7

Designing a Minicomputer

You are ready to tackle a really substantial project to round out your study of hardwired design. Nothing will sharpen your design skills more than wading through the design of a complex project from start to finish. Thus far, you have studied pieces of the design process; in the next three chapters we will help you forge your knowledge into an integrated and workable design tool. What project should we choose? Such an undertaking should be detailed yet elegant, large yet not too large. Let's design a computer!

Our aim is to design an entire operational computer system, taking no shortcuts, leaving nothing out. Most computers, even the smallest microcomputers, are highly complex structures—too complex to be a suitable teaching illustration at the MSI and LSI level of design. Instead, we choose the first minicomputer, the Digital Equipment Corporation PDP-8.

The PDP-8 has had a successful history. More than 100,000 units have been installed, many of which are still in use. The PDP-8 also has an extensive library of software and is a good machine for illustrating device interfacing.

The great advantage of the PDP-8 for our purpose is that it has a simple structure with only eight basic instructions. It exists in several models; each executes the same basic set of instructions, but they differ in minor ways. We will use the PDP-8I as the basis for our exercise. We will develop our design from first principles and make no reference to the Digital Equipment Corporation's design. The result will be functionally equivalent to the PDP-8I—for example, it will run PDP-8I software—but we will use top-down design techniques. The

only detailed information we need about the PDP-8I is a description of the action of each instruction. We shall call our design the LD20.†

The statement of the problem is brief: build a computer that will execute the PDP-8I instruction set.

PDP-8I SPECIFICATIONS

The first step is the obvious one of studying the PDP-8I to see what we must emulate. The major characteristics of the PDP-8I are:

(a) *A 12-bit word size.* This is quite small and will cause memory-addressing limitations. If a memory word is used to hold an address, it can refer to only 4096 (2^{12}) different locations. Therefore, the standard PDP-8 is limited to 4096 words of addressable memory.

(b) *A single accumulator.* Several instructions refer to an accumulator (*AC*), used to store intermediate results for later manipulation. Having only one accessible register forces a programmer to use care in saving and restoring vital data in the *AC*, for example upon subroutine entry and exit. Many computers have several registers, which can speed the execution of programs but which expose the programmer to subtle bugs if the data in all registers is not properly handled. In many applications the single *AC* is a blessing!

(c) *A 3-bit operation code.* Each instruction occupies a 12-bit word, of which 3 bits are devoted to the operation code. This provides eight basic commands—an adequate but hardly abundant number. Only 9 bits remain in the instruction for such purposes as addressing memory, whereas the 4096-word memory requires a full 12-bit address.

(d) *Paging.* Addressing limitations in minicomputers and microcomputers have forced computer architects to find a number of ingenious solutions. The PDP-8's method is based on memory pages of 128 (2^7) words. The 4096-word address space is divided into 32 pages, and each memory-referencing instruction has 7 bits to address a word within a page. The missing 5 bits of the address are not a part of the instruction, but are derived implicitly from the context. Without some trick of this sort there would be no way to pack a 3-bit command and an address into a 12-bit word. Maneuvers such as this are common features of minicomputers. The paging mechanism of the PDP-8 is perhaps the simplest technique and serves as a foundation for studying more complicated schemes used in other computers.

Throughout this design exercise, we will use the octal numbering system to specify particular values of the PDP-8's instructions, addresses, and so on.

† The LD20 design developed in this book is used in instructional laboratories for digital design. The equipment to support this design and the LD30 microprogrammed version (developed in Chapter 10) is produced by Logic Design, Inc. A laboratory manual for the LD20 and LD30 is available. See Readings and Sources at the end of this chapter.

Any such numbers not in octal will have an explicitly designated base. Thus 305 is 305 octal, 1011_2 is 1011 binary, and 42_{10} is 42 decimal.

PDP-8 Memory Addressing

In many memory addressing schemes for small instructions, the location of the current instruction is used to specify part of the operand address. For example, assume a program with five instructions stored sequentially, starting at location 300. Call these instructions CM0 (command 0) through CM4 (command 4). A memory map of this program would be:

Location	Contents
300	CM0
301	CM1
302	CM2
303	CM3
304	CM4

If instruction CM3 is being executed, we know that it is located at address 303, since that is where we placed it. Instruction CM3 can employ a subset of the 12 bits in its word to reference data located close to location 303. In the PDP-8, "close to" means in the same page.

The PDP-8 splits 4096 words of memory into 32 pages of 128 words each, as shown in Fig. 7–1. Instruction CM3 is in page 1; 7 bits are sufficient for that instruction to access any word in that page.

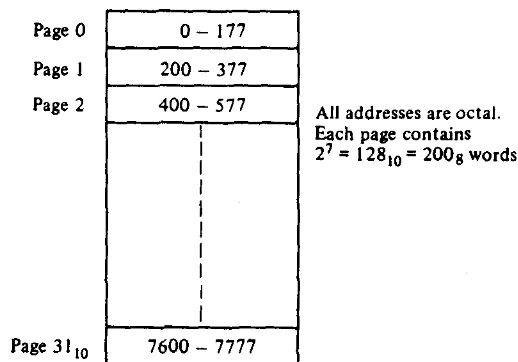
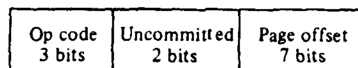


Figure 7–1 Page structure of the memory of the PDP-8.

We now have a mechanism such that an instruction needs only 7 bits to access a memory cell in one particular page. Let us call these 7 bits the *page offset*, and let the page offset occupy the rightmost 7 bits of a PDP-8 instruction:



Suppose location 301 contains the 12 bits 001 XY1 000 101_2, (for the moment we will ignore the 2 bits X and Y). The operation code is 001_2, which means an addition of the AC and the contents of a memory location. Which location? The 7 page-offset bits are 1 000 101_2 = 105_8. The instruction is to add the contents of location 105 in this page (the page containing the add instruction) to the accumulator. We know that the instruction is at location 301, and since the instruction is in page 1, the page offset is referring to page 1. Thus we will get the contents of word 105 in page 1 and add it to the AC.

What if instructions in different pages require the same data? It would be nice if some common page could be accessed by instructions in any page. In the PDP-8, page 0 has this function. We have two precious unused bits in the instruction, and we need one of them to tell if we want word 105 in the current page (page 1 in our example) or word 105 in the common page (page 0). In the PDP-8, the Y bit is used for this page selection; we call it the *page bit*.

If the page bit is 1, the page address of the current instruction is concatenated with the 7-bit offset in the instruction to form a full 12-bit address, which is sufficient to identify any word of the 4096-word memory. If the page bit is 0, the reference will be to a word in page 0 of the memory.

If we execute an instruction at location 301 that contains 001 011 000 101_2, we will add the contents of location 105 in page 1 to the AC. Location 105 in page 1 is memory location 305

$$\underbrace{000\ 01}_{\text{Page address}}\quad \underbrace{1\ 000\ 101_2}_{\text{Page offset}} = 305_8$$

If location 301 contains 001 001 000 101_2, the instruction would mean to add the contents of location 105 in page 0 to the AC. Location 105 in page 0 is memory location 105.

Indirect addressing. We have shown how the page bit and the page offset combine to yield an address either in page 0 or in the current instruction page. What happens if a command in page 2 needs to access a location in page 7? We must use all 12 bits of a word as address bits. We can do this if the word accessed by an instruction is treated not as an operand but as the *address* of an operand. This extra step is called *indirect addressing*. The PDP-8 uses the remaining instruction bit X as the *indirect bit* to specify indirect addressing. The complete format of a memory referencing instruction is

Op code 3 bits	Indirect bit	Page bit	Page offset 7 bits

In the previous examples, the contents of locations 305 or 105 (for page bits 1 or 0) were treated as 12-bit *data* words. If the indirect bit is on, these

contents are treated as 12-bit *addresses* of data. We require one extra memory cycle to access this final indirectly addressed data location.

Indirect addressing is a powerful concept since it provides a way to specify arbitrary 12-bit addresses. Into some memory word *IND* that is close to our instruction or in page 0, we load the address of the final location that we wish to access. We can then access the location by indirectly addressing it through *IND*.

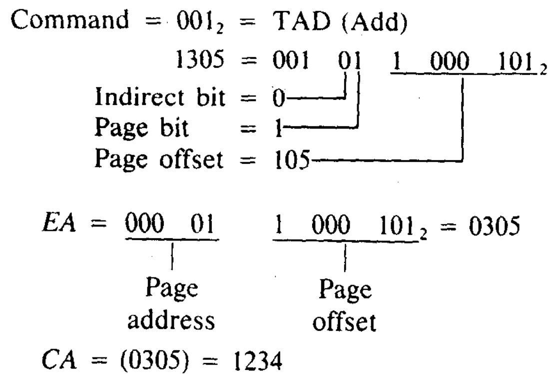
It is useful to have a shorthand for the final memory location referenced in an instruction after all applicable paging and indirect addressing are invoked. We call this final location the *effective address, EA*. The contents of location *EA* is called the *contents of the effective address, CA*. (*CA* is sometimes called the *effective operand*.) Using *EA* and *CA*, we can compactly describe the memory references of any PDP-8 instruction.

Examples of memory addressing. Here are some examples of referencing memory using PDP-8 instructions. The addresses will have 12 bits, since the PDP-8 has 4096 words of memory. We refer to the contents of an addressed memory location by enclosing the address in parentheses: If location 0301 contains 0305, then (0301) = 0305. Note that (*EA*) = *CA*.

Now assume that the following memory locations have been loaded with the data shown:

(0301) = 1305	(0305) = 1234
(0302) = 1105	(0105) = 4321
(0303) = 1705	(1234) = 5567
(0304) = 1505	(4321) = 7765

(a) What are the *EA* and the *CA* for the instruction located at 0301?



This instruction would add the quantity 1234 to the contents of the *AC*.

(b) What are the *EA* and the *CA* for the instruction located at 0302?

Command = 001_2 = TAD

1105 = 001 00 1 000 101_2

Indirect bit = 0

Page bit = 0

Page offset = 105

EA = 0105

CA = (0105) = 4321

This instruction would add the quantity 4321 to the contents of the *AC*.

(c) What are the *EA* and the *CA* for the instruction located at 0303?

Command = 001_2 = TAD

1705 = 001 11 1 000 101_2

Indirect bit = 1

Page bit = 1

Page offset = 105

EA = (0305) = 1234

CA = (1234) = 5567

This instruction would add the quantity 5567 to the contents of the *AC*.

(d) What are the *EA* and the *CA* for the instruction located at 0304?

Command = 001_2 = TAD

1505 = 001 10 1 000 101_2

Indirect bit = 1

Page bit = 0

Page offset = 105

EA = (0105) = 4321

CA = (4321) = 7765

This instruction would add the quantity 7765 to the contents of the *AC*.

Auto indexing. The PDP-8 has a feature called *auto indexing* that provides some flexibility in addressing. Most large computers have index registers to facilitate access to arrays of data. Unfortunately, specifying an index register takes 1 or more bits of the instruction and we have no bits left. The PDP-8's auto indexing is a primitive way to index without using bits in the instruction. An *auto index register* is a word in the memory that will be automatically incremented every time it is used as the source of an indirect address. The word is incremented before it is used as an address. Repeated use of the same auto index register will sequence the effective address *EA* throughout the full address space of the memory. There are 8 auto index registers in the PDP-8's main memory, locations 10_8 through 17_8. When not performing auto indexing, these locations behave like normal memory words.

Here are some examples of auto indexing. Assume that the following locations have the contents shown:

$$(0013) = 4102$$
$$(4102) = 1111$$
$$(4103) = 2000$$

(a) Instruction: $1013 = 001 \ 00 \ 0 \ 001 \ 011_2$

Command = 001_2 = TAD
Indirect bit = 0
Page bit = 0
$EA = 0013$
$CA = (0013) = 4102$

Although location 0013 is the address, there is no auto indexing because the indirect bit is 0. This instruction adds the quantity 4102 to the contents of the *AC*.

(b) Instruction: $1413 = 001 \ 10 \ 0 \ 001 \ 011_2$

Command = 001_2 = TAD
Indirect bit = 1
Page bit = 0

The initial address is 0013. This is an auto index location used as an indirect address. The auto indexing feature causes

$$(0013) \ (+) \ 1 \rightarrow (0013), \text{ or}$$
$$4102 \ (+) \ 1 \rightarrow (0013)$$

Then

$$EA = (0013) = 4103$$
$$CA = (4103) = 2000$$

The effect of executing this instruction is to increment the contents of location 0013 by 1, and to add the quantity 2000 to the contents of the *AC*.

PDP-8I Instructions

Its instruction set characterizes a computer, and therefore we must carefully study the PDP-8I's instructions. The effective address *EA* and contents of the effective address *CA* notations allow a compact description of the memory-referencing instructions.

AND (Twelve-bit logical AND). Operation code $000_2 = 0_8$.

$$AC \cdot CA \rightarrow AC$$

This is a bit-by-bit AND of the *AC* with the effective address contents. For example,

$$AC = 001\ \ 101\ \ 111\ \ 000_2$$
$$CA = \underline{110\ \ 111\ \ 101\ \ 100_2}$$
$$AC \cdot CA = 000\ \ 101\ \ 101\ \ 000_2$$

The value of $AC \cdot CA$ replaces the old contents of the AC.

TAD (Two's-complement add). Operation code $001_2 = 1_8$.

$$AC\ (+)\ CA \rightarrow AC$$

The addition is performed in the two's-complement mode; that is, the instruction implies that the numbers are 12-bit signed quantities represented in the two's-complement notation. If an arithmetic overflow occurs, the CPU toggles (complements) a special flag called the *link bit* (*LINK*).

ISZ (Increment and skip if 0). Operation code $010_2 = 2_8$.

$CA\ (+)\ 1 \rightarrow (EA)$; then, if $CA\ (+)\ 1 = 0$, skip the next instruction; otherwise execute the next instruction.

This instruction is useful in controlling loop execution.

DCA (Deposit and clear *AC*). Operation code $011_2 = 3_8$.

$$AC \rightarrow (EA);\ \text{then}\ 0 \rightarrow AC$$

The contents of the AC goes into the specified memory location, then the AC is set to 0.

JMP (Jump). Operation code $101_2 = 5_8$.

Jump to location with address EA for the next instruction.

JMS (Jump to subroutine). Operation code $100_2 = 4_8$.

Store the address of the word following the JMS instruction (i.e., the return location) in the memory word with address EA. Then jump to the location with address $EA\ (+)\ 1$ for the next instruction.

The return location is the word after the JMS instruction. This instruction stores the return address in the first word of the subroutine and then jumps to the second word, which must contain the starting instruction for the subroutine. The normal entry to a subroutine X is thus with a JMS X, which saves the return address in location X. The normal exit from the subroutine is with a JMP *X (indirect jump through location X).

OP (Operate). Operation code $111_2 = 7_8$. This is by far the most complex command in the PDP-8. It does not reference memory, so the address field bits

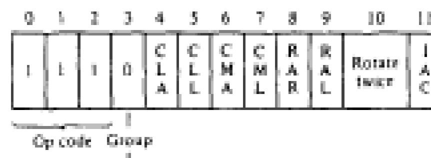
are available for other purposes. The Operate instruction permits the following basic actions:

- Clear accumulator: $0 \rightarrow AC$
- Clear link bit: $0 \rightarrow LINK$
- Complement accumulator: $\overline{AC} \rightarrow AC$
- Complement link bit: $\overline{LINK} \rightarrow LINK$
- Increment accumulator: $AC\ (+)\ 1 \rightarrow AC$
- Rotate the concatenated accumulator and link bit right or left, 1 or 2 bit positions.
- OR console switches with AC: $SR + AC \rightarrow AC$
- Skip on various conditions of the accumulator or link bit.
- Halt the computer.

Each of these operations is controlled by 1 or more bits in the address field of the instruction. These are sometimes called *microcoded instructions* or *microinstructions*. The programmer may invoke combinations of these microinstructions within one Operate instruction. There is a huge number of possible combinations; about 20 of these are useful to the programmer. These combinations of microinstructions ease the pinch of having only eight basic instructions in the PDP-8.

The operation code 111_2 occupies bits 0 through 2, as usual. Instruction bits 3 through 11 have individual functions. The Operate instruction on the PDP-8I is split into two groups, group 1 (G1) and group 2 (G2). Bit 3 specifies the group: in group 1, bit 3 = 0; in group 2, bit 3 = 1.

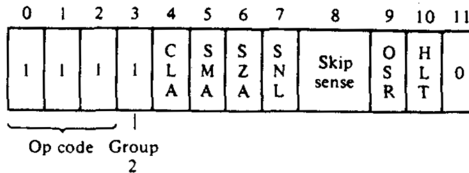
The format for group 1 is



The meaning of the microcode bits in G1 is

Bit	Mnemonic	Name
4	CLA	Clear accumulator
5	CLL	Clear link
6	CMA	Complement accumulator
7	CML	Complement link
8	RAR	Rotate accumulator and link right
9	RAL	Rotate accumulator and link left
10	—	0 = 1-bit rotation; 1 = 2-bit rotation
11	IAC	Increment accumulator

The format for group 2 is



The meaning of the microcode bits in G2 is

Bit	Mnemonic	Name
4	CLA	Clear accumulator
5	SMA	Skip on minus accumulator
6	SZA	Skip on zero accumulator
7	SNL	Skip on nonzero link
8	—	(specifies sense of skips; see discussion)
9	OSR	OR switch register into accumulator
10	HLT	Halt the computer

(In group 2 micro-operations, bit 11 is 0. On the PDP-8I, the condition of bit 11 is irrelevant, but some other models of the PDP-8 computer have another set of microinstructions, group 3, identified by bits 3 and 11, both of which are set to 1.)

To find the exact result of combining microinstructions, we must define the sequence in which the operations of each group occur. The PDP-8 describes the sequence in terms of *priorities*. There are four priority levels, 1 through 4: priority 1 operations occur before priority 2, and so on. The priority sequences of the micro-operations of G1 and G2 are

Priority	Group 1	Group 2
1	CLA CLL	Skips
2	CMA CML	CLA
3	IAC	OSR HLT
4	Rotates	

The group 2 "skip" microinstructions require further explanation. There are three conditions for skipping: SMA, SZA, and SNL. Bit 8 determines the skip mode. The operations are as follows:

If bit 8 is 0: a skip occurs if *any* of the chosen conditions is satisfied; otherwise, no skip occurs.

If bit 8 is 1: *no* skip occurs if any of the chosen conditions is satisfied; otherwise, a skip occurs.

IOT (input-output transfer). The operation code is $110_2 = 6_8$. The PDP-8 has a primitive but adequate facility for the input and output of data. We wil discuss the IOT instruction more thoroughly later; but now we will note hov data enters and leaves the computer. Outgoing data (from the PDP-8 to the external world) comes from the *AC*. Incoming data reaches the *AC* by being ORed with the existing contents of the *AC*. There is a programmable facility for clearing the *AC* prior to accepting incoming data. Thus the basic input operations are

$$0 \rightarrow AC \quad \text{(optional)}$$
$$\text{Input.Data} + AC \rightarrow AC$$

The IOT instruction also permits the programmer to enable and disable the PDP-8's interrupt system. These IOT subcommands are ION (Interrupt System On) and IOF (Interrupt System Off), and have instruction bit patterns 6001_8 and 6002_8, respectively. The presence of interrupt commands alerts us to the need to investigate the interrupt mechanism.

Interrupts. The PDP-8 specification requires that the machine be able to sense the presence of an external interrupt request. This request originates in some peripheral device and means that the device wishes to report an event of interest to the computer program. Any number of devices can request interrupt processing through this one external interrupt request line. When the PDP-8's interrupt system is activated, the computer monitors the interrupt request signal to see if any device needs servicing. If so, then at an appropriate time in the normal instruction processing cycle, the PDP-8 will force an automatic subroutine jump (JMS) to a fixed memory location (cell 0000). It is the programmer's responsibility to see that a valid subprogram for processing interrupts begins at location 0000. This subroutine is responsible for reading data from the peripheral device, writing data, or perhaps placing control information into the device. The characteristics of the device generating the interrupt determine what the interrupt subprogram must do. Therefore, the interrupt subprogram must determine which device is responsible for the interrupt and then perform actions tailored to that device. After servicing the interrupt, the subprogram will make a normal subroutine return through cell 0000 and processing of regular instructions will resume.

Interrupt requests originate from external devices running at their own pace, and may interrupt the program at any time. This is both a blessing and a curse to the programmer. Interrupt requests can occur whenever a peripheral device decides it needs service from the main computer. This is a potent programming tool, since the computer program need not waste time continually checking its peripheral devices to see if one needs service.

Interrupts are powerful; they are also tricky. The difficulty arises because interrupt requests originate from external devices and are therefore not reproducible. An interrupt may occur when the resident computer program is not prepared to handle it. For instance, suppose that the programmer has not established an interrupt service routine beginning at memory location 0000. Then the program

will not run correctly if the computer recognizes an interrupt and jumps to location 0000. Even if the interrupt service program is present, it may not properly treat all the interrupt requests that may arise. These problems are difficult to diagnose, since the debugger of the program cannot reproduce the exact sequence of instructions that led to the difficulty. Interrupt programming requires much more foresight and care than conventional programming.

To allow more control over this difficult programming task, computers with interrupts always allow the programmer to *enable* (turn on) and *disable* (turn off) the computer's interrupt detection apparatus. The programmer may select those times when interrupt requests may result in the interruption of the program. Some computers permit the handling of several types of interrupts, each type having its own interrupt jump location. We will not pursue this subject, because our focus is on the PDP-8's interrupt capabilities.

The PDP-8 programmer may enable or disable the recognition of interrupts by using the ION (Interrupt System On) and IOF (Interrupt System Off) subcommands of the IOT instruction. The PDP-8's hardware will automatically disable the interrupt system whenever an interrupt causes a jump to location 0000. This action is needed to give the programmer's interrupt service routine enough time to react to one interrupt without the danger of another interrupt occurring in the middle of the processing of the first interrupt. It is the programmer's responsibility to enable the interrupt system again at the proper time, to permit the detection of further interrupts. This gives rise to a subtle problem. The interrupt subroutine will normally leave the interrupt system disabled until it is time to return to the main (interrupted) program. At this time the interrupt subprogram must enable the interrupt system and return. The last two instructions of the subprogram are:

```
        .
        .
        .
        ION       (Turn on interrupt system)
        JMP *0    (Indirect jump to point of interruption)
```

We must make sure that we can execute the return jump to get back to the main program. Consider what would happen if an interrupt request is pending at the time the ION command is executed. The ION would reenable the interrupt system and the computer would immediately jump again to location 0000 *without* executing the jump instruction after the ION. The interrupt-forced JMS 0 causes cell 0000 to receive the address of the point of interruption—the address following the ION in this example. This act destroys the old return address in location 0000 which the unexecuted JMP *0 instruction wanted to use. The PDP-8's solution to this dilemma is to inhibit the recognition of interrupt requests for one instruction following an ION command, thus allowing the program the time to execute the crucial JMP *0 to return to the interrupted program before the computer recognizes any additional interrupt requests.

Interrupts are a complex feature of computers, and they place a heavy