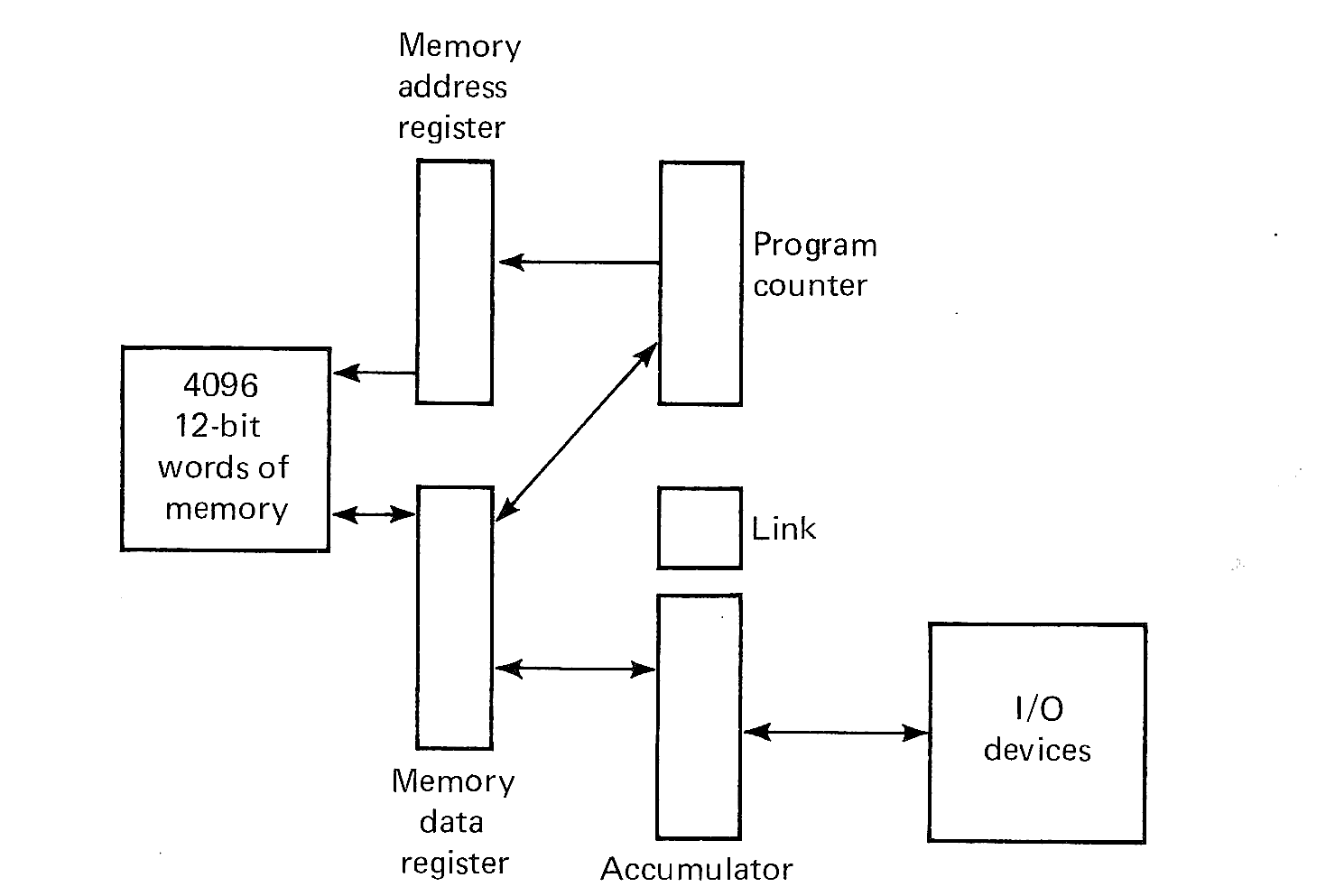
**10.2 THE PDP-8**

The PDP-8 is a small but easy to use and simple computer. It was first sold in 1965. Since then several versions have been manufactured as new hardware technology became available. The PDP-8/I, PDP-8/E, PDP-8/S, PDP-8/L and PDP-8/A are all models of this same computer. The PDP-8 is a product of the Digital Equipment Corporation. It is mainly used in dedicated data collecting or control functions, like running steel mills, medical laboratory experiments, or monitoring air pollution. The PDP-8/A was available with a CRT terminal for about $5,000.

**FIGURE 10.1** A PDP-8A computer, the most recent version of the very successful PDP-8 architecture. The two boards in the foreground are the central processor (left) and the memory (right). (Photo courtesy of Digital Equipment Corporation.)

**Memory**

The PDP-8 is a 12-bit binary machine. It uses two's complement arithmetic. With 12-bit addresses, up to 4096 words of memory can be addressed, so most PDP-8's have 4K of main memory. There are two registers in the PDP-8, the A register (a 12-bit accumulator) and the Link bit. There is also a 12-bit program counter, but this is not directly accessible to the programmer. A block diagram of the PDP-8 is shown in Figure 10.2.  
**FIGURE 10.2** A block diagram of the PDP-8. All registers except the Link bit are 12 bits.

**Instruction set**

The PDP-8 has eight instructions. These can be grouped into three classes:

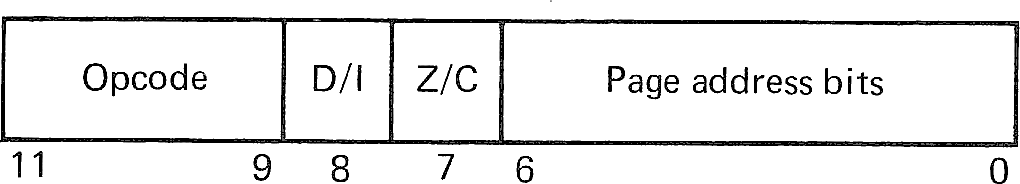
1. memory reference instructions
2. operate instruction
3. input-output instruction

For eight instructions, a 3-bit opcode is needed. In a 12-bit memory reference instruction, this leaves 9 bits to specify a memory address. But 9 bits will address only 512 words, so special addressing techniques must be used.

One technique is *indirect addressing*. One bit associated with each memory reference instruction specifies whether the address in the instruction is (0) the address of the memory location wanted (no indirection), or (1) the address of the address of the memory location (indirect addressing). Indirect addressing is at most *one* level. In order to specify the entire 4K memory, all 12 bits of a memory location are needed, so there is no bit left over in a 12-bit word to indicate if further indirection is needed.

This leaves us with eight bits in the instruction with which to specify an address. One more bit is used to specify a *page*. Memory is considered to be split into 32 *pages* of 128 words. The first page is addresses 0000 to 0177 (octal), the next page is from 0200 to 0377 (octal), 0400 to 0577 (octal), and so forth. In effect, a 12-bit address is broken into two parts: a 5-bit page number and a 7-bit location within a page.

Each memory reference instruction has one bit which is used to specify what page the address is on. This bit specifies that the address is either (0) on the *zero* page (locations 0000 to 0177) or (1) on the *current* page (same page as the current instruction). The remaining seven bits in the instruction specify the location in the page. This scheme allows certain locations (zero page) to be accessed by any instruction (allowing global variables), while the current page can be used to store local variables.

  
**FIGURE 10.3** Memory reference instruction format (PDP-8)

The memory reference instruction format is given in Figure 10.3. To interpret the instruction at location P, the Z/C bit is examined. If Z/C is zero, the high-order five bits of the memory address are zero (zero page); if Z/C is one, the high order five bits of the memory address are the same as the high order 5 bits of the address P (current page). The low-order seven bits are the address field of the instruction. This specifies a 12-bit memory address. Now if the D/I bit is zero, then this is the effective address (direct addressing); if the D/I bit is one, then the contents of the memory address are fetched, and these contents are the effective address (indirect addressing). The effective address is used in all memory reference instructions.

There are six memory reference instructions:

|  |  |  |  |
| --- | --- | --- | --- |
| *Instruction* | *Mnemonic* | *Opcode* | *Time* |
| Logical AND | AND | 0 | 2 |
| Two's complement add | TAD | 1 | 2 |
| Increment and skip if zero | ISZ | 2 | 2 |
| Deposit and clear accumulator | DCA | 3 | 2 |
| Jump to subroutine | JMS | 4 | 2 |
| Jump | JMP | 5 | 1 |

The time for each instruction is the number of memory cycles needed. The actual time varies from 1.5 to 8 microseconds per memory cycle, depending upon the model. Indirect addressing adds another memory cycle, of course.

In more detail, the instructions are

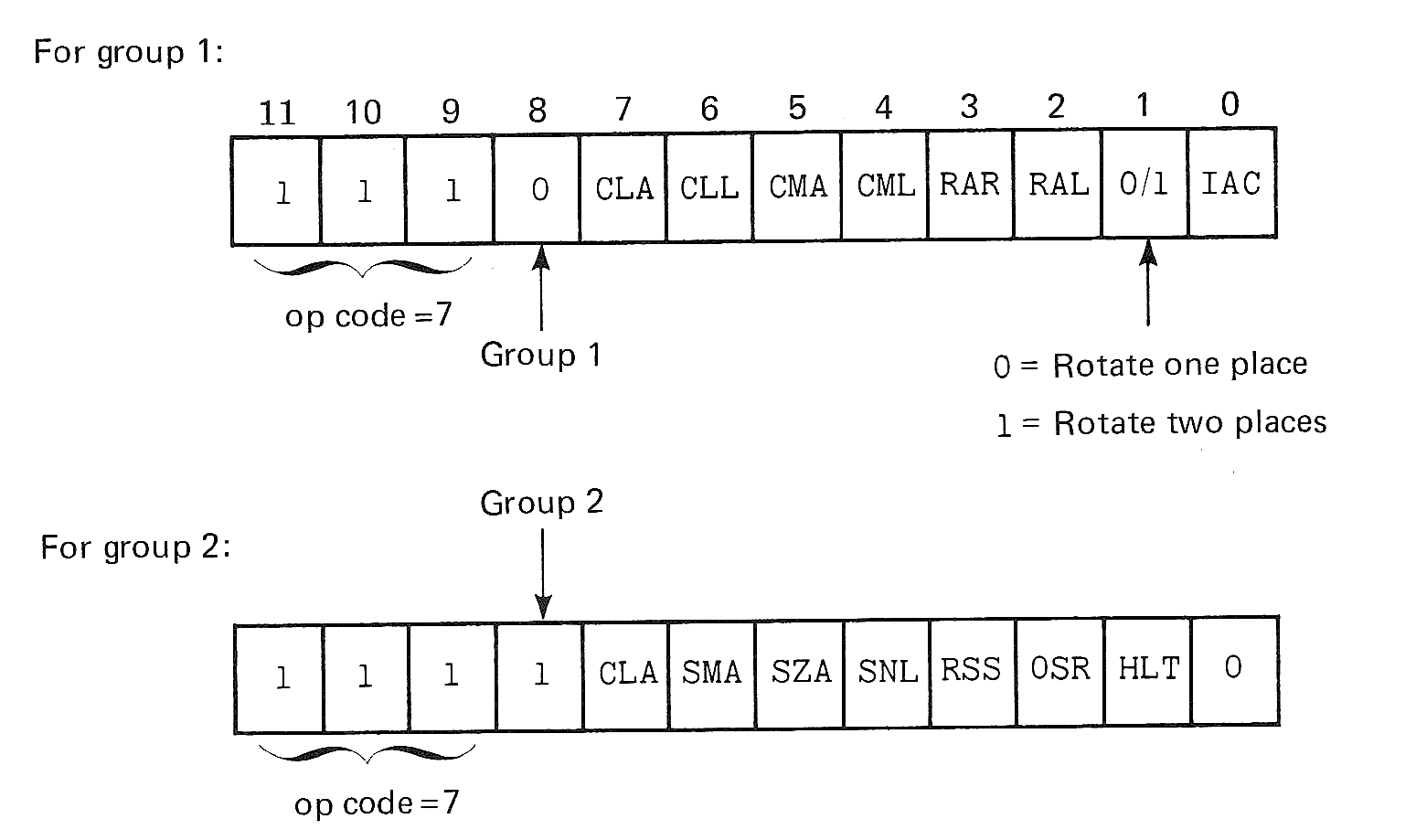
|  |  |
| --- | --- |
| AND | The contents of the effective address are ANDed with the A register. ANDing is done bitwise. The result is left in the A register; memory is not changed. |
| TAD | The contents of the effective address are added to the A register. Addition is 12-bit, two's complement integer arithmetic. The result is left in the A register; memory is not changed. A carry out of the high-order bit (sign bit) will complement the Link bit. |
| ISZ | The contents of the effective address are incremented by one and put back in the same memory location. If the result of the increment is zero, the next instruction is skipped (i.e., the program counter is incremented by 2, rather than 1). |
| DCA | Store the contents of the A register in the effective address and clear the A register (i.e., set A register to zero). The original contents of the memory location are lost. |
| JMS | The address of the next location (program counter plus one) is stored at the effective address and the program counter is set to the effective address plus one. |
| JMP | The program counter is set to the effective address. |

These instructions are a little different, but very similar to some instructions in the MIX machine. TAD is addition to the A register. DCA is a store into memory. AND is used for masking. JMP allows transfer of control. JMS stores the return address in the first word of the subroutine and starts execution at the next location; a JMP indirect through the entry point will return to the main program. The ISZ instruction is used for loops. The negative of the number of loop iterations wanted is stored in memory some place, then the ISZ instruction counts each loop. If the count is nonzero, the next instruction (a JMP to start of loop) is executed; when count is zero, we skip over the JMP and continue.

For example, to multiply the A register by 10, (where X has -10, and Y is a temporary)

|  |  |  |  |
| --- | --- | --- | --- |
|  | DCA | Y | /STORE A IN Y TO CLEAR IT |
|  | TAD | Y | /ADD OLD VALUE FROM Y TEN TIMES |
|  | ISZ | X | /X STARTS WITH NEGATIVE TEN |
|  | JMP | \*-2 | /REPEAT JUMP BACK TEN TIMES |
|  | ... | ... | /A REGISTER NOW HAS TEN TIMES OLD A |

There are still a large number of things we want to do as programmers. The *Operate* instruction is a special instruction which allows for many different functions. These functions are encoded in a very few bits. The operate instruction specifies operations which affect only the A register, Link bit, and program counter. Thus, the space used in memory reference instructions for specifying a memory address can be used for other purposes. The operate instructions require only one cycle, since they do not reference memory. There are two formats for the operate instruction; these are called group 1 and group 2 operate instructions. Bit 8 distinguishes between these two groups. The instruction format is shown in Figure 10.4.

  
**FIGURE 10.4** Format of the operate instruction of the PDP-8.

The effect of the operate instruction is determined by which of the subinstructions are selected. Each subinstruction is selected by setting the corresponding bit to one. The subinstructions are:

|  |  |
| --- | --- |
| CLA | Clear the A register; set it to zero. |
| CLL | Clear the Link bit. |
| CMA | Complement the A register (bit by bit, change 1 to 0 and 0 to 1). |
| CML | Complement the Link bit. |
| RAR | Rotate the A register right (one bit if bit 1 of the instruction is zero; two bits if bit 1 of the instruction is one). A rotate is a circular shift of the A register and Link bit. The Link bit is shifted into bit 12 of the A register, and bit 0 of the A register is shifted into the Link bit. |
| RAL | Rotate the A register left. Rotate one bit if bit 1 of the instruction is zero; two bits if bit 1 of the instruction is one. |
| RTR | Special mnemonic for rotating two bits right (sets bit 1 in the instruction). |
| RTL | Special mnemonic for rotating two bits left. |
| IAC | Add 1 to the A register. |
| SMA | Skip on Minus A. If the A register is negative, skip the next instruction. |
| SZA | Skip on Zero A. If the A register is zero, skip the next instruction. |
| SNL | Skip on Nonzero Link. If the Link bit is one, skip the next instruction. |
| RSS | Reverse Skip Sense. If this bit is one, the SMA, SZA, and SNL subinstructions will skip on the opposite condition. That is, SMA skips on positive or zero, SZA skips on nonzero, and SNL skips if the Link is zero. |
| OSR | OR from the Switch Register. The contents of the switch register on the console are ORed into the A register. |
| HLT | Halt. |

These subinstructions can be combined independently of each other to form more complicated instructions. Thus,

|  |  |
| --- | --- |
| CLA | Clear the A register. |
| CLA CLL | Clear both the A register and the Link. |
| CLA CMA | Clear the A register, then complement (set the A register to all ones). |
| CMA IAC | Complement and add 1 (two's complement). |
| CLL RAL | Clear Link; rotate one place left (multiply the A register by two; put sign bit in Link). |
| SMA SZA | Skip if the A register is less than or equal to zero. |
| CLA SZA | First, test if A is zero or not. Then clear A. If A was zero, skip next instruction. |

This last example points out that the order in which the subinstructions are executed is very important. The PDP-8 interprets these instructions for group 1 as follows:

1. CLA and CLL (if selected of course)
2. CMA and CML
3. IAC
4. RAR, RAL, RTR, and RTL

For group 2,

1. Test SMA, SZA, SNL. If any of these are selected and the condition is true, set the Skip flag. If all selected conditions are false, clear the Skip flag. (If none are selected, the Skip flag is cleared.)
2. If RSS is selected, complement the Skip flag.
3. CLA
4. OSR
5. HLT

Notice that subinstructions can only be selected from one group, group 1 or group 2. These different groups cannot be combined in one instruction. Possible combinations are a subset of

CLA, CLL, CMA, CML, IAC, (RAR, RAL, RTR, or RTL)

or

SMA, SZA, SNL, RSS, CLA, OSR, HLT

Any subset of the instructions may be selected, but only one of the RAR, RAL, RTR, or RTL subinstructions may be selected per operate instruction.

Bit 0 of a group 2 operate instruction is always zero. Setting this bit to one (leaving bits 11, 10, 9, and 8 one) specifies an additional set of instructions which are executed by an Extended Arithmetic Element (EAE) for doing multiplies, divides, and shifts. The EAE is an optional feature of the PDP-8 (and costs extra).

**Assembly language**

Several assembly languages for the PDP-8 exist. One is the PAL-III assembler. It is extremely simple, since the assembler must run on such a small computer. Most assembly language statements are of the form:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| label, | opcode | I | operand | / comments |

Any field may be omitted. A label, if it occurs, is the first symbol on the line and is followed by a comma. Symbols can be up to six characters long, must start with a letter, and cannot be opcodes or the letter I. The opcodes are any of the mnemonics presented in the last section plus a few extras. Additional mnemonic instructions have been added to the assembler for commonly used combinations of the operate instruction.

|  |  |
| --- | --- |
| NOP | No instructions selected; no operation |
| SPA | SMA RSS (Skip on Positive A register) |
| SNA | SZA RSS (Skip on Nonzero A register) |
| SZL | SNL RSS (Skip on Zero Link) |
| SKP | RSS (Always skip) |
| CIA | CMA IAC (Complement and Increment A register) |
| LAS | CLA OSR (Load A register from Switch Register) |
| STL | CLL CML (Set Link) |

Some mnemonics are also added for common I/O instructions and EAE instructions.

Comments are indicated by the slash and continue to the end of the card. Indirect addressing is indicated by the letter I. The symbol "." (period) refers to the value of the location counter. Fields can be either symbols or octal numbers or the period.

Only two pseudo-instructions are recognized. The ORIG function in MIX is accomplished in the PDP-8 by an assembly language statement of the form,

|  |  |
| --- | --- |
|  | *\*nnnn* |

where *nnnn* is an octal number. This resets the value of the location counter to *nnnn*. The END function of MIX is simply a card with a $ on it for PAL-III. Constants can be defined by omitting an opcode, as

|  |  |  |
| --- | --- | --- |
| C100, | 144 | / CONSTANT 100 |

Remember that all constants are octal. There are no literals, local symbols, character strings (ALF), or EQUs.

The PAL-III assembler is a two-pass assembler (or three-pass if you want a listing). Only one symbol table is used, including opcodes and user symbols into this one table. (This is why you cannot use I or mnemonics for labels). The assembly language is admittedly very simple, but even so, it is an improvement over machine language and has enough features to allow reasonable assembly language programs to be written.

**Programming techniques**

Even though there are very few instructions on the PDP-8, there are enough. Below we list some of the fundamental programming techniques.

**Loading**

One major obvious lack is the absence of a load instruction. Loading the A register is done by first clearing the A register and then adding the storage location to be loaded. For example, to load the A register with the contents of location X, either

|  |  |  |
| --- | --- | --- |
|  | DCA | some place |
|  | TAD | X |

or

|  |  |  |
| --- | --- | --- |
|  | CLA |  |
|  | TAD | X |

**Subtraction**

Subtraction is done by complementing and adding. To subtract Y from X, and leave the difference in z

|  |  |  |  |
| --- | --- | --- | --- |
|  | CLA |  | / A IS ZERO |
|  | TAD | Y | / 0 + Y = Y |
|  | CMA IAC |  | / -Y |
|  | TAD | X | / X - Y |
|  | DCA | Z | / Z = X - Y, A = 0 |

To subtract X from the A register can be done in two ways: (1) simple

|  |  |  |  |
| --- | --- | --- | --- |
|  | DCA | TEMP | / SAVE A REGISTER |
|  | TAD | X | / X |
|  | CMA IAC |  | / -X |
|  | TAD | TEMP | / A-X |

or (2) clever

|  |  |  |  |
| --- | --- | --- | --- |
|  | CMA IAC |  | / -A |
|  | TAD | X | / X-A |
|  | CMA IAC |  | / A-X |

**Comparisons**

To compare two numbers X and Y, we use the old "subtract and compare difference to zero" trick.

|  |  |  |  |
| --- | --- | --- | --- |
|  | CLA |  |  |
|  | TAD | Y | / A = Y |
|  | CMA IAC |  | / A = -Y |
|  | TAD | X | / X-Y |
|  | SNA |  |  |
|  | JMP | EQUAL | / X-Y = 0, X=Y |
|  | SMA |  |  |
|  | JMP | GREATER | / X-Y > 0, X > Y |
|  | JMP | LESS | / X-Y < 0, X<Y |

**Loops**

The ISZ instruction is the easy way to execute a loop. For example to search a list of numbers starting at location X, for one equal to the A register with the length of the list in the variable N

|  |  |  |  |
| --- | --- | --- | --- |
|  | DCA | TEMP | / SAVE A |
|  | TAD | N |  |
|  | CMA IAC |  | / -N FOR ISZ |
|  | DCA | LOOPN |  |
| / |  |  |  |
| LOOP, | TAD | X |  |
|  | CMA IAC |  | / -X |
|  | TAD | TEMP | / A-X |
|  | SNA CLA |  | / SKIP IF NOT EQUAL, CLEAR A |
|  | JMP | FOUND | / FOUND IT |
|  | ISZ | LOOP | / MODIFY ADDRESS OF X |
|  | ISZ | LOOPN | / TEST END OF LOOP |
|  | JMP | LOOP |  |
|  | ... | ... | / NOT FOUND IN LIST |

Notice that we use the fact that the SNA test is done before the CLA to assure that the test is done correctly and that the A register is zero when we get back to LOOP. Also notice that we are using *address modification*. There are no index registers on the PDP-8, so addressing through a loop must be done either by modifying the address portion of an instruction (as above) or by indirection, as follows.

|  |  |  |  |
| --- | --- | --- | --- |
|  | DCA | TEMP | / SAVE A FOR COMPARISON |
|  | TAD | N |  |
|  | CMA IAC |  |  |
|  | DCA | LOOPN | / LOOP COUNTER = -N |
|  | TAD | XADR | / ADDRESS OF LIST |
|  | DCA | ADDR | / FOR INDIRECTION |
| LOOP, | TAD I | ADDR | / INDIRECT LOAD |
|  | CMA IAC |  | / |
|  | TAD | TEMP | / A REGISTER - X |
|  | SNA CLA |  |  |
|  | JMP | EQUAL |  |
|  | ISZ | ADDR | / INCREMENT ADDRESS |
|  | ISZ | LOOPN | / LOOP COUNTER |
|  | JMP | LOOP |  |
|  | ... | ... | / NOT FOUND IN LIST |

where XADR has the address of X as its contents.

A special feature on the PDP-8 is *auto-Indexing*. In page 0, locations 0010 through 0017 (octal) automatically increment their contents by one *before* they are used as the address of the operand when it is addressed indirectly. Thus, if we assign ADDR to location 0010 in the above code, we do not need the ISZ ADDR, since this will be done automatically. We do need to store, not the address of X, but one less than the address of X (since auto-indexing is done before using the address for indirection).

**Subroutines**

With as simple a machine as the PDP-8, subroutines are used a lot. Subroutine linkage is done by the JMS, which stores the return address in its operand and starts execution at the next location. For example, a subroutine to decrement one from the A register:

|  |  |  |  |
| --- | --- | --- | --- |
| DEC1, | NOP |  | / WILL BE RETURN ADDRESS |
|  | CMA IAC |  | / -K |
|  | CMA |  | / -(-K) - 1 |
|  | JMP I | DEC1 | / INDIRECT RETURN |

The call is simply

|  |  |  |  |
| --- | --- | --- | --- |
|  | JMS | DEC1 |  |

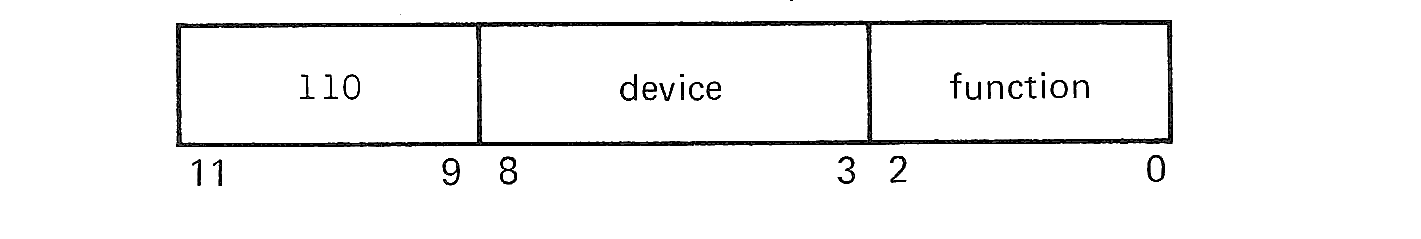
We can make this a decrement and skip if zero by

|  |  |  |  |
| --- | --- | --- | --- |
| DSZ, | NOP |  | / RETURN ADDRESS |
|  | CMA IAC |  |  |
|  | CMA |  |  |
|  | SNA |  |  |
|  | ISZ | DSZ | / INCREMENT ADDRESS IF ZERO |
|  | JMP I | DSZ |  |

Parameters are almost always passed by reference, after the call to the subroutine, or in global variables on the zero page.

**Input/output**

The one instruction we have ignored so far is the Input/Output transfer (IOT) instruction. It has an opcode of 6 and two fields, a 6-bit device number and a 3-bit function field. A device can have up to eight different functions and each device can have any eight functions which are appropriate for that device. Each device normally has a one-bit *device flag*. If the flag is 0, the device is busy; if the flag is 1, the device is ready. The ASCII character code is used. Most I/O transfers go through the A register, one character at a time.

  
**FIGURE 10.5** Instruction format for opcode 6, I/O instructions.

To illustrate the use of the input/output instructions, consider the functions of a Teletype input keyboard.

|  |  |  |
| --- | --- | --- |
| Function | Mnemonic | Explanation |
| 0 | KCF | Clear the flag, but do not start the device |
| 1 | KSF | Skip next instruction if flag is 1 |
| 2 | KCC | Clear the A register and flag |
| 4 | KRS | Read a character from device into A register |
| 6 | KRB | Read a character into A register, clear flag |

For the Teletype printer,

|  |  |  |
| --- | --- | --- |
| Function | Mnemonic | Explanation |
| 0 | TFL | Set flag |
| 1 | TSF | Skip if Flag is 1 |
| 2 | TCF | Clear Flag |
| 4 | TPC | Output character from A register and start printing it |
| 6 | TLS | Clear Flag and Output Character |

To input one character from the keyboard and echo print it on the printer

|  |  |  |  |
| --- | --- | --- | --- |
|  | KCC |  | / CLEAR FLAG ON KEYBOARD |
|  | KSF |  | / WAIT UNTIL CHARACTER READ |
|  | JMP | .-1 |  |
|  | KRB |  | / READ CHARACTER INTO A |
|  | TLS |  | / OUTPUT CHARACTER TO PRINTER |
|  | TSF |  |  |
|  | JMP | .-1 | WAIT UNTIL DONE |

This program first clears the flag for the keyboard. Clearing the flag is a signal for the keyboard to input a character. When a key is hit on the keyboard, the keyboard reads the key, constructs the appropriate ASCII character code, and saves it in a buffer register. Then the flag is set. In the meantime, the CPU has been repetitively testing the flag, waiting for it to become set. When the flag is set, the CPU reads the character from the buffer register into the A register. Then it outputs this character to the buffer register for the printer, and clears the flag, telling the printer to print the character in its buffer register. The CPU waits until the printer signals that it has printed the character by setting the flag.

Normally, the program would try to overlap its input, output, and computing, of course.

Suppose we have several different I/O devices, *d1*, *d2*, *d3* and *d4*, and we want to do I/O on all of them simultaneously. We also have some computing to do. We can do all our I/O on each device one at a time or try to overlap them. Suppose we are inputting from *d1* and *d2* into buffers in memory and outputting from buffers to *d3* and *d4*. All of the devices operate at different speeds. If we program them as above for the Teletype we will spend most of our time in loops like

|  |  |  |  |
| --- | --- | --- | --- |
|  | KSF |  | / IS KEYBOARD READY |
|  | JMP | .-1 |  |

What we need is to test each device at regular intervals; if any device is ready, we will service it; if not, we will go compute for a while, and come back to check again later. The KSF and TSF commands are like Skip if ready, so we will say SKR *di* for device *di*. We can then write a subroutine

|  |  |  |  |
| --- | --- | --- | --- |
| POLL, | NOP |  | / RETURN ADDRESS |
|  | SKR | D1 | / IS D1 READY |
|  | SKP |  | / NO |
|  | JMS | SERVD1 | / YES, SERVICE D1 |
|  | SKR | D2 | / IS D2 READY |
|  | SKP |  | / NO |
|  | JMS | SERVD2 | / YES, SERVICE D2 |
|  | SKR | D3 | / IS D3 READY |
|  | SKP |  | / NO |
|  | JMS | SERVD3 | / YES, SERVICE D3 |
|  | SKR | D4 | / IS D4 READY |
|  | SKP |  | / NO |
|  | JMS | SERVD4 | / YES, SERVICE D4 |
|  | JMP I | POLL |  |

In our main program we can now add JMS POLL at regular intervals. The length of the interval depends upon how long we are willing to tolerate having an I/O device finish and not be served. In the worst case (must respond to each device finishing as soon as possible), this may be after each instruction.

|  |  |  |  |
| --- | --- | --- | --- |
|  | DCA | TEMP |  |
|  | JMS | POLL |  |
|  | TAD | N |  |
|  | JMS | POLL |  |
|  | CMA IAC |  |  |
|  | JMS | POLL |  |

This is called *polling*. Although it is better than busy loop waiting (JBUS \*), it takes a lot of time.

*Interrupts* do this polling in hardware. Each device has an interrupt request flag. The interrupt system can be either on or off. If it is off, execution is just as we have always thought it to be. If the interrupt system is on, however, the following changes take place (on the PDP-8).

After every instruction is executed, the CPU looks at all of the interrupt request flags. If they are all off, the CPU continues to the next instruction. If any flag is on, the CPU

1. executes a JMS 0, storing the program counter in location 0 and executing the instruction at location 1, and
2. turns the interrupt system off.

This allows the programmer to be informed immediately that one of the I/O devices needs attention. After the I/O device is serviced, and the programmer wishes to resume the computation which had been executing when the I/O interrupt occurred, it is necessary to only do a JMP I 0. Thus, an interrupt forces a subroutine jump to location 0.

The normal use of the interrupt system for I/O is,

1. Start all I/O devices.
2. Turn on the interrupt system. (On the PDP-8, the interrupt system is device 0, so I/O instructions are used to turn it on and off.)
3. Go do some computation, or twiddle your thumbs (JMP .) if you have nothing to do, while you wait for an interrupt.

When an interrupt occurs,

1. The address of the current instruction is stored in location 0. The interrupt system is turned off to prevent interrupting an interrupt.
2. The instruction in location 1 is executed. This is normally a JMP to an interrupt service routine.
3. Save all registers.
4. Determine what device caused the interrupt.
5. Service that device, possibly restarting it on something new (next char acter).
6. Check if any other devices want service too; if so, go back to 5.
7. Restore the registers.
8. Turn the interrupt system back on.
9. Return to the interrupted program by a JMP I 0.

The addition of an interrupt system to the design of a computer system is necessary if I/O is to be effectively overlapped with computation and other I/O. Almost all modern computers have an interrupt system. The major features of the interrupt system are that it can be turned on or off, and that interrupts cause a forced jump to some location in such a way that the interrupted program can be restarted without knowing that it was interrupted. Thus, the background computation can proceed correctly, without special programming being necessary because of the frequent interrupts of the CPU to service I/O devices.

The best source of more complete information on the PDP-8 is from its manufacturer, Digital Equipment Corporation. DEC publishes several manuals about the PDP-8. Of particular interest are the "Introduction to Programming" and "Small Computer Handbook" manuals.

**EXERCISES**

1. Describe the memory and registers of the PDP-8. What is the word size? What is the address size?
2. How is the memory of the PDP-8 logically organized? Describe the effective address calculation for a memory reference instruction.
3. The PDP-8 has only a 3-bit opcode. Does this mean that it only has eight instructions? If so, why are there more than eight mnemonics in the assembly language?
4. Are all of the instructions for the PDP-8 necessary, or could the number of instructions be reduced even more? For example, are the ISZ and JMS instructions really necessary? If not, why do you think they were included in the instruction set of the PDP-8?
5. What is the meaning of the PDP-8 instructions
   1. CMA, IAC
   2. SMA, CLA, IAC
6. We wish to test the high-order bit of the switch register on the PDP-8. One student wrote

|  |  |  |  |
| --- | --- | --- | --- |
|  | CLA, OSR, SMA, RSS |  |  |
|  | (JMP for sign bit on) |  |  |

1. Why does this not work?
2. The MIX computer is much more powerful than the PDP-8 because the MIX computer has a much larger instruction set. To show this, consider both the MIX code and the PDP-8 code needed to jump to NNEG if a location labeled TEA is nonnegative and jump to NGE is not. The MIX code is

|  |  |  |  |
| --- | --- | --- | --- |
|  | LDA | TEA |  |
|  | JANN | NNEG |  |
|  | JMP | NGE |  |

1. Write the PDP-8 code to do this same function. (Assume the A register may have any initial value.)
2. The last problem showed that the MIX computer is better than the PDP-8. However, for some purposes the PDP-8 may be better. Write the MIX code and the PDP-8 code which would add one to a variable TOPS and jump to LOOP if the resulting sum (which should be stored back in TOPS) is nonzero, or continues on at the next instruction (falls through) if TOPS is zero.
3. Write a subroutine for the PDP-8 to add the elements of an array. Gall your subroutine SUM. Define an appropriate calling sequence. How does your code compare with the subroutine SUMMER in Chapter 6?