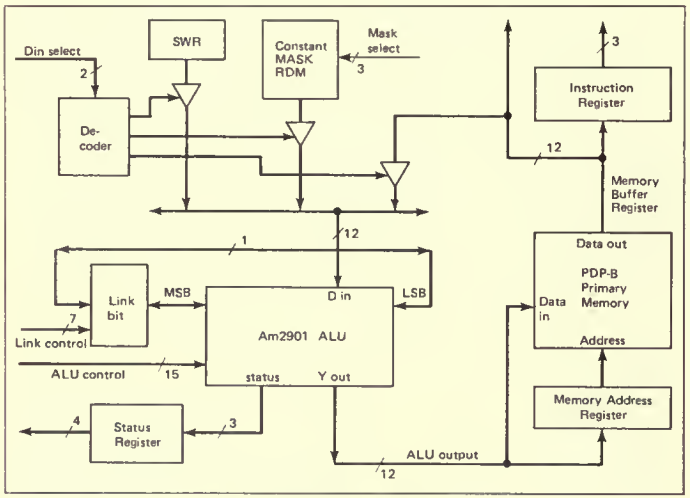


Fig. 3. Micromachine—ALU and data.

Table 1 The Microword Format and Required Bits per Field

Bits per field	*Normal*	*Minimum*	*Actual (ISP)*
Micro sequencer control			
Microinstruction	4	2	6
Next microaddress	12	7	9
Condition code select	(6)	6	6
ALU control			
ALU instruction			
Source	3	3	3
Function	3	3	3
Destination	3	3	3
RAM A port select	4	3	3
RAM B port select	4	3	3
Direct input select	(2)	2	3
Constant mask select	(3)	3	3
Miscellaneous control signals			
Control signal select	(4)	4	6
Total	48	39	48

Condition Code Input Selection

There is only one condition code (CC) input for the Am2910. The status conditions have to be multiplexed into this input. The assignments for the multiplexer input lines can be found in the ISP description in Appendix 1 (ISPS procedure Condition.Code). Five bits are used to select one out of 32 different input signals. The sixth bit in this field is used to select between the original signal and the complement of the signal. In this manner, the micromachine can branch when the signal is either high or low. When an unconditional microprogram branch is required, a logic 0 can be selected for the CC input.

Each bit from the Instruction Register (IR, 5 bits) or from the Memory Buffer Register (MBR, 12 bits) can be selected individually. This capability is used for the basic PDP-8 instruction decode, effective address calculation, and the Group 7 microinstruction decode. Random combinational logic is used to generate a single skip.enable signal for the portion of the microprogram that decodes the PDP-8 skip conditions. Interrupt requests are also handled by using combinational logic in a similar manner.

ALU Operations and the Link Bit

Three Am2901 ALU chips are cascaded to form the PDP-8 ALU section. The ALU requires a 9-bit opcode: source, function, and destination. Six bits are used to encode the A port (3 bits) and B port (3 bits) select, since only a subset of the sixteen ALU RAM registers is used in this implementation.

The PDP-8 Link bit is constructed from random logic controlled by a set of signals. For economic reasons, random logic is used rather than adding another Am2901 chip. The Link bit does not correspond to any Am2901 function, and its control would

have to be separately microprogrammed. Another alternative for the PDP-8 Link bit is to use one of the Am2901 RAM registers for storing the value. In this case, additional Link-handling microcode would have to be inserted after each PDP-8 ALU operation, increasing the target instruction execution time.

Data Input to the ALU

There is only one method of writing external data into the Am2901 ALU. It is through the Direct (D) input. In this PDP-8 design, three sources are connected to share the D input: data from the main memory (MBR), constants for ALU operations (the Mask ROM), and data in the switch register (SWITCHES). These three sources are connected by an input bus to the D input port on the ALU. The microword selects which one of the three will be the source during any given microcycle.

The use of a separate ROM to store the constants can be debated. An alternative is to store the constants in the microword. It is wasteful to dedicate a microword bit field to this purpose, since the width of this field must be the same as the ALU width and constants are used infrequently. If the microword fields are multiplexed, we violate the design goal of clarity. Hence, a constant ROM is a good compromise between the two conflicting objectives. One need only store the address of the constant in the microprogram.

Miscellaneous Control Signals

The data part of this design requires many miscellaneous control signals. For example, the Link bit uses seven different signals to control its operation. Analysis indicates that only one of these signals needs to be asserted during any given microcycle. The Miscellaneous Control Select field in the microword selects one and only one signal during each microcycle. The selection code is decoded and directed to the associated destinations. The assignments of the signals can be found in the ISPS description.

The PDP-8 Primary Memory

The primary memory (MP) for the PDP-8 target machine is assumed to be constructed from "static" semiconductor memory chips. In this type of memory, the output constantly displays the content of the location selected by the address input, unless a write operation is in progress. In this PDP-8 design, the ALU output is connected with the Memory Address Register (MAR) and with the data input port of the MP. When the write enable line of the MP is asserted, the content of the ALU output port is latched into the location selected by the MAR. The Memory Buffer Register (MBR), an ISP implementation pseudoregister, is constantly displaying the content of the location selected by the MAR. For the ISPS simulation, the memory access speed is assumed to be less than one microcycle. One can read the value of MBR (containing data from MP) two microcycles after a "write" into the MAR.

The Microprogram

The encoded microprogram that emulates the PDP-8 basic instruction set is listed in Appendix 2. This program listing is extracted from an ISPS simulator command file used to simulate this microprogrammable machine. The content of the constant ROM (Mask) is defined using the ISPS simulator "set" command, e.g., "set Mask[4]=#0177." The content of the microprogram store is also defined in this manner. As an example, the instruction fetch cycle is now described. (For readability, the encoded microword is broken into seven fields separated by dashes.)

```
set uMP[000] = #03–010–10–403–12–00–10
!RUN: MAR←LastPC←PC, IF PDP8.go = 0 goto HALT:
```

If the PDP-8.go bit is off (Condition code select 10), the microprogram jumps to Halt: (location 010). The content of PC (ALU RAM[1]) is pushed to the ALU output. The value is also latched into LastPC (ALU RAM[2]). Concurrently, the value is latched into the Memory Address Register (MAR) using the control code 10.

```
set uMP[001] = # 16–000–00–503–11–21–00 !PC←PC+1
```

The value #0001 is selected from the constant Mask ROM (21). The PC value is selected at the ALU A port, added to the constant, and then latched back into PC.

```
set uMP[002] = #03–040–41–703–05–10–15
!IR←ALU.Mb←MBR, goto Exec:
```

The content of the Memory Buffer Register (obtained by the MP[MAR] operation) is latched into the ALU.Mb (ALU RAM[5]). In this cycle, the MBR is also latched into the Instruction Register (IR) by the control signal I5. The microprogram jumps to the instruction execution section (location 040, Exec:) by forcing a pass-test condition (4I) into the Am2910 sequencer Condition Code input.

```
set uMP[004] = #03–000–03–74I–00–20–10
!ENDex: MAR←0, IF no interrupt goto RUN:
```

When the instruction execution is finished, the microprogram returns to this point. The MAR is set to zero in anticipation of interrupt servicing. The MAR will be reset to the correct PC value by microinstruction uMP[001] later on. If the interrupt request is not granted (condition code 03), the microprogram jumps back to RUN: (location 000). Otherwise, the program continues to location uMP[005] to handle the interrupt.

Implementation and Simulation Results

The micromachine and the microcode were simulated and tested by the ISPS simulator. The results are presented here.

Chip Count

Since the micromachine was not actually built, the chip count is an estimate of the required hardware parts. The goal of this exercise is to identify the inefficient area in terms of the parts count, and to suggest alternative IC chip types that may reduce the parts count. (See Table 2.)

The parts count for this microprogrammed PDP-8 implementation is 35 chips. Of these IC parts, over two-thirds (25 chips) are SSI or MSI types. If IC custom-made parts are available for the Link bit, the Skip-condition generate, and the Pipeline Register, the design can be reduced to 22 chips.

Target-Machine Instruction Execution Speed

Two methods of comparing this microprogrammed PDP-8 and a basic PDP-8 are discussed here. By counting the average number of microinstructions executed for a target instruction, one can estimate the execution speed of the emulated PDP-8. Or one can compare the execution speed of the two ISPS simulators.

For each target PDP-8 instruction, the microprogram must execute the following number of microinstructions (Table 3).

On the average, 18 microwords (4 + 3 + 6 +5 or 4 + 3 + 11) are needed to do one PDP-8 target instruction. At the manufacturer-recommended microcycle time of 150 ns, and not counting the PDP-8 Mp access time, the microprogram execution speed is 2.7 μs per target instruction (150 ns × 18). The Mp access time is usually quoted at 1.3 μs for PDP-8/E and /M [Bell, Mudge, and McNamara, 1978]. For an average instruction (i.e., indirect memory reference), three memory accesses are required: instruction fetch, pointer to data (one level of indirection), and the actual data fetch. When these are added to the 2.7-μs microprogram execution time, the projected maximum average instruction time is 6.6 μs.

Another method of comparison involves the ISPS simulator. Several PDP-8 benchmark and diagnostic programs were simulated. The CPU times used by each simulator were compared. The microcoded PDP-8 uses approximately 20 times the CPU time used by the basic PDP-8 ISP. Translated into simulation CPU time, the ISP simulator of the micromachine executes approximately 1.5 PDP-8 target instructions for every CPU second on a DEC KL-10 processor.

Table 2 Chip Count for a Microprogrammed PDP-8

Chip count	*Description*
6	Microstore. The microword width is between 39 bits and 48 bits (see Table 1). In using 8-bit-wide ROM or EPROM parts, six such chips are required. Since the microprogram is less than 128 words (7 address bits), many commercially available memory chips can be used here.
6	Pipeline Register (Pipe). Eight-bit-wide *D* flip-flops are assumed here. This register is very expansive in terms of chip count. An alternative would be having a special ROM type that can latch the data in the output buffer. Another alternative is to latch the microaddress instead of the microword. In this second design, the microword fetch and ALU-Sequencer operations are in series rather then in parallel as in the original design. This is a classical cost-performance tradeoff.
1	Am2910 microsequencer. The advantage of using the Am2910 instead of the Am2909 Sequencer is evident here. The Am2909 requires two chips instead of one Am2910 for this example.
3	Am2901 ALU bit slices. Three slices are used to provide the 12-bit-wide PDP-8 data path.
5 (estimated)	Link bit and associated hardware. The link bit in this design is constructed of a *D* flip-flop, some tristate drivers, and input multiplexers. SSI implementation of the Link bit requires 14 percent (5 out of 35) of the total chip count. An alternative is to use a custom-made MSI chip for the Link bit. A second alternative is to implement the Link bit in the ALU RAM registers. In this second design, additional microcode will have to be inserted to handle the special cases, degrading the overall performance.
3	Condition Code input multiplexer. Two 16-to-1 MUXs and two 2-to-1 MUXs.
4	PDP-8 Skip condition generate. The argument for a custom MSI chip can also be made here.
3	Constant Mask ROM and associated ALU D input selection control. The Constant Mask uses two ROM chips. The D input control uses one 2-to-4 decoder. The source registers for the ALU D input bus are assumed to have build-in tristate drivers.
4 (estimated)	Other miscellaneous parts.

Table 3 Average Number of Microinstructions Executed for a Target Instruction

Words	*Description*
4	PDP-8 instruction fetch cycle. Check PDP-8.go, fetch target instruction, increment PC, check interrupt conditions.
3	Instruction decodes. A straightforward binary decision decode tree is implemented in microcode. An alternative is to use the Instruction Decode Mapping ROM capability of the Am2910. The advantage of this alternative is not clear in view of the simple PDP-8 ISP.
6	Effective Address Calculation. Depending on the addressing mode, there are five possibilities 2 words PDP-8 Page 0 address 4 words current page 6 words indirect address, Page 0 8 words indirect address, current page 9 words auto index On the average, approximately six microinstructions are needed to calculate the PDP-8 effective address (equivalent to the Page 0 indirect address).
5	Memory Reference Instructions. For each target instruction, the microcode fetches data from primary memory, executes the operation, and deposits the result in memory. Depending on the particular target instruction, anywhere between two microinstructions (JMP) and eight microinstructions (ISZ) are needed. On the average, five microinstructions are assumed.
(11)	PDP-8 OPR group microinstructions. The decoding and execution of the PDP-8 OPR instructions are highly sequential in nature. Therefore, 11 microinstructions executed is taken as the average.

Summary

In this chapter, the design of a microprogrammed PDP-8 was presented. The central component of this micromachine was the AMD bit-sliced microprocessor. Although the design was optimized toward the basic PDP-8 configuration, many issues common to all microprogramming and RT-level hardware designs were illustrated. In simulating the micromachine, the usefulness of the ISP descriptive language as a design tool was also demonstrated.

References

Bell, Mudge, and McNamara [1978].

APPENDIX 1 ISP OF A PDP-8 EMULATOR USING THE AM2901 AND AM2910

```
AMD8 :=
    begin

! ISP of a PDP8 emulator using the AM2901 and AM2910 bit slice uPc chips

! The AM2901 description is expanded to fit the 12 bits wide PDP8 data path.
! The AM2910 sequencer uses 7 address bits to address 128 microwords.

! This AMD8 description contains the following major sections.
! (1) declarations for the PDP8 target machine, the sequencer, and the ALU.
! (2) implementation pseudo registers for the ISPS description.
! (3) the main microcycle execution loop.
! (4) the sequencer Condition Code (CC) input mutiplexer.
! (5) the generator for the PDP8 skip condition and miscellaneous control signals.

**PDP8.State**

    MP[0:4095]<11:0>,                   ! Basic PDP8 4k memory
    switches<11:0>,                     ! Switch Register
    L<>,                                ! Link Register
    interrupt.enable<>,                 ! Interrupt Enable
    PDP8.go<>,                          ! RUN bit for the PDP8 target machine

    IR<0:4>,                            ! Instruction register & pb & ib
        pb<> := IR<4>,                  ! Page bit
        ib<> := IR<3>,                  ! indirect bit
    MAR<11:0>,                          ! Memory Address Register

**Sequencer.State**

    uMP[0:127]<47:0>,                   ! Microprogram memory
    PIPE<47:0>,                         ! Pipeline Register
        ! sequencer controls
        uIR<6:0>        := PIPE<47:42>, ! AM2910 instruction, padded to 8 bits
        nxtAddr<8:0>    := PIPE<41:33>, ! next micro-address field
        CCsel<5:0>      := PIPE<32:27>, ! select condition code input

        ! ALU controls
        ALUir<8:0>   := PIPE<26:18>,    ! ALU Instruction
            src<2:0> := ALUir<8:6>,     !     Source
            fnc<2:0> := ALUir<5:3>,     !     function
            dst<2:0> := ALUir<2:0>,     !     destination
        ALUport<5:0> := PIPE<17:12>,    ! ALU RAM port select
        DircSel<2:0> := PIPE <11:9>,    ! ALU direct input Select
        MaskSel<2:0> := PIPE <8:6>,     ! constant mask select

        MisCntr<5:0> := PIPE <5:0>,     ! Miscellaneous Control signals

**ALU.State**

    mask[0:15] <11:0>,                  ! constant mask ROM
    status<3:0>,                        ! ALU result status
        SEQ.ovr<> := status<3>,         ! SEQ stack overflow
        ALU.ovr<> := status<2>,         ! ALU result overflow
        ALU.n<>   := status<1>,         ! ALU result is negative
        ALU.z<>   := status<0>,         ! ALU result is zero


**Implementation.Registers**

    AMD8.go<>,                          ! go bit for the micro machine

    interrupt.request<>,                ! Interrupt request
    MBR<11:0>,                          ! Memory Buffer Register, output of MP[MAR]
    MB<0:11>    := MBR<11:0>,           ! PDP8 bit assignments
    group<>     := MB<3>,               ! Microinstruction group
    CLA<>       := MB<4>,               ! Clear AC
    CLL<>       := MB<5>,               ! Clear Link
    CMA<>       := MB<6>,               ! Complement AC
    CML<>       := MB<7>,               ! Complement Link
    RAR<>       := MB<8>,               ! Rotate right
    RAL<>       := MB<9>,               ! Rotate left
    RTx<>       := MB<10>,              ! Rotate twice
    IAC<>       := MB<11>,              ! Increment AC
    SMA<>       := MB<5>,               ! Skip on minus AC
    SPA<>       := MB<5>,               ! Skip on positive AC
    SZA<>       := MB<6>,               ! Skip on zero AC
    SNA<>       := MB<8>,               ! Skip on AC not zero
    SNL<>       := MB<7>,               ! Skip on Link not zero
    SZL<>       := MB<7>,               ! Skip on Link zero
    is<>        := MB<8>,               ! Invert skip sense
    OSR<>       := MB<9>,               ! Logical or AC with switches
    HLT<>       := MB<10>,              ! Halt the processor

    uMP.out <47:0>,                     ! uStore output
    CCode<>,                            ! condition code input

    Din<11:0>,                          ! direct input to ALU
    ALU.Cin<>,                          ! ALU carry in

    Temp.AM2910<15:0>,
    uMP.addr<6:0> := Temp.AM2910<8:0>,  ! effective uStore address

    Temp.AM2901<17:0>,
    ALU.out<11:0> := Temp.AM2901<11:0>, ! ALU result Y output
    ALU.Cout<>    := Temp.AM2901<15>,   ! ALU carry out
    ALU.lsb<>     := Temp.Am2901<18>,   ! ALU LSB output from RAM shifter
    ALU.msb<>     := Temp.Am2901<17>,   ! ALU MSB output from RAM shifter

**AMD8.Execution**

start.AMD8 (main) :=
    begin

    ! initialize the micro machine
    AMD8.go = 1;                        ! get the uMachine going
    uMP.out = uMP[0];                   ! start at uAddr 0

    ! initialize the target machine
    ! force interrupt handling which begins at PDP8 PC=1
    PDP8.go = 1;                        ! get the target-machine going
    interrupt.enable = 1;               ! enable interrupt
    interrupt.request = 1 next          ! request interrupt

    run.AMD8 :=                         ! one uCycle
        begin

        ! First half of the cycle
        PIPE = uMP.out next             ! latch uWord

        uALU.opr :=                     ! ALU operations
            begin
            DECODE DircSel =>           ! select Direct input to ALU
                begin
                #0 := Din = 0,
                #1 := Din = MBR,
                #2 := Din = mask[MaskSel],
                #3 := Din = switches
                end;
            DECODE MisCntr =>           ! set ALU carry in bit
                begin
                #14 := ALU.Cin = 1,
                OTHERWISE := ALU.Cin = 0
                end next

            ! Do ALU computation, input instruction, Aport, Bport,
            ! Direct input, carry in, MSB, LSB, and enable output
            Temp.Am2901 = AM2901(ALUir, ('0 @ ALUport<5:3>),
                        ('0 @ ALUport<2:0>), Din, ALU.Cin, L, L, '0)
            end;                        ! end of uALU.opr

        uSeq.opr :=                     ! sequencer operation
            begin
            Condition.Code()  next

            ! Do sequncer computation, input instrcution, next address,
            ! condition code, etc.
            Temp.AM2910 = AM2910(uIR<3:0>, ('000 @ nxtAddr<8:0>),
                                    CCode, '0110) next
            uMP.out = uMP[uMP.addr]     ! uStore ROM access
            end next                    ! end of uSEQ.opr

        ! Second half of the cycle
        status<3:0> = Temp.Am2910<15> @ Temp.Am2901<14>
                    @ Temp.Am2901<13> @ Temp.Am2901<12>;
        Do.Control() next

        IF AMD8.go => RESTART run.AMD8
        end                             ! end of run.AMD8

    end,                                ! end of start.AMD8

**Condition.Code**

    Condition.Code :=                   ! calculate condition code
        begin
        Decode ('0 @ CCsel<4:0>) =>     ! look at lower 6 bits
            begin
            #00 := CCode = 0,           ! #00 => always pass test
                                        ! #40 => always fail test
            #01 := CCode = 1,           ! #01 => always fail test
                                        ! #41 => always pass test
            #02 := CCode = Skip.Cond(), ! calculate Skip condition
            #03 := CCode = interrupt.request AND interrupt.enable,
                                        ! check for interrupt handling

            #04 := CCode = status<0>,   ! ALU.z, ALU output eql 0
            #05 := CCode = status<1>,   ! ALU.n, ALU MSB
            #06 := CCode = status<2>,   ! ALU.ovr, ALU operation overflow
            #07 := CCode = status<3>,   ! SEQ.ovr, sequncer stack overflow

            #10 := CCode = PDP8.go,     ! target machine RUN bit
            #11 := CCode = IR<2>,       ! LSB of the 3 bits IR
            #12 := CCode = IR<1>,
            #13 := CCode = IR<0>,       ! MS8 of the 3 bits IR
            #14 := CCode = pb,          ! page bit
            #15 := CCode = ib,          ! indirect bit
```

```
            #20 := CCode = MB<11>,      ! IAC, MQR<0>, LSB
            #21 := CCode = MB<10>,      ! RIx,  HLT
            #22 := CCode = MB<09>,      ! RAL, OSR
            #23 := CCode = MB<08>,      ! RAR, is
            #24 := CCode = MB<07>,      ! CMI, SNL, SZL
            #25 := CCode = MB<06>,      ! CMA, SZA, SNA
            #26 := CCode = MB<05>,      ! CLL, SMA, SPA
            #27 := CCode = MB<04>,      ! CLA
            #30 := CCode = MB<03>,      ! group
            #31 := CCode = MB<02>,
            #32 := CCode = MB<01>,
            #33 := CCode = MB<00>,      ! MBR<11>, MS0
            #34 := CCode = L,           ! Link bit

            OTHERWISE := CCode = 0
            end next                    ! end of decode

        ! when CCsel<5>=0 will pass test if the selected signal is low, "0"
        ! when CCsel<5>=1 will pass test if the selected signal is high, "1"
        IF CCsel<5> => CCode = not CCode

        end,                            ! end of Condition.Code


**Skip.Conditions.for.PDP8**

    Skip.Cond<> :=                      ! calculate PDP8 skip condition
        begin
        DECODE is =>
            begin
            '0 := Skip.Cond =           ! normal skip sense
                        (SNL and L) or
                        (SZA and ALU.z) or
                        (SMA and ALU.n),
            '1 := Skip.Cond =           ! invert skip sense
                        (NOT (SZL or SNA or SPA)) or
                        (SZL and NOT L) or
                        (SNA and NOT ALU.z) or
                        (SPA and NOT ALU.n)
            end
        end,                            ! end of Skip.Cond


**Miscellaneous.Controls**

    Do.Control :=                       ! do the miscellaneous control functions
        begin
        DECODE MisCntr =>               ! 8 bits
            begin
            #00 := NO.OP(),             ! no op
            #01 := IF ALU.Cout =>       ! IF carry.out THEN complement Link
                        L = not L,

            #02 := L = ALU.lsb,         ! L = old LSB
            #03 := L = ALU.msb,         ! L = old MSB

            #04 := L = 0,               ! clear Link
            #05 := IF CLL => L = 0,     ! conditional clear

            #08 := L = not L,           ! negate Link
            #07 := IF CML => L = not L, ! conditional negate

            #10 := (MAR = ALU.out next  ! set MAR
                    MQR = MP[MAR]),     ! read PDP8 main memory
            #11 := (MP[MAR] = ALU.out next
                    MBR = MP[MAR]),     ! write into PDP8 MP

            #12 := interrupt.enable = 0,! clear Intrrupt enable
            #13 := interrupt.enable = 1,! set Interrupt enable

            #14 := ALU.Cin = 1,         ! set ALU carry in
            #15 := IR = MQ<0:4>,        ! get instruction

            #18 := PDP8.go = 0,         ! stop the target machine
            #17 := AMD8.go = 0,         ! stop the micro machine


            OTHERWISE := NO.OP()
            end
        end,                            ! end of Do.Control

! logical end of the AMD8 description
```

```
! ISPS of the AM2910 and AM2901 descriptions.
! Only the key parts of the description are listed here.
! The intension is to show the necessary modifications
! on the original ISP, making the description callable
! from the AMD8 description.


! ********************************************************************
!! AMD 2910 microsequencer description in procedureal form

!AM2910 (I<3:0>, D<11:0>, CC<>, Misc.in<3:0>) <15:0> :=
!    begin
!
!**AM2910.External.State**
!
!    CCEN<>      := Misc.in<3>,          ! enable CC input
!    C1<>        := Misc.in<2>,          ! carry in to increament uPC
!    RLD<>       := Misc.in<1>,          ! load R register with D input
!    OE<>        := Misc.in<0>,          ! enable output
!
!    FULL<>,                             ! Stack full flag
!    MAP<>  := enable<1>,                ! Map address enable flag
!    PL<>   := enable<2>,                ! Pipeline address enable flag
!    VECT<> := enable<0>,                ! Vector address enable flag
!
!
!**AM2910.Operation.Cycle**[us]
!
!    run.AM2910 [main] :=                ! Basic operation loop
!       begin
!       IF not RLD => R = D next         ! Forced (external) load of reg.
!       Y() next                         ! Put out selected address
!       AM2910 = full 8 enable<2:0> 0 Y next
!       uPC = Y + C1                     ! Increment pc
!       end,                             ! end of run.AM2910
!
!       ...                              ! description of Y()
!
!    end,                                ! end of AM2901 description
!
!
!
! ********************************************************************
!! AM2901 ALU description in procedural form
!
!AM2901 (I<8:0>, A<3:0>, B<3:0>, D<11:0>, Cn<>, R3in<>, R0in<>, OE<>) <17:0> :=
!    begin
!
!**AM2901.External.State**
!
!    out<17:0>,                          ! Output carrier
!       R3out<> := out<17>,              ! RAM3 output
!       R0out<> := out<16>,              ! RAM0 output
!       Cn4<>   := out<15>,              ! Carry out
!       OVR<>   := out<14>,              ! Overflow
!       F3<>    := out<13>,              ! Sign bit out
!       FEQL0<> := out<12>,              ! High if ALU output = 0
!       Y<11:0> := out<11:0>,            ! Data outputs
!    c.out<> := alu<12>,                 ! Carry out
!
!**AM2901.Instruction.Cycle**
!
!    run.AM2901[main] :=                 ! Initialization
!       begin
!       OVR = feq10 = c.out = 0 next     ! init flags
!       source() next
!       exec() next
!       destination() next
!       F3 = f<11>; Cn4 = c.out next     ! set final flags
!       AM2901 = out                     ! push out ALU result
!       end,
!
!       ...                              ! description of source(), etc..
!
!    end                                 ! end of AM2901 description
!
!
!    end                                 ! end of AMD8 description
```

APPENDIX 2 SIMULATOR COMMAND FILE FOR AM2900 IMPLEMENTATION OF THE PDP-8

```
!
!
! Simulator command file for the AM2900 implementation of the POP8
!
radix octal
!
! constant Mask ROM, used as input to the ALU Direct input port
s Mask[0]=#0000
s Mask[1]=#0001
s Mask[2]=#0002
s Mask[3]=#0010
s Mask[4]=#0177
s Mask[5]=#7600
s Mask[6]=#7770
s Mask[7]=#0777
!
! the micro program
!       micro word format
!          #||  Sequencer instruction
!               |          01      CJS,    conditional jump to subroutine
!               |          03      CJP,    conditional jump to program
!               |          12      CRTN,   conditional return from subroutine
!               |          18      CONT,   continue
!
!               #|||       next micro address
!
!                  #||     select conditional input signals
!                    |     (see ISPS Cond.Code for signal assignments)
!
!                    #|||  ALU instructions (see AM2901 descriptions)
!                       |          431     used as NO.OP
!
!                      #||        Aport, Bport (ALU RAM register assignments)
!                        |                 0       AC
!                        |                 1       PC
!                        |                 2       Last.PC
!                        |                 4       ALU.ma, for effective addr calc.
!                        |                 5       ALU.mb, copy of Mp output
!                        |                 others - not used
!
!                        #||      Dinput source select, constant Mask select
!
!                          #||    Miscellaneous controls
!                                 (See ISP description Do.Control for detail)
!
!
! instruction fetch and interrupt handling cycle
!          # #  # #  # # # #
s uMP[000]=#0301010403120010    !RUN:           MAR←LastPC←PC, IF POP8.go=0 goto HALT:
s uMP[001]=#1600000503112100    !               PC←PC+1
s uMP[002]=#0304041703051015    !               IR←Mb←M8R, goto Exec:
!
s uMP[004]=#0300003741002010    !ENOex:         MAR←0, IF no-Intr goto RUN:
s uMP[005]=#1600000301010011    !               MP[0]←PC
s uMP[006]=#0300041703012100    !               PC←1, goto RUN:
!
s uMP[010]=#0301041231000017    !HALT:          stop uMachine, goto to Halt:
!
!
!
!
! effective address calculation
! use ALU.mb, and Last.PC, return ALU.ma
!          # #  # #  # # # #
s uMP[020]=#0302314543542410    !subMA:         MAR←ma←#0177 & mb, IF pb=0 goto MAa:
s uMP[021]=#1600000540202500    !               Q←LastPC & #7600 ! get current page #
s uMP[022]=#1600000033440010    !               MAR←ma+Q or ma ! form 12bits addr
s uMP[023]=#1200015540402600    !MAa:   crtn,   Q←ma & #7770, IF Ib=0 RETURN
s uMP[024]=#1600000660002300    !               Q ← Q xor #0010 ! check auto index
s uMP[025]=#0302704703041000    !               ma←MP[MAR], IF no-Auto-Index goto MAb:
s uMP[026]=#1600000503442111    !               MP[MAR]←ma←ma+1 ! incr. Auto-index Reg
!
s uMP[027]=#1600000401400010    !MAb:           MAR←ma ! latch new addr
s uMP[030]=#1200041403000000    !       crtn,   RETURN ! extra cycle for Mp access
!
!
!
! do M8=MP[ma()], fetch data in Mp pointed by ALU.ma
!          # #  # #  # # # #
s uMP[032]=#0102041431000000    !MpMa:  cjs,    call SUBma:
s uMP[033]=#1200041703051000    !       crtn,   ALU.mb←MP[MAR], RETURN
!
! do MP[ma]=M8, depposit ALU.mb in POP8 Mp
s uMP[034]=#1600000401400010    !MpMaMb:        MAR←ALU.ma
s uMP[035]=#0300441401500011    !               MP[ma]←mb, goto ENOex:
!
!
!
! instruction execution
!          # #  # #  # # # #
s uMP[040]=#0306653431000000    !Exec:          IF IR<0>=1 goto IR4567:
s uMP[041]=#0305052431000000    !               IF IR<1>=1 goto IR23:
s uMP[042]=#0103241431000000    !       cjs,    call MpMa:
s uMP[043]=#0304551431000000    !               IF IR<2>=1 goto IAD:
s uMP[044]=#0300441143500000    !AND:           AC←AC & ALU.mb, goto ENOex:
s uMP[045]=#0300441103500001    !IAU:           LAC←LAC+ALU.mb, goto ENOex:
!
! ISZ and OCA
!          # #  # #  # # # #
s uMP[050]=#0305551431000000    !IR23:          IF IR<2>=1 goto OCA:
s uMP[051]=#0103241431000000    !ISZ:   cjs,    call MpMa:
s uMP[052]=#1600000503552100    !               ALU.mb ← ALU.mb+1
s uMP[053]=#R303404431000000    !               IF ALU.Z=0 goto MpMaMb:
s uMP[054]=#0303441503112100    !               PC←PC+1, goto MpMaMb:
!
s uMP[055]=#0102041431000000    !OCA:   cjs,    call subMa:
s uMP[056]=#1600000403050000    !               ALU.mb ← AC
s uMP[057]=#0303441443000000    !               AC←0, goto MpMaMb:
!
!
! the other 4 instructions
!          # #  # #  # # # #
s uMP[066]=#0307752431000000    !IR4567:        IF IR<1>=1 goto IR07:
s uMP[067]=#0102041431000000    !       cjs,    call subMa:
s uMP[070]=#0307351431000000    !               IF IR<2>=1 goto JMP:
s uMP[071]=#1600000403150000    !JMS:           ALU.mb←PC
s uMP[072]=#0303441502412100    !               PC←ALU.ma+1, goto MpMaMb:
s uMP[073]=#0300441403410000    !JMP:           PC ← ALU.ma, goto ENOex:
!
!
!
! group 6 instruction, IOTs
! turn on and turn off interrupt enable
!          # #  # #  # # # #
s uMP[077]=#0312051431000000    !IR67:          IF IR<2>=1 goto OPR:
s uMP[100]=#1600000540502700    !IOT:           Q←ALU.mb & #0777
s uMP[101]=#1600000661002100    !               ALU.out←Q xor #0001
s uMP[102]=#0310404431000000    !               skip IF ALU.z=0
s uMP[103]=#0300041431000013    !!              enable interrupt, goto RUN:
s uMP[104]=#1600000661002200    !               ALU.out←Q xor #0002
s uMP[105]=#0300404431000000    !               IF ALU.z=0 goto ENOex:
s uMP[106]=#0300441431000012    !               disable interr, goto ENOex:
!
!
!
! Group 7 operating instructions
!          # #  # #  # # # #
s uMP[120]=#0314070431000000    !OPR:           IF GRP=1 goto mGRP2:
s uMP[121]=#0312327231000005    !mGRP1:         skip IF CLA=0, cond clear Link
s uMP[122]=#1600000443000000    !               AC←0
s uMP[123]=#0312525770002007    !               skip IF CMA=0, Q←#7777,cond comp Link
s uMP[124]=#1600000053000000    !               AC← not AC
s uMP[125]=#0312720431000000    !               skip IF IAC=0
s uMP[126]=#1600000503002101    !               LAC←LAC+1, carry-out cond comp Link
!
! do rotates
!          # #  # #  # # # #
s uMP[127]=#0313363431000000    !               IF RAR=1 goto Right:
s uMP[130]=#0300422431000000    !               IF RAL=0 goto ENOex:
s uMP[131]=#0300421407000003    !Left:          LAC←LAC*2, IF rt=0 goto ENOex:
s uMP[132]=#0300441407000003    !               LAC←LAC*2, goto ENOex:
s uMP[133]=#0300421405000002    !Right:         LAC←LAC/2, IF rt=0 goto ENOex:
s uMP[134]=#0300441405000002    !               LAC←LAC/2, goto ENOex:
!
!
!
! group 2 micro instructions
!          # #  # #  # # # #
s uMP[140]=#0300460431000000    !mGRP2:         IF Mb<11>=1 goto ENOex:
s uMP[141]=#0314321401000000    !               skip IF halt=0, Y←Ac
s uMP[142]=#1600000401000018    !               POP8.go←0, Y←AC
s uMP[143]=#0314502431000000    !               skip IF Skip.cond=0
s uMP[144]=#1600000503112100    !               PC←PC+1
s uMP[145]=#0314727431000000    !               skip IF cla=0
s uMP[146]=#1600000443000000    !               ac←0
s uMP[147]=#0300422431000000    !               IF osr=0 goto ENOex:
s uMP[150]=#0300441533003000    !               AC←AC or SWR, goto ENOex:
!
! end of microcode
!
```

Chapter 46

The PDP-8 Family[1]

C. G. Bell / J. E. McNamara

Figure 1 depicts the PDP-8 family tree. The family ancestry began with the Laboratory Instrument Computer (LINC) initially built at the MIT Lincoln Laboratory in 1962, which, incidentally, we believe was the earliest personal computer. DEC began manufacturing LINC's in 1965. Eventually a PDP-8 and LINC were combined in a dual processor called the LINC-8.

In 1962, the need arose to produce a replacement for an analog monitoring system as a front end to a reactor control complex. A 12-bit real time control computer, the PDP-5, was constructed.

[1]Abstracted from C. G. Bell, J. C. Mudge, and J. E. McNamara, *Computer Engineering: A DEC View of Hardware Systems Design*, Digital Press, Maynard, Mass., 1978, pp. 175–208.

The analog nature of the initial application was addressed by building an analog-to-digital converter into the Accumulator. The concept of an I/O bus was introduced instead of the radial I/O structure of previous DEC designs. The I/O Bus permitted equipment options to be added incrementally from a zero base instead of having the pre-allocated space, wiring, and cable drivers that characterized the radial structure. This lowered the entry cost of the system and simplified the later reconfiguring of machines in the field.

Although the design was optimized around the 4-Kword memory, the PDP-5 ultimately evolved to 32-Kword configurations using a memory extension unit. Similarly, although the base machine design did not include built-in multiply and divide functions, these were added later in the form of an Extended Arithmetic Element.

While the PDP-5 had been a reasonably successful computer, it soon became evident that a new machine capable of far greater performance was required. New logic technology promised a substantial speed improvement, and new core memory technology was becoming available that would permit the memory cycle

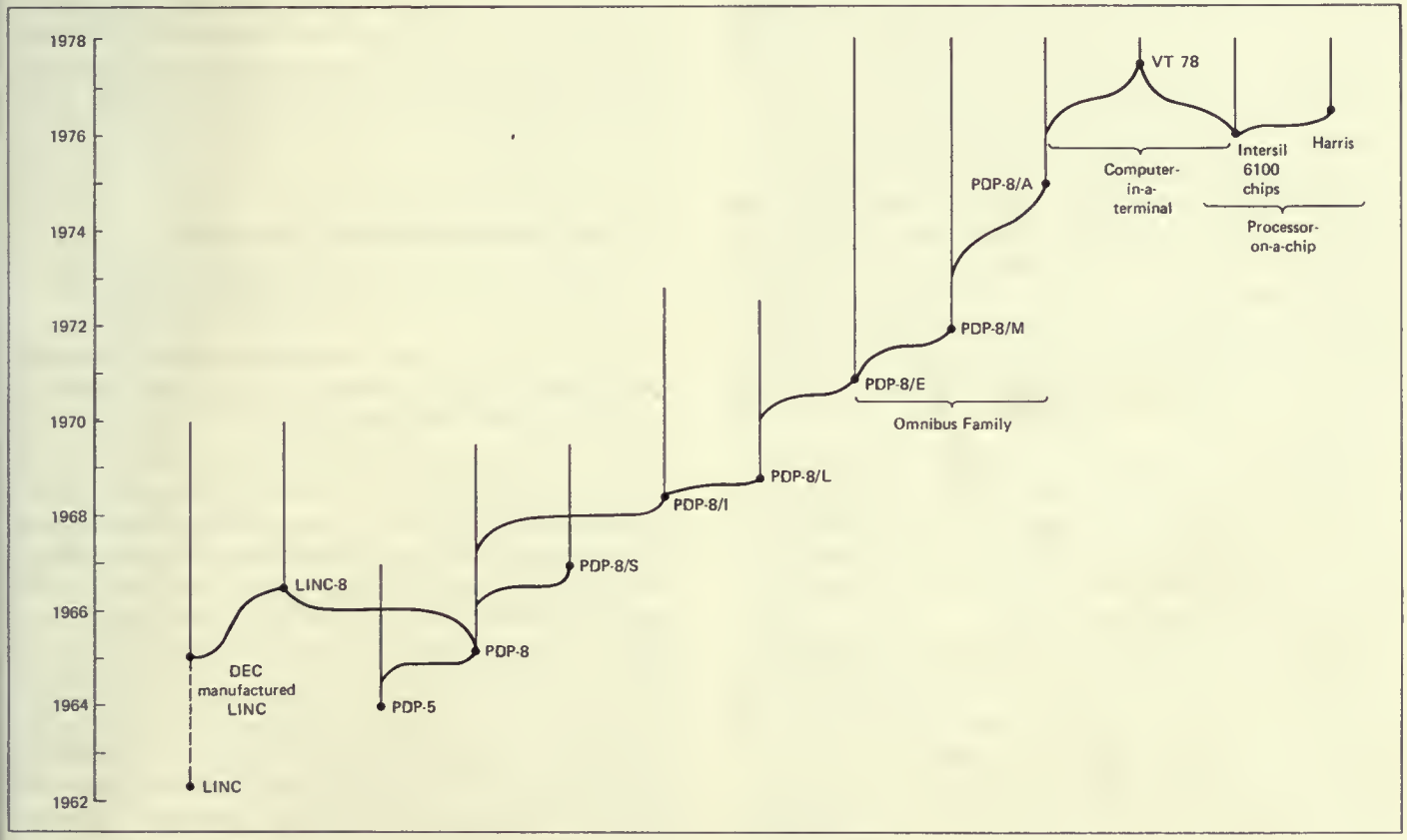


Fig. 1. PDP-8 family tree.

time to be shortened from 6 microseconds in the PDP-5 to 1.6 microseconds in the new machine. In addition, the cost of logic was now low enough so that the program counter could be moved from the memory to a separate register, substantially reducing instruction execution times. The new machine was called the PDP-8.

The new 12-bit machine was only half the size of its predecessor, occupying only half a cabinet. The net small size meant that the PDP-8 was the first true minicomputer. It could be placed on top of a lab bench or built into equipment. It was this latter property that was the most important, as it laid the groundwork for the original equipment manufacturer (OEM) purchase of computers to be integrated into total systems sold by the OEM.

Like its predecessor the PDP-5, the PDP-8 was a single-address 12-bit computer designed for task environments with minimum arithmetic computing and small primary memory requirements. Typical of these environments were process control applications and laboratory applications such as controlling pulse height analyzers and spectrum analyzers.

The PDP-8 was the first of the "8 Family." A subset, called "Omnibus 8" machines, is introduced later when the PDP-8/E, PDP-8/M, and PDP-8/A machines are discussed. Finally, computers which implement the PDP-8 instruction set in a single complementary metal oxide semiconductor (CMOS) chip will be referred to as "CMOS-8" based systems.

The PDP-8, which was first shipped in April 1965, and the other 8-Family machines that followed it achieved a production status formerly reserved for IBM computers with about 50,000 machines produced by 1979, excluding the CMOS-8 based computers. During the 15 years that these machines have been produced, logic cost per function has decreased by orders of magnitude, permitting the cost of entire systems to be reduced by a factor of 10. Thus, the 8 Family offers a rare opportunity to study the effect of technology on implementations of the same instruction set processor from early second generation to late fourth generation.

The PDP-8 was followed in late 1966 by the PDP-8/S, a cost-reduced version. The PDP-8/S was quite small in size, scarcely larger than a file cabinet drawer. It achieved its low cost by implementing the PDP-8 instruction set in serial fashion. This did reduce the cost, but it so radically reduced the performance that the machine was not a good seller.

In 1968, the PDP-8/I was produced, using medium-scale integration (MSI) integrated circuits to implement the PDP-8 instruction set with better performance than the PDP-8, and at two-thirds the price. For those customers wishing a package with less option mounting space but the same performance, the PDP-8/L was introduced later the same year.

The PDP-8/S, PDP-8/I, and PDP-8/L are mentioned only briefly here because their characteristics were basically dictated by the cost and performance improvements made possible by the emerging integrated circuit technology. The cost and performance figures for these machines are examined in greater detail in Figs. 4 to 8 and Table 1.

Shortly after the introduction of the PDP-8/L, it became evident that customers wanted a faster and more expandable machine. The continuing technological trend toward higher-density logic and some new concepts in packaging made it possible to satisfy both of these requirements but to still produce a new machine that would be cheaper than its predecessor. The new machine was the PDP-8/E. The PDP-8/E incorporated an adapter for interconnecting to PDP-8/I and PDP-8/L I/O devices. In addition, signal converters were available for interconnecting to the older PDP-5, PDP-8, and PDP-8/S I/O devices. Thus it was not necessary to design a complete new set of options at the time the machine was introduced, and existing customers could upgrade to the new computer without having to buy new peripherals. The reason for using an adapter to connect to existing I/O devices was that the PDP-8/E featured a new unified-bus I/O Bus implementation called the Omnibus.

The Omnibus, which is still in use in the PDP-8/A, has 144 pins, of which 96 are defined as Omnibus signals. The remainder are power and ground. The large number of signals permit a great number of intraprocessor communications links as well as I/O signals to be accommodated. The Omnibus signals can be grouped as follows:

1 Master timing to all components
2 Processor state information to the console
3 Processor request to memory for instructions and data
4 Processor to I/O device commands and data transfer
5 I/O device to processor, signaling completion (interrupts)
6 I/O Direct Memory Access control for both direct and Three Cycle Data Break transfers

The approximately 30 signals in groups 4 and 5 provide programmed I/O capability. There are about 50 signals in group 6 to provide the Direct Memory Access capability. These 80 signals are nearly equivalent in quantity and function to the preceding PDP-8 I/O Bus design, making the conversion from Omnibus structure to PDP-8/I and PDP-8/L I/O equipment very simple.

The processor for the PDP-8/E occupied three 8- × 10-inch boards; 4 Kwords of core memory took up three more boards; a memory shield board, a terminator board, a teleprinter control board, and the console board completed the minimum system configuration. Thus, a total of ten 8- × 10-inch boards formed a complete system. The three-board PDP-8/E processor, occupying 240 in^2, was in striking contrast to the 100-board PDP-5 processor, which occupied 2,100 in^2.

The PDP-8/E implementation was determined by the availability of integrated circuits. Multiplexers, register files, and basic arithmetic logic units performed the basic operations in a straightforward fashion using a simple sequential controller. Microprogrammed control was not feasible because suitable read-only memories were not available. Integrated circuit read-only memories available at that time were too small, holding only about 64 bits.

The PDP-8/E was mounted in a chassis which had space and power to accommodate two blocks of Omnibus slots. Thirty-eight modules could be mounted in the slots, allowing space for the processor and almost 30 peripheral option controllers. Many customers wanted to build the PDP-8/E into small cabinets and have it control only a few things. They found the large chassis and its associated price to be more than they wanted. To reach this market, the PDP-8/M was designed.

THE PDP-8/M was essentially a PDP-8/E cut in half. The cabinet had half the depth of a PDP-8/E, and the power supply was half as big. There were 18 slots available, enough for the basic processor-memory system and about eight options. The processor was the same as that for a PDP-8/E.

By 1975, DEC had been building "hex" size printed circuit boards. The hex boards were 8 × 15 inches, half again as big as the "quad" boards used in the PDP-8/E and PDP-8/M, which were 8 × 10 inches. The dimensional difference was along the contact side of the board. A hex board had six sets of 36 contacts while the quad board had only four sets. Semiconductor memory chips had also become available, so a new machine was designed to utilize the larger boards and new memories to extend the PDP-8/E, PDP-8/M to a new, lower price range. The new machine was the PDP-8/A. The PDP-8/A processor and register transfer diagram is shown in Fig. 2.

The hex modules permitted some of the peripheral controller options that had occupied several boards in the PDP-8/E to fit on a single board in the PDP-8/A. The availability of hex boards and of larger semiconductor read-only memories permitted the PDP-8/A processor to use microprogrammed control and fit onto a single board. It should be noted here that when a logic system occupies more than one board, a lot of space on each board is used by etch runs going to the connectors. This was particularly true of the PDP-8/E and PDP-8/M processor boards, due to the contacts on two edges of the boards. When an option is condensed to a single board, more space becomes available than square inch comparisons would at first indicate because many of the etch lines to the contacts are no longer required.

The first PDP-8/A semiconductor memory took only 48 chips (1 Kbit each) to implement 4 Kwords of memory. Memories of 8 Kwords and 16 Kwords were also offered. In 1977, only 96 16-Kbit chips were needed to form a 128-Kword memory. With greater use of semiconductor memory, especially read-only memory, a scheme was devised and added to the PDP-8/A to permit programs written for read-write memory to be run in read-only memory. The scheme adds a 13th bit to the read-only memory to signify that a particular location is actually a location that is both read and written. When the processor detects the assertion of the 13th bit, the processor uses the other 12 bits to address a location in some read-write memory which holds the variable information. This effectively provides an indirect memory reference.

In 1976, an option to improve the speed of floating-point computation was added to the PDP-8/A. This option is a single accumulator floating-point processor occupying two hex boards. It supports 3- or 6-word floating-point arithmetic (12-bit exponent and 24- or 60-bit fraction) and 2-word double precision 24-bit arithmetic. As a completely independent processor with its own instruction set processor, it has its own program counter and eight index registers. The performance, approximately equal to that of an IBM 360 Model 40, provides what is probably the highest performance/cost ratio of any computer.

More Omnibus 8 computers (PDP-8/E, PDP-8/M, PDP-8/A) have been constructed than any of the previous models. The high demand for this model appears to be due to the basic simplicity of the design, together with the ability of the user to easily build rather arbitrary system configurations.

In 1976, Intersil offered the first PDP-8 processor to occupy a single chip, using CMOS technology. (Here we should note that an internal to DEC processor-on-a-chip effort, the PDP-8/B, yielded a design in 1973.) DEC verified that it was a PDP-8 and began to apply it to a product in the fall of 1976. In the meantime, in additon to Intersil, Harris Semiconductor became a second source of chip supply for DEC.

The CMOS-8 processor block diagram is given in Fig. 3. The block diagram looks very much like a conventional PDP-8/E processor design using medium scale integrated circuits. It has a common data path for manipulating the Program Counter (PC), Memory Address (MA), Multiplier-Quotient (MQ), Accumulator (AC), and Temporary (Temp) registers. The Instruction Register (IR), however, does not share the common arithmetic logic unit (ALU). Register transfers, including those to the "outside world," are controlled by a programmable logic array (PLA), as indicated by the dotted lines in the figure.

While the CMOS-8 is the first DEC processor to be built on a single chip, the most interesting thing about it is the systems configurations that it makes possible. It is not only small in size (a single 40-pin chip), but it also has miniscule power requirements due to its CMOS construction. Thus, some very compact systems can be built using it.

An excellent example of the use of a CMOS-8 as part of a packaged system is the VT78 video terminal. The goals for this terminal were to drastically reduce costs by including the keyboard, cathode ray tube, and processor in a single package the

Table 1 Characteristics of PDP-8 Family Computers

	PDP-5	*PDP-8*	*PDP-8/S*	*PDP-8/I*
Project start, first ship	9/63	4/65	9/66	4/68
Goals	Lowest cost computer, interfaceability	Cost, much greater performance	Cost; tabletop	Better cost, more function than 8
Applications	Process control monitoring; laboratory	+message switch control Lab processing for instruments	Standalone calculator	Remote job entry station, TSS/8
Innovations/improvements	I/O bus; ISP	Wire-wrap; producible; low cost bit-sample communications controller	Serial implementation	Integrated circuits
Processor + 4 Kword memory (K$)	25.8	16.2	8.79	11.6
Same + terminal (K$)	27.0	18.0	9.99	12.8
Price/memory word ($)	1.83	1.83	0.73	1.46
Processor + 8 Kword + terminal + mass storage	51.1	38.8	30.4	28.9
Memory cycle time	6.0	1.6	8.0	1.5
Processor Mwords accessed/s	0.1	0.63	0.04	0.67
Processor bits accessed/s/$	93	466	55	651
Performance/price improvement (over predecessor)	....	5.0	0.12	11.8
Price improvement	....	1.6	1.84	0.76
Performance improvement	....	6.3	0.06	16.75
Product life (years)	3	5	3	3
Programmed I/O bus	49	49	43 + Bus	40
DMA I/O bus	49	49	49	50

size of an ordinary terminal. The CMOS-8 chip and high density RAM chips made this possible. To form a complete, stand-alone computer system that supports five terminals, mass storage was added. Because the mass storage was floppy disks, it was not in the terminal but in a small cabinet. Even without the mass storage, however, the VT78 forms an "intelligent terminal." An intelligent terminal is usually defined to include a computer whose program can be loaded (usually via a communications line) to take on a variety of characteristics—i.e., it can learn. An intelligent terminal can be used either as part of a network or as a stand-alone computer system. In the former case, the application is determined by the network to which the terminal is attached, but in the latter case, the terminal functions as a desk-top computer running various PDP-8 software.

PDP-8/L	*PDP-8/E*	*PDP-8/M*	*PDP-8/A*	*VT78*
11/68	3/71	6/72	1/75	6/77
Lower cost	Easy to configure; more functions better performance	Lower cost, limited system	Lower cost higher density	Cost; complete system in a terminal
	+Business data processing, testing;		Computer-in-a-desk	Word processing; desk-top computer, terminal
Less package	Omnibus		Semiconductor memory; floating-point processor	Processor-on-a-chip; low power
7.0	4.99	3.69	2.6	NA
8.5	6.49	5.19	4.1	NA
0.98	0.73	0.61	33.0	NA
24.1	15.3	....	....	11.6
1.6	1.3	1.3	1.5	3.6
0.63	0.76	0.76	0.67	0.28
1080	1828	2472	3092	....
1.65	1.69	1.35	1.25	....
1.66	1.4	1.35	1.42	....
0.94	1.23	1.0	0.87	0.42
3	7+	5+	2+	....
30	96	8E	8E	5 connectors
50	....	....	....	....

Technology, Price, and Performance of the 12-Bit Family

The PDP-8 has been re-implemented 10 times with new technology, from early second-generation to late fourth-generation, over a period of 15 years. Its implementations have included a minimal minicomputer and a minimal microcomputer. The performance characteristics of these implementations are given in Fig. 4. New technology can be utilized in the computer industry in three ways: lower cost implementations as constant performance and functionality, higher performance implementations at constant cost, implementation of new basic structures. Of these three ways, the PDP-8 Family has primarily used lower cost implementations of constant performance and functionality.

The points in Fig. 5 are arranged to show the cost trends of

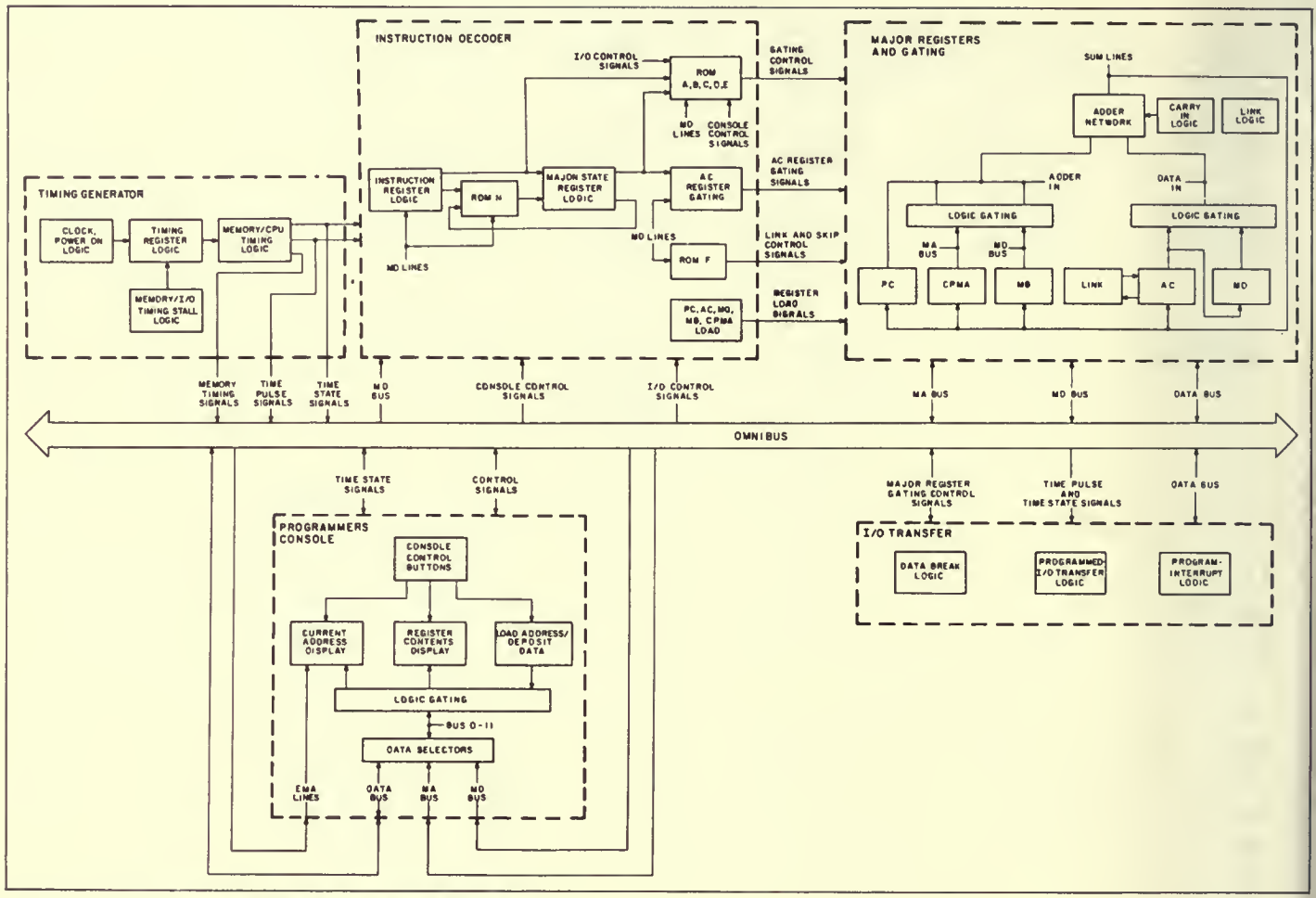


Fig. 2. PDP-8/A processor and register transfer diagram.

three configurations. The first configuration is merely a central processor with 4 Kwords of primary memory. The second configuration adds a console terminal, and the third configuration adds DECtapes or floppy disks for file storage. Note that the basic system represented in the first configuration has declined in price most rapidly: 22 percent per year in the early days and 15 percent per year in recent years. The price of primary memory, on the other hand, has declined at the rate of 19 percent per year, as seen in Fig. 6.

The price and performance trajectories for the PDP-8 family of machines are plotted in Fig. 7, with lines of constant price/performance separated at factors of 2. Note that the early implementations had significantly lower performance than the original PDP-8. Memory performance and instruction execution performance were directly related in all of these machines except the PDP-5 (which kept the Program Counter in primary memory) and the PDP-8/S (which was a serial machine). Thus, with the design emphasis on lowering the cost with each new machine, performance continued to lag behind that of the PDP-8 until higher speed primary memory was available without a cost penalty. Other performance improvements, such as the addition of floating-point hardware or the addition of a cache, are not treated in this comparative analysis.

Figure 8 gives the performance/price ratio for the PDP-8 Family machines. Setting aside the PDP-5 design point, the improvement for the 12-bit machines has been 22 percent per year.

Rather than try to fit a single exponential to the performance/

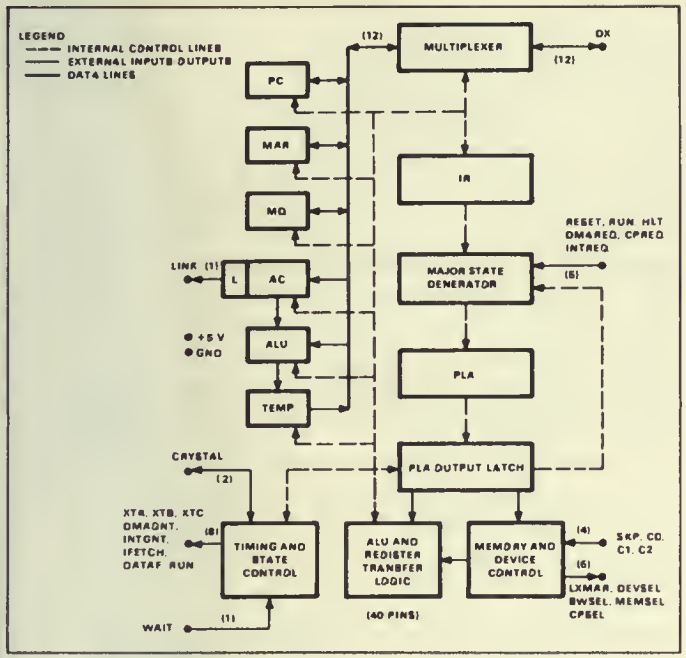


Fig. 3. Block diagram of CMOS-8.

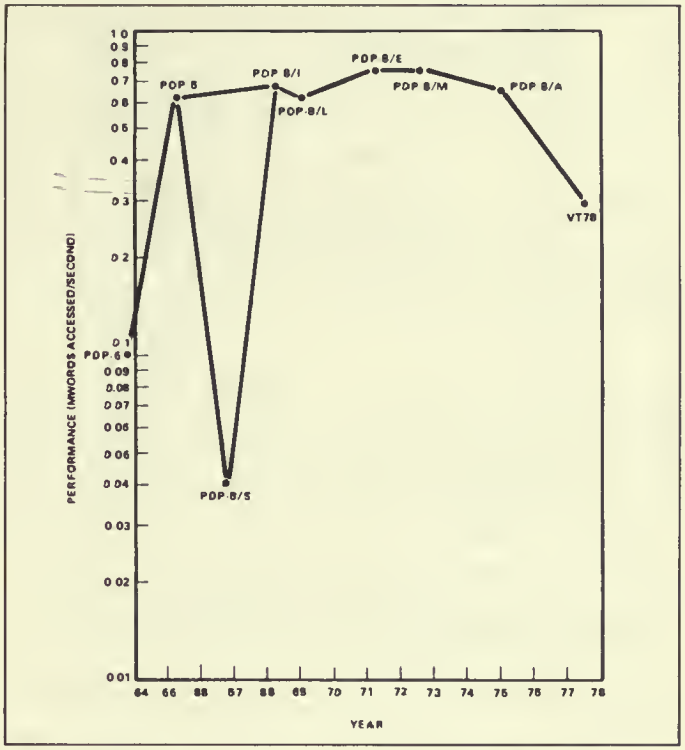


Fig. 4. Performance of DEC's 12-bit computer versus time.

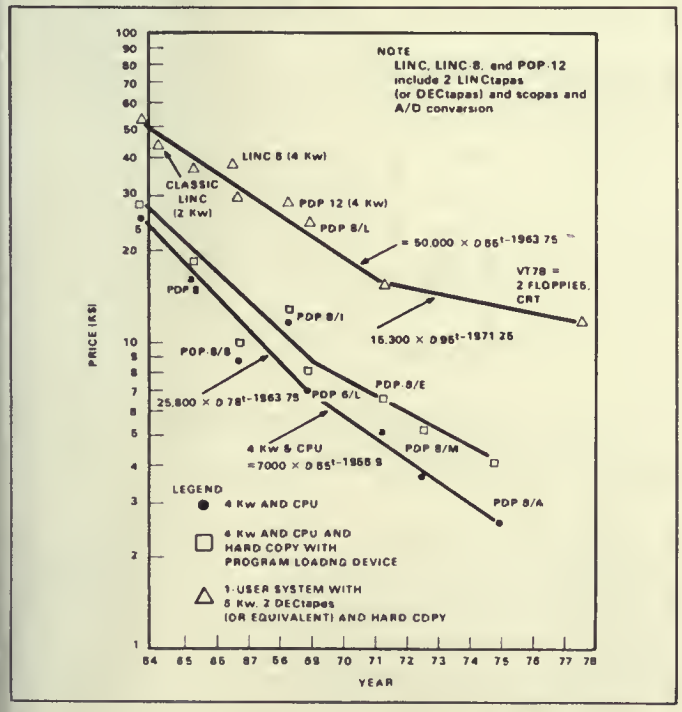


Fig. 5. Price of DEC's 12-bit computers versus time (log).

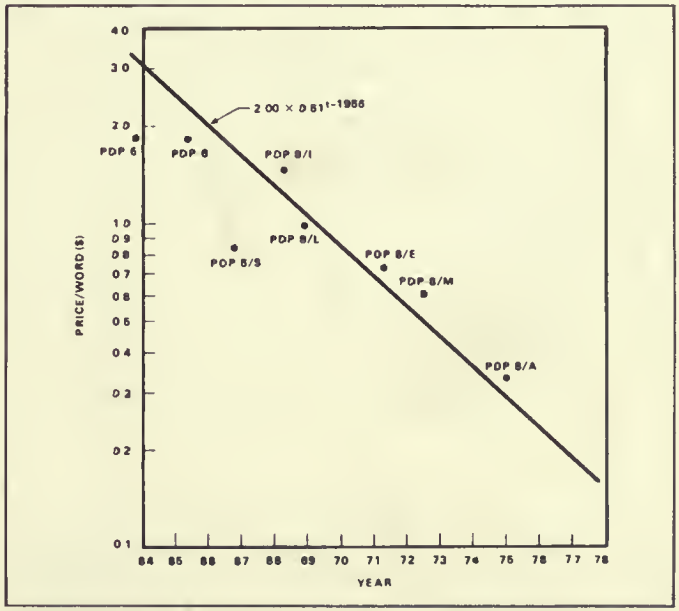


Fig. 6. Price per word of 12-bit memory versus time.

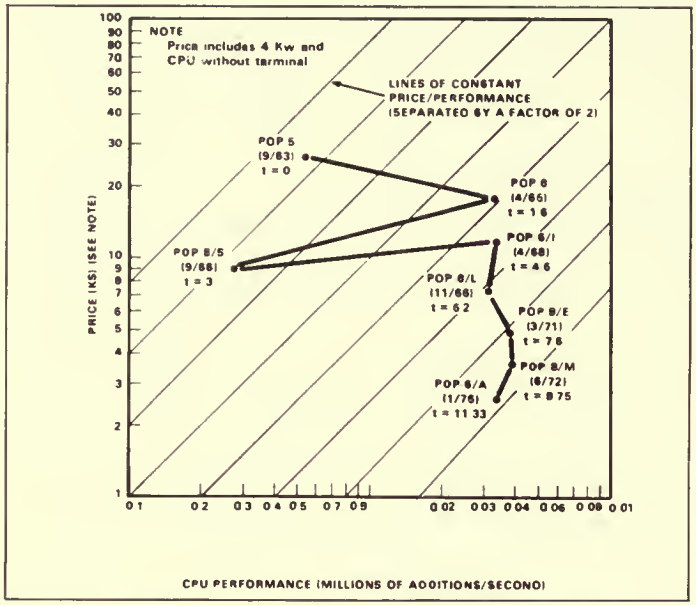


Fig. 7. Price versus performance of DEC's 12-bit computers.

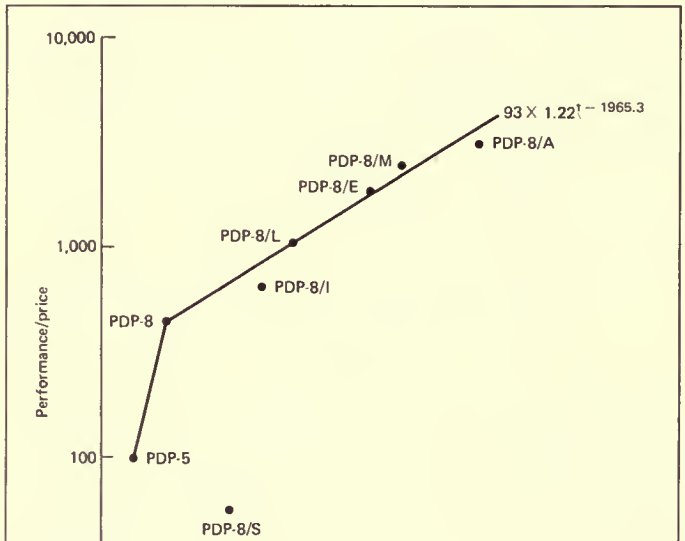


Fig. 8. Bits accessed by the central processor/s/$ versus time (for 4 K word + processor systems).

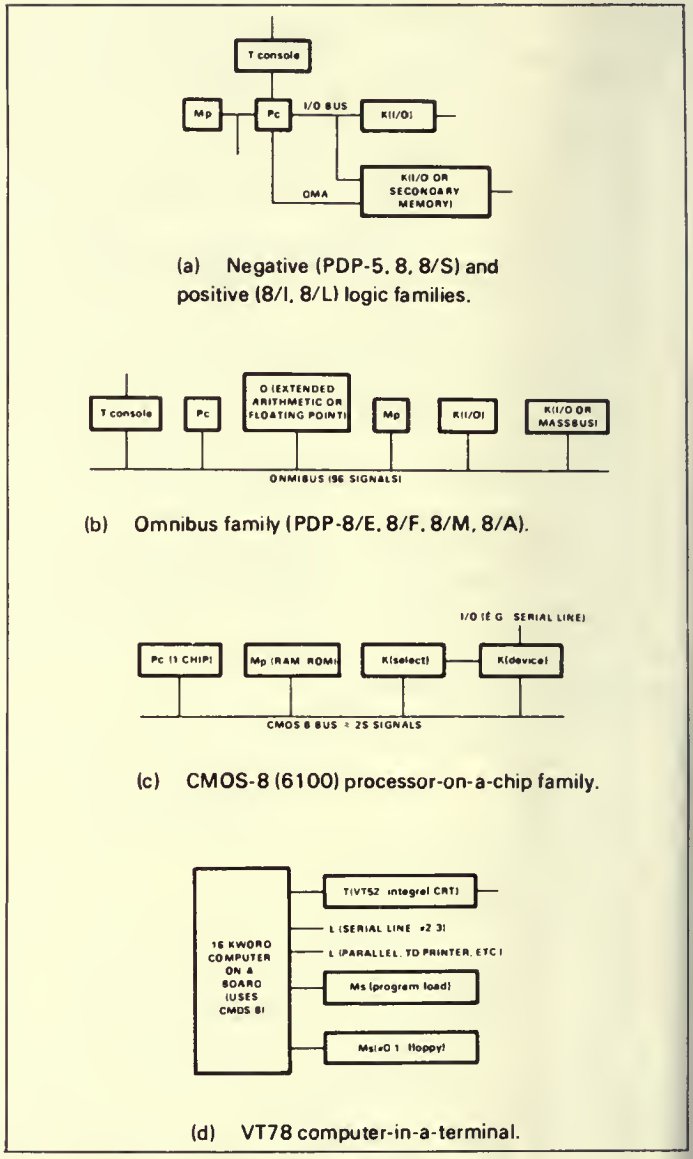


Fig. 9. Evolution of PDP-8 family PMS structures.

price data points in Fig. 8, it might be better to try two independent exponentials. The reason for this is that the data points really mark the transition between two generations. The PDP-5 was a mid-second (transistor) generation machine, and the PDP-8 represents a late second generation machine. The PDP-8/I and PDP-8/L were beginning third (integrated circuit) generation designs. These four machines represent a relatively rapid evolution from 1963 to 1968. After the PDP-8/L, the evolution slows

somewhat between 1968 and 1977, as medium-scale integrated circuits continued to be the implementation technology, and the cost of packaging and connecting components continued to be controlled by the relatively wide bus structure.

During their evolution, the DEC 12-bit computers have significantly changed in physical structure, as can be seen from the block diagrams in Fig. 9. The machines up through the PDP-8/L had a relatively centralized structure with three buses to interface to memory, program-controlled I/O devices, and Direct Memory Access devices. The Omnibus-8 machines bundled these connections together in a simpler physical structure. The CMOS-8 avoids the wide bus problem by moving the bus to lines on a printed circuit board. The number of interconnection signals on the bus is then reduced by roughly a factor of 4 to about 25 signals which can be brought into and out of the chips within the number of pins available.

Figures 4 and 7 illustrate the oscillating price/performance history of the design evolution summarized below:

1 While the PDP-5 was designed to keep price at a minimum, the PDP-8 had additions to improve the performance while not increasing price significantly over that of a slower speed design. The cost per word was modestly higher with the PDP-8 than with the PDP-5, but the PDP-8 had 6 times the performance of a PDP-5. Thus, the PDP-8 crosses three lines of constant price/performance in Fig. 7.

2 The PDP-8/S was an attempt to achieve a minimum price by using serial logic and a minimum price memory design. However, the performance of the PDP-8/S was low.

3 The market pressures created by PDP-8/S performance probably caused the return to the PDP-8 design, but in an integrated circuit implementation, the PDP-8/I.

4 The PDP-8/I was relatively expensive, so the PDP-8/L was quickly introduced to reduce cost and bring the design into line with market needs and expectations.

5 The PDP-8/E was introduced as a high performance machine that would permit the building of systems larger than those possible with the PDP-8/L.

6 The PDP-8/Mwas a lower cost, smaller cabinet version of the PDP-8/E and was intended to meet the needs of the OEM market.

The design goal of machines subsequent to the PDP-8/M has been primarily one of price reduction. The PDP-8/A was introduced to further reduce cost from the level of the PDP-8/E and PDP-8/M, although some large system configurations are still built with PDP-8/E machines. The CMOS-8 chips represent a substantial cost reduction but also a substantial performance reduction. The CMOS-8 performance is one-third that of a PDP-8/A, so a stand-alone system using a CMOS-8 is less cost-effective than an PDP-8/A when the central processor is used as the only performance criterion. The main reason for using large-scale integration is the reduced cost and smaller package rather than performance. Obviously, the next step is increased performance or more memory, or both more performance and more memory on the same chip.