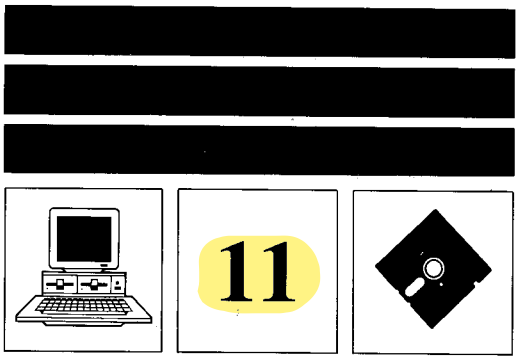


11

SAP-2

SAP-1 is a computer because it stores a program and data before calculations begin; then it automatically carries out the program instructions without human intervention. And yet, SAP-1 is a primitive computing machine. It compares to a modern computer the way a Neanderthal human would compare to a modern person. Something is missing, something found in every modern computer.

SAP-2 is the next step in the evolution toward modern computers because it includes *jump* instructions. These new instructions force the computer to repeat or skip part of a program. As you will discover, jump instructions open up a whole new world of computing power.

11-1 BIDIRECTIONAL REGISTERS

To reduce the wiring capacitance of SAP-2, we will run only one set of wires between each register and the bus. Figure 11-1*a* shows the idea. The input and output pins are shorted; only one group of wires is connected to the bus.

Does this shorting the input and output pins ever cause trouble? No. During a computer run, either *LOAD* or *ENABLE* may be active, but not both at the same time. An active *LOAD* means that a binary word flows from the bus to the register input; during a load operation, the output lines are floating. On the other hand, an active *ENABLE* means that a binary word flows from the register to the bus; in this case, the input lines float.

The IC manufacturer can internally connect the input and output pins of a three-state register. This not only reduces the wiring capacitance; it also reduces the number of I/O pins. For instance, Fig. 11-1*b* has four I/O pins instead of eight.

Figure 11-1*c* is the symbol for a three-state register with internally connected input and output pins. The double-headed arrow reminds us that the path is *bidirectional*; data can move either way.

11-2 ARCHITECTURE

Figure 11-2 shows the architecture of SAP-2. All register outputs to the W bus are three-state; those not connected to the bus are two-state. As before, the controller-sequencer sends control signals (not shown) to each register. These control signals load, enable, or otherwise prepare the register for the next positive clock edge. A brief description of each box is given now.

Input Ports

SAP-2 has two input ports, numbered 1 and 2. A hexadecimal keyboard encoder is connected to port 1. It allows us to enter hexadecimal instructions and data through port 1. Notice that the hexadecimal keyboard encoder sends a *READY* signal to bit 0 of port 2. This signal indicates when the data in port 1 is valid.

Also notice the *SERIAL IN* signal going to pin 7 of port 2. A later example will show you how to convert serial input data to parallel data.

Program Counter

This time, the program counter has 16 bits; therefore, it can count from

PC = 0000 0000 0000 0000

to

PC = 1111 1111 1111 1111

This is equivalent to 0000H to FFFFH, or decimal 0 to 65,535.

A low $\overline{CLR}$ signal resets the PC before each computer run; so the data processing starts with the instruction stored in memory location 0000H.

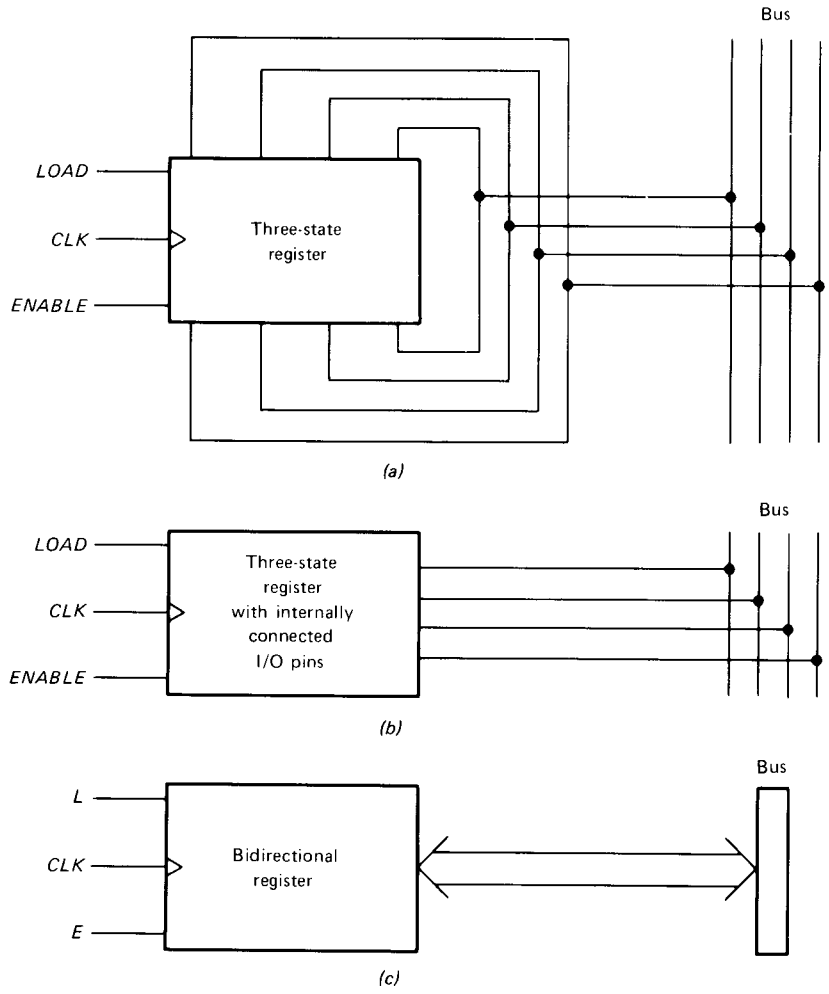


Fig. 11-1 Bidirectional register.

MAR and Memory

During the fetch cycle, the MAR receives 16-bit addresses from the program counter. The two-state MAR output then addresses the desired memory location. The memory has a 2K ROM with addresses of 0000H to 07FFH. This ROM contains a program called a *monitor* that initializes the computer on power-up, interprets the keyboard inputs, and so forth. The rest of the memory is a 62K RAM with addresses from 0800H to FFFFH.

Memory Data Register

The memory data register (MDR) is an 8-bit buffer register. Its output sets up the RAM. The memory data register receives data from the bus before a write operation, and it sends data to the bus after a read operation.

Instruction Register

Because SAP-2 has more instructions than SAP-1, we will use 8 bits for the op code rather than 4. An 8-bit op code can accommodate 256 instructions. SAP-2 has only 42 instructions, so there will be no problem coding them with 8 bits. Using an 8-bit op code also allows upward compatibility with the 8080/8085 instruction set because it is based on an 8-bit op code. As mentioned earlier, all SAP instructions are identical with 8080/8085 instructions.

Controller-Sequencer

The controller-sequencer produces the control words or microinstructions that coordinate and direct the rest of the computer. Because SAP-2 has a bigger instruction set, the controller-sequencer has more hardware. Although the **CON** word is bigger, the idea is the same: the control word or microinstruction determines how the registers react to the next positive clock edge.

Accumulator

The two-state output of the accumulator goes to the ALU; the three-state output to the W bus. Therefore, the 8-bit word in the accumulator continuously drives the ALU, but this same word appears on the bus only when E_A is active.

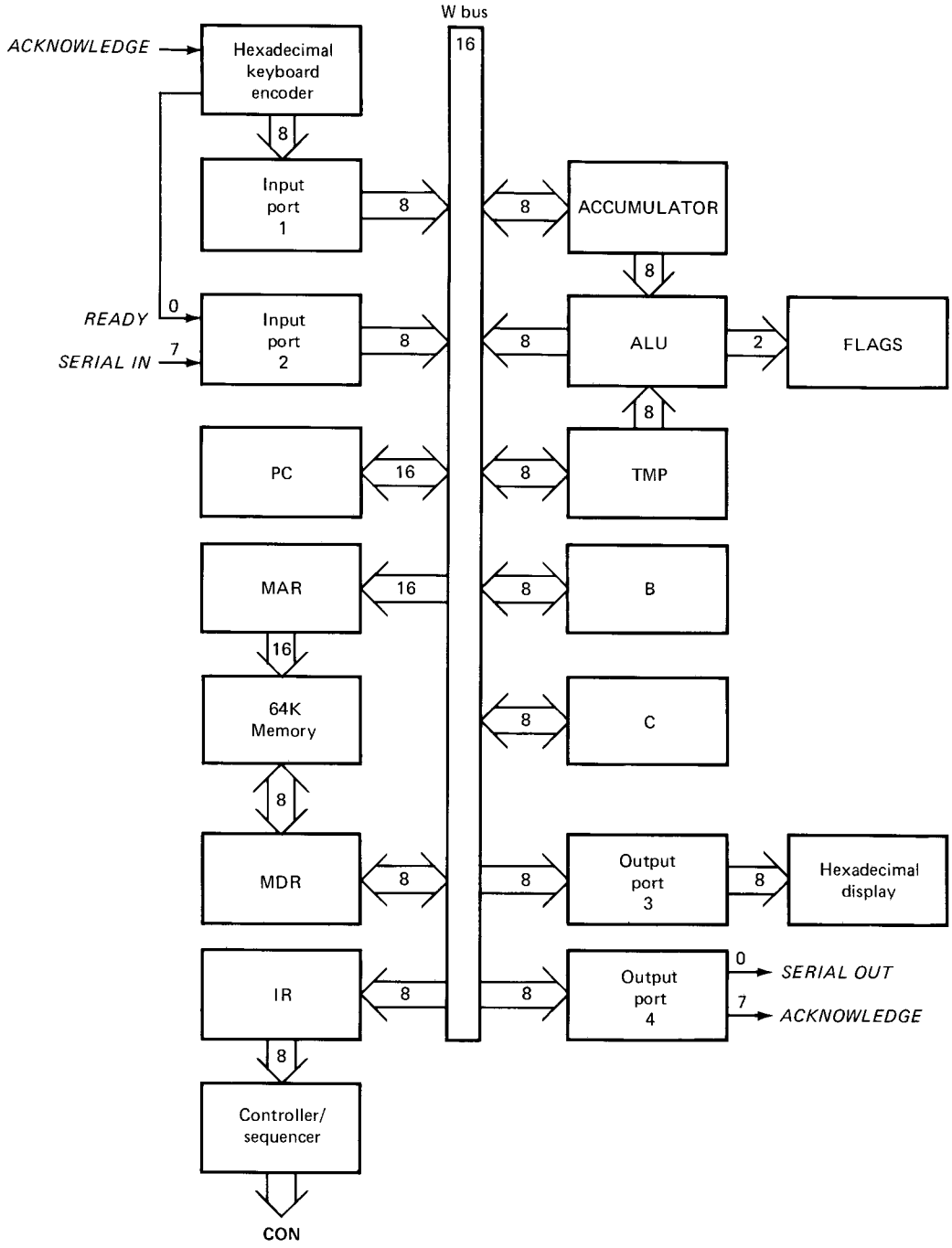


Fig. 11-2 SAP-2 block architecture.

ALU and Flags

Standard ALUs are commercially available as integrated circuits. These ALUs have 4 or more control bits that determine the arithmetic or logic operation performed on words **A** and **B**. The ALU used in SAP-2 includes arithmetic and logic operations.

In this book a *flag* is a flip-flop that keeps track of a changing condition during a computer run. The SAP-2 computer has two flags. The *sign flag* is set when the accumulator contents become negative during the execution of some instructions. The *zero flag* is set when the accumulator contents become zero.

TMP, B, and C Registers

Instead of using the B register to hold the data being added or subtracted from the accumulator, a *temporary* (TMP) register is used. This allows us more freedom in using the B register. Besides the TMP and B registers, SAP-2 includes a C register. This gives us more flexibility in moving data during a computer run.

Output Ports

SAP-2 has two output ports, numbered 3 and 4. The contents of the accumulator can be loaded into port 3, which drives a hexadecimal display. This allows us to see the processed data.

The contents of the accumulator can also be sent to port 4. Notice that pin 7 of port 4 sends an *ACKNOWLEDGE* signal to the hexadecimal encoder. This *ACKNOWLEDGE* signal and the *READY* signal are part of a concept called *handshaking*, to be discussed later.

Also notice the *SERIAL OUT* signal from pin 0 of port 4; one of the examples will show you how to convert parallel data in the accumulator into serial output data.

11-3 MEMORY-REFERENCE INSTRUCTIONS

The SAP-2 fetch cycle is the same as before. T_1 is the address state, T_2 is the increment state, and T_3 is the memory state. All SAP-2 instructions therefore use the memory during the fetch cycle because a program instruction is transferred from the memory to the instruction register.

During the execution cycle, however, the memory may or may not be used; it depends on the type of instruction that has been fetched. A memory-reference instruction (MRI) is one that uses the memory during the execution cycle.

The SAP-2 computer has an instruction set with 42 instructions. What follows is a description of the memory-reference instructions.

LDA and STA

LDA has the same meaning as before: *load the accumulator* with the addressed memory data. The only difference is that more memory locations can be accessed in SAP-2 because the addresses are from 0000H to FFFFH. For example, LDA 2000H means to load the accumulator with the contents of memory location 2000H.

To distinguish the different parts of an instruction, the mnemonic is sometimes called the *op code* and the rest of the instruction is known as the *operand*. With LDA 2000H, LDA is the op code and 2000H is the operand. Therefore, "op code" has a double meaning in microprocessor work; it may stand for the mnemonic or for the binary code used to represent the mnemonic. The intended meaning is clear from the context.

STA is a mnemonic for *store the accumulator*. Every STA instruction needs an address. STA 7FFFH means to store the accumulator contents at memory location 7FFFH. If

$$\mathbf{A} = 8AH$$

the execution of STA 7FFFH stores 8AH at address 7FFFH.

MVI

MVI is the mnemonic for *move immediate*. It tells the computer to load a designated register with the byte that immediately follows the op code. For instance,

```
MVI A,37H
```

tells the computer to load the accumulator with 37H. After this instruction has been executed, the binary contents of the accumulator are

$$\mathbf{A} = 0011\ 0111$$

You can use MVI with the A, B, and C registers. The formats for these instructions are

```
MVI A,byte
MVI B,byte
MVI C,byte
```

Op Codes

Table 11-1 shows the op codes for the SAP-2 instruction set. These are the 8080/8085 op codes. As you can see, 3A is the op code for LDA, 32 is the op code for STA, etc. Refer to this table in the remainder of this chapter.

EXAMPLE 11-1

Show the mnemonics for a program that loads the accumulator with 49H, the B register with 4AH, and the C register with 4BH; then have the program store the accumulator data at memory location 6285H.

SOLUTION

Here's one program that will work:

Mnemonics

```
MVI A,49H
MVI B,4AH
MVI C,4BH
STA 6285H
HLT
```

The first three instructions load 49H, 4AH, and 4BH into the A, B, and C registers. STA 6285H stores the accumulator contents at 6285H.

Note the use of HLT in this program. It has the same meaning as before: halt the data processing.

TABLE 11-1. SAP-2 OP CODES

Instruction	Op Code	Instruction	Op Code
ADD B	80	MOV B,A	47
ADD C	81	MOV B,C	41
ANA B	A0	MOV C,A	4F
ANA C	A1	MOV C,B	48
ANI byte	E6	MVI A,byte	3E
CALL address	CD	MVI B,byte	06
CMA	2F	MVI C,byte	0E
DCR A	3D	NOP	00
DCR B	05	ORA B	B0
DCR C	0D	ORA C	B1
HLT	76	ORI byte	F6
IN byte	DB	OUT byte	D3
INR A	3C	RAL	17
INR B	04	RAR	1F
INR C	0C	RET	C9
JM address	FA	STA address	32
JMP address	C3	SUB B	90
JNZ address	C2	SUB C	91
JZ address	CA	XRA B	A8
LDA address	3A	XRA C	A9
MOV A,B	78	XRI byte	EE
MOV A,C	79		

EXAMPLE 11-2

Translate the foregoing program into 8080/8085 machine language using the op codes of Table 11-1. Start with address 2000H.

SOLUTION

Address	Contents	Symbolic
2000H	3EH	MVI A,49H
2001H	49H	
2002H	06H	MVI B,4AH
2003H	4AH	
2004H	0EH	MVI C,4BH
2005H	4BH	
2006H	32H	STA 6285H
2007H	85H	
2008H	62H	
2009H	76H	HLT

There are a couple of new ideas in this machine-language program. With the

MVI A,49H

instruction, notice that the op code goes into the first address and the byte into the second address. This is true of all 2-byte instructions: op code into the first available memory location and byte into the next.

The instruction

STA 6285H

is a 3-byte instruction (1 byte for the op code and 2 for the address). The op code for STA is 32H. This byte goes into the first available memory location, which is 2006H. The address 6285H has 2 bytes. The lower byte 85H goes into the next memory location, and the upper byte 62H into the next location.

Why does the address get programmed with the lower byte first and the upper byte second? This is a peculiarity of the original 8080 design. To keep upward compatibility, the 8085 and some other microprocessors use the same scheme: lower byte into lower memory, upper byte into upper memory.

The last instruction HLT has an op code of 76H, stored in memory location 2009H.

In summary, the MVI instructions are 2-byte instructions, the STA is a 3-byte instruction, and the HLT is a 1-byte instruction.

11-4 REGISTER INSTRUCTIONS

Memory-reference instructions are relatively slow because they require more than one memory access during the instruction cycle. Furthermore, we often want to move data directly from one register to another without having to go through the memory. What follows are some of the SAP-2 register instructions, designed to move data from one register to another in the shortest possible time.

MOV

MOV is the mnemonic for *move*. It tells the computer to move data from one register to another. For instance,

MOV A,B

tells the computer to move the data in the B register to the accumulator. The operation is nondestructive, meaning that the data in B is copied but not erased. For example, if

A = 34H and **B** = 9DH

then the execution of MOV A,B results in

A = 9DH
B = 9DH

You can move data between the A, B, and C registers. The formats for all MOV instructions are

```
MOV A,B
MOV A,C
MOV B,A
MOV B,C
MOV C,A
MOV C,B
```

These instructions are the fastest in the SAP-2 instruction set, requiring only one machine cycle.

ADD and SUB

ADD stands for add the data in the designated register to the accumulator. For instance,

```
ADD B
```

means to add the contents of the B register to the accumulator. If

A = 04H and **B** = 02H

then the execution of ADD B results in

A = 06H

Similarly, SUB means subtract the data in the designated register from the accumulator. SUB C will subtract the contents of the C register from the accumulator.

The formats for the ADD and SUB instructions are

```
ADD B
ADD C
SUB B
SUB C
```

INR and DCR

Many times we want to increment or decrement the contents of one of the registers. INR is the mnemonic for *increment;* it tells the computer to increment the designated register. DCR is the mnemonic for *decrement,* and it instructs the computer to decrement the designated register. The formats for these instructions are

```
INR A
INR B
INR C
DCR A
DCR B
DCR C
```

As an example, if

B = 56H and **C** = 8AH

then the execution of INR B results in

B = 57H

and the execution of a DCR C produces

C = 89H

EXAMPLE 11-3

Show the mnemonics for adding decimal 23 and 45. The answer is to be stored at memory location 5600H. Also, the answer incremented by 1 is to be stored in the C register.

SOLUTION

As shown in Appendix 2, decimal 23 and 45 are equivalent to 17H and 2DH. Here is a program that will do the job:

Mnemonics

```
MVI A,17H
MVI B,2DH
ADD B
STA 5600H
INR A
MOV C,A
HLT
```

EXAMPLE 11-4

To *hand-assemble* a program means to translate a source program into a machine-language program by hand rather than machine. Hand-assemble the program of the preceding example starting at address 2000H.

SOLUTION

Address	Contents	Symbolic
2000H	3EH	MVI A,17H
2001H	17H	
2002H	06H	MVI B,2DH
2003H	2DH	
2004H	80H	ADD B
2005H	32H	STA 5600H
2006H	00H	
2007H	56H	
2008H	3CH	INR A
2009H	4FH	MOV C,A
200AH	76H	HLT

Notice that the ADD, INR, MOV, and HLT instructions are 1-byte instructions; the MVI instructions are 2-byte instructions, and the STA is a 3-byte instruction.

11-5 JUMP AND CALL INSTRUCTIONS

SAP-2 has four *jump* instructions; these can change the program sequence. In other words, instead of fetching the next instruction in the usual way, the computer may jump or *branch* to another part of the program.

JMP

To begin with, JMP is the mnemonic for jump; it tells the computer to get the next instruction from the designated memory location. Every JMP instruction includes an address that is loaded into the program counter. For instance,

JMP 3000H

tells the computer to get the next instruction from memory location 3000H.

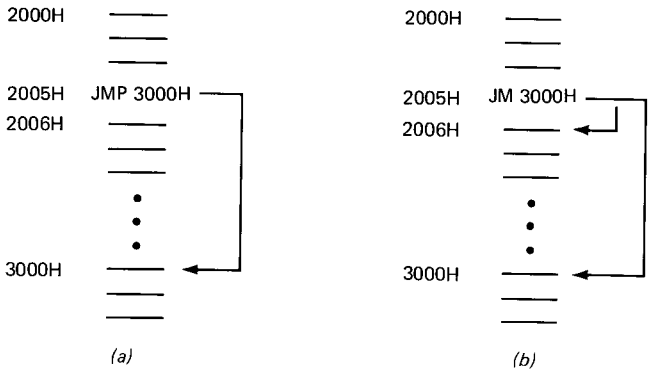


Fig. 11-3 (*a*) Unconditional jump; (*b*) conditional jump.

Here is what happens. Suppose JMP 3000H is stored at 2005H, as shown in Fig. 11-3*a*. At the end of the fetch cycle, the program counter contains

PC = 2006H

During the execution cycle, the JMP 3000H loads the program counter with the designated address:

PC = 3000H

When the next fetch cycle begins, the next instruction comes from 3000H rather than 2006H (see Fig. 11-3*a*).

JM

SAP-2 has two flags called the *sign flag* and the *zero flag*. During the execution of some instructions, these flags will be set or reset, depending on what happens to the accumulator contents. If the accumulator contents become negative, the sign flag will be set; otherwise, the sign flag is cleared. Symbolically,

$$S = \begin{cases} 0 & \text{if } A \geq 0 \\ 1 & \text{if } A < 0 \end{cases}$$

where S stands for sign flag. The sign flag will remain set or clear until another operation that affects the flag.

JM is a mnemonic for *jump if minus;* the computer will jump to a designated address if and only if the sign flag is set. As an example, suppose a JM 3000H is stored at 2005H. After this instruction has been fetched,

PC = 2006H

If $S = 1$, the execution of JM 3000H loads the program counter with

PC = 3000H

Since the program counter now points to 3000H, the next instruction will come from 3000H.

If the jump condition is not met ($S = 0$), the program counter is unchanged during the execution cycle. Therefore, when the next fetch cycle begins, the instruction is fetched from 2006H.

Figure 11-3*b* symbolizes the two possibilities for a JM instruction. If the minus condition is satisfied, the computer jumps to 3000H for the next instruction. If the minus condition is not satisfied, the program *falls through* to the next instruction.

JZ

The other flag affected by accumulator operations is the zero flag. During the execution of some instructions, the accumulator will become zero. To record this event, the zero flag is set; if the accumulator contents do not go to zero, the zero flag is reset. Symbolically,

$$Z = \begin{cases} 0 & \text{when } A \neq 0 \\ 1 & \text{when } A = 0 \end{cases}$$

JZ is the mnemonic for *jump if zero;* it tells the computer to jump to the designated address only if the zero flag is set. Suppose a JZ 3000H is stored at 2005H. If $Z = 1$ during the exection of JZ 3000H, the next instruction is fetched from 3000H. On the other hand, if $Z = 0$, the next instruction will come from 2006H.

JNZ

JNZ stands for *jump if not zero*. In this case, we get a jump when the zero flag is clear and no jump when it is set. Suppose a JNZ 7800H is stored at 2100H. If $Z = 0$, the next instruction will come from 7800H; however, if $Z = 1$, the program falls through to the instruction at 2101H.

JM, JZ, and JNZ are called *conditional jumps* because the program jump occurs only if certain conditions are satisfied. On the other hand, JMP is *unconditional;* once this instruction is fetched, the execution cycle always jumps the program to the specified address.

CALL and RET

A *subroutine* is a program stored in the memory for possible use in another program. Many microcomputers have subroutines for finding sines, cosines, tangents, logarithms, square roots, etc. These subroutines are part of the software supplied with the computer.

CALL is the mnemonic for *call the subroutine*. Every CALL instruction must include the starting address of the desired subroutine. For instance, if a square-root subroutine starts at address 5000H and a logarithm subroutine at 6000H, the execution of

CALL 5000H

will jump to the square-root subroutine. On the other hand, a

CALL 6000H

produces a jump to the logarithm subroutine.

RET stands for *return*. It is used at the end of every subroutine to tell the computer to go back to the original program. A RET instruction is to a subroutine as a HLT is to a program. Both tell the computer that something is finished. If you forget to use a RET at the end of a subroutine, the computer cannot get back to the original program and you will get computer trash.

When a CALL is executed in the SAP-2 computer, the contents of the program counter are automatically saved in memory locations FFFEH and FFFFH (the last two memory locations). The CALL address is then loaded into the program counter, so that execution begins with the first instruction in the subroutine. After the subroutine is finished, the RET instruction causes the address in memory locations FFFEH and FFFFH to be loaded back into the program counter. This returns control to the original program.

Figure 11-4 shows the program flow during a subroutine. The CALL 5000H sends the computer to the subroutine located at 5000H. After this subroutine has been completed, the RET sends the computer back to the instruction following the CALL.

CALL is unconditional, like JMP. Once a CALL has been fetched into the instruction register, the computer will jump to the starting address of the subroutine.

More on Flags

The sign or zero flag may be set or reset during certain instructions. Table 11-2 lists the SAP-2 instructions that can affect the flags. All these instructions use the accumulator during the execution cycle. If the accumulator goes negative or zero while one of these instructions is being executed, the sign or zero flag will be set.

For instance, suppose the instruction is ADD C. The contents of the C register are added to the accumulator contents. If the accumulator contents become negative or zero in the process, the sign or zero flag will be set.

A word about the INR and DCR instructions. Since these instructions use the accumulator to add or subtract 1 from the designated register, they also affect the flags. For instance, to execute a DCR C, the contents of the C register are decremented by sending these contents to the accumulator, subtracting 1, and sending the result back to the C register. If the accumulator goes negative while the DCR C is executed, the sign flag is set; if the accumulator goes to zero, the zero flag is set.

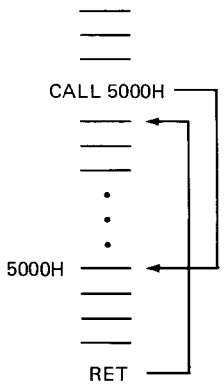


Fig. 11-4 CALL instruction.

TABLE 11-2. INSTRUCTIONS AFFECTING FLAGS

Instruction	Flags Affected
ADD	S, Z
SUB	S, Z
INR	S, Z
DCR	S, Z
ANA	S, Z
ORA	S, Z
XRA	S, Z
ANI	S, Z
ORI	S, Z
XRI	S, Z

EXAMPLE 11-5

Hand-assemble the following program starting at address 2000H:

```
MVI C,03H
DCR C
JZ 0009H
JMP 0002H
HLT
```

SOLUTION

Address	Contents	Symbolic
2000H	0EH	MVI C,03H
2001H	03H	
2002H	0DH	DCR C
2003H	CAH	JZ 2009H
2004H	09H	
2005H	20H	
2006H	C3H	JMP 2002H
2007H	02H	
2008H	20H	
2009H	76H	HLT

EXAMPLE 11-6

In the foregoing program, how many times is the DCR instruction executed?

SOLUTION

Figure 11-5 illustrates the program flow. Here is what happens. The MVI C,03H instruction loads the C register with 03H. DCR C reduces the contents to 02H. The contents are greater than zero; therefore, the zero flag is reset, and the JZ 2009H is ignored. The JMP 2002H returns the computer to the DCR C instruction.

The second time the DCR C is executed, the contents drop to 01H; the zero flag is still reset. JZ 2009H is again ignored, and the JMP 2002H returns the computer to DCR C.

The third DCR C reduces the contents to zero. This time the zero flag is set, and the JZ 2009H jumps the program to HLT instruction.

A *loop* is part of a program that is repeated. In this example, we have passed through the loop (DCR C and JZ 2009H) 3 times, as shown in Fig. 11-5. Note that the number of passes through the loop equals the number initially loaded into the C register. If we change the first instruction to

```
MVI C,07H
```

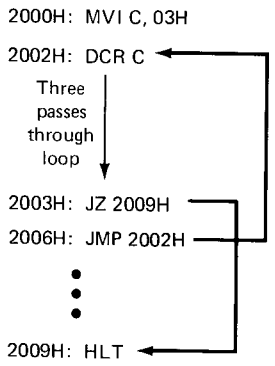


Fig. 11-5 Looping.

the computer will loop 7 times. Similarly, if we wanted to pass through the loop 200 times (equivalent to C8H), the first instruction would be

```
MVI C,C8H
```

The C register acts like a presettable down counter. This is why it is sometimes referred to as a *counter*.

The point to remember is this. We can set up a loop by using an MVI, DCR, JZ, and JMP in a program. The number loaded into the designated register (the counter) determines the number of passes through the loop. If we put new instructions inside the loop, these added instructions will be executed *X* times, the number preset into the counter.

EXAMPLE 11-7

When you buy a microcomputer, you often purchase software to do different jobs. One of the programs you can buy is an *assembler*. The assembler allows you to write programs in mnemonic form. Then the assembler converts these mnemonics into machine language. In other words, if you have an assembler, you no longer have to hand-assemble your programs; the computer does the work for you.

Show the assembly-language version of the program in Example 11-5. Include *labels* and *comments*.

SOLUTION

Label	Instruction	Comment
	MVI C,03H	;Load counter with decimal.3
REPEAT:	DCR C	;Decrement counter
	JZ END	;Test for zero
	JMP REPEAT	;Do it again
END:	HLT	

When you write a program, it helps to include your own comments about what the instruction is supposed to do. These comments jog your memory if you have to read the program months later. The first comment reminds us that we are presetting the down counter with decimal 3, the second comment reminds us that we are decrementing the counter, the third comment tells us that we are testing for zero before jumping, and the fourth comment tells us that the program will loop back.

When the assembler converts your source program into an object program, it ignores everything after the semicolon. Why? Because that's the way the assembler program is written. The semicolon is a coded way to tell the computer that your personal comments follow. (Remember the ASCII code. 3BH is the ASCII for a semicolon. When the assembler encounters 3BH in your source programs, it knows comments follow.)

Labels are another programming aid used with jumps and calls. When we write an assembly-language program, we often have no idea what address to use in a jump or call instruction. By using a label instead of a numerical address we can write programs that make sense to us. The assembler will keep track of our labels and automatically assign the correct addresses to them. This is a great laborsaving feature of an assembler.

For instance, when the assembler converts the foregoing program to machine language, it will replace JZ by CA (op code of Table 11-1) and END by the address of the HLT instruction. Likewise, it will replace JMP by C3 (op code) and REPEAT by the address of the DCR C instruction. The assembler determines the addresses of the HLT and JMP by counting the number of bytes needed by all instructions and figuring out where the HLT and DCR C instructions will be in the final assembled program.

All you have to remember is that you can make up any label you want for jump and call instructions. The same label followed by a colon is placed in front of the instruction you are trying to jump to. When the assembler converts your program into machine language, the colon tells it a label is involved.

One more point about labels. With SAP-2, the labels can be from one to six characters, the first of which must be a letter. Labels are usually words or abbreviations, but numbers can be included. The following are examples of acceptable labels:

REPEAT
DELAY
RDKBD
A34
B12C3

The first two are words; the third is an abbreviation for read the keyboard. The last two are labels that include numbers. The restrictions on length (no more than six characters) and starting character (must be letter) are typical of commercially available assemblers.

EXAMPLE 11-8

Show a program that multiplies decimal 12 and 8.

SOLUTION

The hexadecimal equivalents of 12 and 8 are 0CH and 08H. Let us set up a loop that adds 12 to the accumulator during each pass. If the computer loops 8 times, the accumulator contents will equal 96 (decimal) at the end of the looping.

Here's one assembly-language program that will do the job:

Label	Mnemonic	Comment
	MVI A,00H	;Clear accumulator
	MVI B,0CH	;Load decimal 12 into B
	MVI C,08H	;Preset counter with 8
REPEAT:	ADD B	;Add decimal 12
	DCR C	;Decrement the counter
	JZ DONE	;Test for zero
	JMP REPEAT	;Do it again
DONE:	HLT	;Stop it

The comments tell most of the story. First, we clear the accumulator. Next, we load decimal 12 into the B register. Then the counter is preset to decimal 8. These first three instructions are part of the initialization before entering a loop.

The ADD B begins the loop by adding decimal 12 to accumulator. The DCR C reduces the count to 7. Since the zero flag is clear, JZ DONE is ignored the first time through and the program flow returns to the ADD B instruction.

You should be able to see what will happen. ADD B is inside the loop and will be executed 8 times. After eight passes through the loop, the zero flag is set; then the JZ DONE will take the program out of the loop to the HLT instruction.

Since 12 is added 8 times,

$$12 + 12 + 12 + 12 + 12 + 12 + 12 + 12 = 96$$

(Because decimal 96 is equivalent to hexadecimal 60, the accumulator contains 0110 0000.) Repeated addition like this is equivalent to multiplication. In other words, adding 12 eight times is identical to 12×8. Most microprocessors do not have multiplication hardware; they only have an adder-subtracter like the SAP computer. Therefore, with the typical microprocessor, you have to use some form of programmed multiplication such as repeated addition.

EXAMPLE 11-9

Modify the foregoing multiply program by using a JNZ instead of a JZ.

SOLUTION

Look at this:

Label	Mnemonic	Comment
	MVI A,00H	;Clear accumulator
	MVI B,0CH	;Load decimal 12 into B
	MVI C,08H	;Preset counter with 8
REPEAT:	ADD B	;Add decimal 12
	DCR C	;Decrement the counter
	JNZ REPEAT	;Test for zero
	HLT	;Stop it

This is simpler. It eliminates one JMP instruction and one label. As long as the counter is greater than zero, the JNZ will force the computer to loop back to REPEAT. When the counter drops to zero, the program will fall through the JNZ to the HLT.

EXAMPLE 11-10

Hand-assemble the foregoing program starting at address 2000H.

SOLUTION

Address	Contents	Symbolic
2000H	3EH	MVI A,00H
2001H	00H	
2002H	06H	MVI B,0CH
2003H	0CH	
2004H	0EH	MVI, C,08H
2005H	08H	
2006H	80H	ADD B
2007H	0DH	DCR C
2008H	C2H	JNZ 2006H
2009H	06H	
200AH	20H	
200BH	76H	HLT

The first three instructions initialize the registers before the multiplication begins. If we change the initial values, we can multiply other numbers.

EXAMPLE 11-11

Change the multiplication part of the foregoing program into a subroutine located at starting address F006H.

SOLUTION

Address	Contents	Symbolic
F006H	80H	ADD B
F007H	0DH	DCR C
F008H	C2H	JNZ F006H
F009H	06H	
F00AH	F0H	
F00BH	C9H	RET

Here's what happened. The initializing instructions depend on the numbers we are multiplying, so they don't belong in the subroutine. The subroutine should contain only the multiplication part of the program.

In relocating the program we *mapped* (converted) addresses 2006H–200BH to F006H–F00BH. Also, the HLT was changed to a RET to get us back to the original program.

EXAMPLE 11-12

The multiply subroutine of the preceding example is used in the following program. What does the program do?

```
MVI A,00H
MVI B,10H
MVI C,0EH
CALL F006H
HLT
```

SOLUTION

Hexadecimal 10H is equivalent to decimal 16, and hexadecimal 0EH is equivalent to decimal 14. The first three instructions clear the accumulator, load the B register with decimal 16, and preset the counter to decimal 14. The CALL sends the computer to the multiply subroutine of the preceding example. When the RET is executed, the accumulator contents are E0H, which is equivalent to 224.

Incidentally, a *parameter* is a piece of data that the subroutine needs to work properly. The multiply subroutine located at F006H needs three parameters to work properly (*A*, *B*, and *C*). We pass these parameters to the multiply subroutine by clearing the accumulator, loading the B register with the multiplicand, and presetting the C register with the multiplier. In other words, we set $A = 00H$, $B = 10H$, and $C = 0EH$. Passing data to a subroutine in this way is called *register parameter passing*.

11-6 LOGIC INSTRUCTIONS

A microprocessor can do logic as well as arithmetic. What follows are the SAP-2 logic instructions. Again, they are a subset of the 8080/8085 instructions.

CMA

CMA stands for "complement the accumulator." The execution of a CMA inverts each bit in the accumulator, producing the 1's complement.

ANA

ANA means to AND *the accumulator* contents with the designated register. The result is stored in the accumulator. For instance,

ANA B

means to AND the contents of the accumulator with the contents of the B register. The ANDing is done on a bit-by-bit basis. For example, suppose the two registers contain

$$\mathbf{A} = 1100\ 1100 \qquad (11\text{-}1)$$

and

$$\mathbf{B} = 1111\ 0001 \qquad (11\text{-}2)$$

The execution of an ANA B results in

$$\mathbf{A} = 1100\ 0000$$

Notice that the ANDing is bitwise, as illustrated in Fig. 11-6. The ANDing is done on pairs of bits; A_7 is ANDed with B_7, A_6 with B_6, A_5 with B_5, and so on, with the result stored in the accumulator.

Two ANA instructions are available in SAP-2: ANA B and ANA C. Table 11-1 shows the op codes.

ORA

ORA is the mnemonic for OR *the accumulator* with the designated register. The two ORA instructions in SAP-2 are ORA B and ORA C. As an example, if the accumulator and B register contents are given by Eqs. 11-1 and 11-2, then executing ORA B gives

$$\mathbf{A} = 1111\ 1101$$

XRA

XRA means XOR *the accumulator* with the designated register. The SAP-2 instruction set contains XRA B and XRA C. If the accumulator and B contents are given by Eqs. 11-1 and 11-2, the execution of XRA B produces

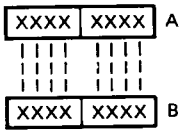


Fig. 11-6 Logic instructions are bitwise.

$$\mathbf{A} = 0011\ 1101$$

ANI

SAP-2 also has immediate logic instructions. ANI means AND *immediate*. It tells the computer to AND the accumulator contents with the byte that immediately follows the op code. For instance, if

$$\mathbf{A} = 0101\ 1110$$

the execution of ANI C7H will AND

0101 1110 with 1100 0111

to produce new accumulator contents of

$$\mathbf{A} = 0100\ 0110$$

ORI

ORI is the mnemonic for OR *immediate*. The accumulator contents are ORed with the byte that follows the op code. If

$$\mathbf{A} = 0011\ 1000$$

the execution of ORI 5AH will OR

0011 1000 with 0101 1010

to produce new accumulator contents of

0111 1010

XRI

XRI means XOR *immediate*. If

$$\mathbf{A} = 0001\ 1100$$

the execution of XRI D4H will XOR

0001 1100 with 1101 0100

to produce

$$\mathbf{A} = 1100\ 1000$$

11-7 OTHER INSTRUCTIONS

This section looks at the last of the SAP-2 instructions. Since these instructions don't fit any particular category, they are being collected here in a miscellaneous group.

NOP

NOP stands for *no operation*. During the execution of a NOP, all *T* states are do nothings. Therefore, no register changes occur during a NOP.

The NOP instruction is used to waste time. It takes four *T* states to fetch and execute the NOP instruction. By repeating a NOP a number of times, we can delay the data processing, which is useful in timing operations. For instance, if we put a NOP inside a loop and execute it 100 times, we create a time delay of 400 *T* states.

HLT

We have already used this. HLT stands for *halt*. It ends the data processing.

IN

IN is the mnemonic for *input*. It tells the computer to transfer data from the designated port to the accumulator. Since there are two input ports, you have to designate which one is being used. The format for an input operation is

IN byte

For instance,

IN 02H

means to transfer the data in port 2 to the accumulator.

OUT

OUT stands for *output*. When this instruction is executed, the accumulator word is loaded into the designated output port. The format for this instruction is

OUT byte

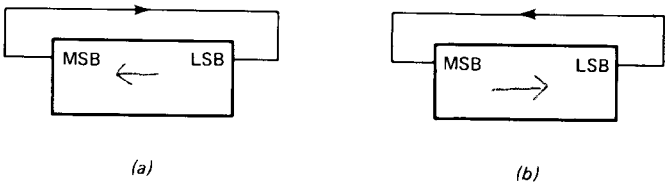
Since the output ports are numbered 3 and 4 (Fig. 11-2), you have to specify which port is to be used. For instance,

OUT 03H

will transfer the contents of the accumulator to port 3.

RAL

RAL is the mnemonic for *rotate the accumulator left*. This instruction will shift all bits to the left and move the MSB into the LSB position, as illustrated in Fig. 11-7*a*. As an example, suppose the contents of the accumulator are



(a) (b)

Fig. 11-7 Rotate instructions: (*a*) RAL; (*b*) RAR.

A = 1011 0100

Executing the RAL will produce

A = 0110 1001

As you see, all bits moved left, and the MSB went to the LSB position.

RAR

RAR stands for *rotate the accumulator right*. This time, the bits shift to the right, the LSB going to the MSB position, as shown in Fig. 11-7*b*. If

A = 1011 0100

the execution of a RAR will result in

A = 0101 1010

EXAMPLE 11-13

The bits in a byte are numbered 7 to 0 (MSB to LSB). Show a program that can input a byte from port 2 and determine if bit 0 is a 1 or a 0. If the bit is a 1, the program is to load the accumulator with an ASCII Y (yes). If the bit is a 0, the program should load the accumulator with an ASCII N (no). The yes or no answer is to be sent to output port 3.

SOLUTION

Label	Mnemonic	Comment
	IN 02H	;Get byte from port 2
	ANI 01H	;Isolate bit 0
	JNZ YES	;Jump if bit 0 is a 1
	MVI A,4EH	;Load N into accumulator
	JMP DONE	;Skip next instruction
YES:	MVI A,59H	;Load Y into accumulator
DONE:	OUT 03H	;Send answer to port 3
	HLT	

The IN 02H transfers the contents of input port 2 to the accumulator to get

$$\mathbf{A} = A_7A_6A_5A_4A_3A_2A_1A_0$$

The immediate byte in ANI 01H is

0000 0001

This byte is called a *mask* because its 0s will mask or blank out the corresponding high bits in the accumulator. In other words, after the execution of ANI 01H the accumulator contents are

$$\mathbf{A} = 0000\ 000A_0$$

If A_0 is 1, the JNZ YES will produce a jump to the MVI A,59H; this loads a 59H (the ASCII for Y) into the accumulator. If A_0 is 0, the program falls through to the MVI A,4EH. This loads the accumulator with the ASCII for N.

The OUT 03H loads the answer, either ASCII Y or N, into port 3. The hexadecimal display therefore shows either 59H or 4EH.

EXAMPLE 11-14

Instead of a parallel output at port 3, we want a serial output at port 4. Modify the foregoing program so that it converts the answer (59H or 4EH) into a serial output at bit 0, port 4.

SOLUTION

Label	Mnemonic	Comment
	IN 02H	
	ANI 01H	
	JNZ YES	
	MVI A,4EH	
	JMP DONE	
YES:	MVI A,59H	
DONE:	MVI C,08H	;Load counter with 8
AGAIN:	OUT 04H	;Send LSB to port 4
	RAR	;Position next bit
	DCR C	;Decrement count
	JNZ AGAIN	;Test count
	HLT	

In converting from parallel to serial data, the A_0 bit is sent first, then the A_1 bit, then the A_2 bit, and so on.

EXAMPLE 11-15

Handshaking is an interaction between a CPU and a peripheral device that takes place during an I/O data transfer.

In SAP-2 the handshaking takes place as follows. After you enter two digits (1 byte) into the hexadecimal encoder of Fig. 11-2, the data is loaded into port 1; at the same time, a *high READY* bit is sent to port 2.

Before accepting input data, the CPU checks the *READY* bit in port 2. If the *READY* bit is low, the CPU waits. If the *READY* bit is high, the CPU loads the data in port 1. After the data transfer is finished, the CPU sends a high *ACKNOWLEDGE* signal to the hexadecimal keyboard encoder; this resets the *READY* bit to 0. The *ACKNOWLEDGE* bit then is reset to low.

After you key in a new byte, the cycle starts over with new data going to the port 1 and a high *READY* bit to port 2.

The sequence of SAP-2 handshaking is

1. *READY* bit (bit 0, port 2) goes high.
2. Input the data in port 1 to the CPU.
3. *ACKNOWLEDGE* bit (bit 7, port 4) goes high to reset *READY* bit.
4. Reset the *ACKNOWLEDGE* bit.

Write a program that inputs a byte of data from port 1 using handshaking. Store the byte in the B register.

SOLUTION

Label	Mnemonic	Comment
STATUS:	IN 02H	;Input byte from port 2
	ANI 01H	;Isolate *READY* bit
	JZ STATUS	;Jump back if not ready
	IN 01H	;Transfer data in port 1
	MOV B,A	;Transfer from A to B
	MVI A,80H	;Set *ACKNOWLEDGE* bit
	OUT 04H	;Output high *ACKNOWLEDGE*
	MVI A,00H	;Reset *ACKNOWLEDGE* bit
	OUT 04H	;Output low *ACKNOWLEDGE*
	HLT	

If the *READY* bit is low, the ANI 01H will force the accumulator contents to go to zero. The JZ STATUS therefore will loop back to IN 02H. This looping will continue until the *READY* bit is high, indicating valid data in port 1.

When the *READY* bit is high, the program falls through the JZ STATUS to the IN 01H. This transfers a byte from port 1 to the accumulator. The MOV sends the byte to the B register.The MVI A,80H sets the *ACKNOWLEDGE* bit

(bit 7). The OUT 04H sends this high *ACKNOWLEDGE* to the hexadecimal encoder where the internal hardware resets the *READY* bit. Then the *ACKNOWLEDGE* bit is reset in preparation for the next input cycle.

11-8 SAP-2 SUMMARY

This section summarizes the SAP-2 *T* states, flags, and addressing modes.

T States

The SAP-2 controller-sequencer is microprogrammed with a variable machine cycle. This means that some instructions take longer than others to execute. As you recall, the idea behind microprogramming is to store the control routines in a ROM and access them as needed.

Table 11-3 shows each instruction and the number of *T* states needed to execute it. For instance, it takes four *T* states to execute the ADD B instruction, seven to execute the ANI byte, eighteen to execute the CALL, and so on. Knowing the number of *T* states is important in timing applications.

Notice that the JM instruction has *T* states of 10/7. This means it takes 10 *T* states when a jump occurs but only 7 without the jump. The same idea applies to the other conditional jumps; 10 *T* states for a jump, 7 with no jump.

Flags

As you know, the accumulator goes negative or zero during the execution of some instructions. This affects the sign and zero flags. Figure 11-8 shows the circuits used in SAP-2 to set the flags.

When the accumulator contents are negative, the leading bit A_7 is a 1. This sign bit drives the lower AND gate. When the accumulator contents are zero, all bits are zero and the output of the NOR gate is a 1. This NOR output drives the upper AND gate. If gating signal L_F is high, the flags will be updated to reflect the sign and zero condition of the accumulator. This means the Z_{FLAG} will be high when the accumulator contents are zero; the S_{FLAG} will be high when the accumulator contents are negative.

Not all instructions affect the flags. As shown in Table 11-3, the instructions that update the flags are ADD, ANA, ANI, DCR, INR, ORA, ORI, SUB, XRA, and XRI. Why only these instructions? Because the L_F signal of Fig. 11-8 is high only when these instructions are executed. This is accomplished by microprogramming an L_F bit for each instruction. In other words, in the control ROM we store a high L_F bit for the foregoing instructions, and a low L_F bit for all others.

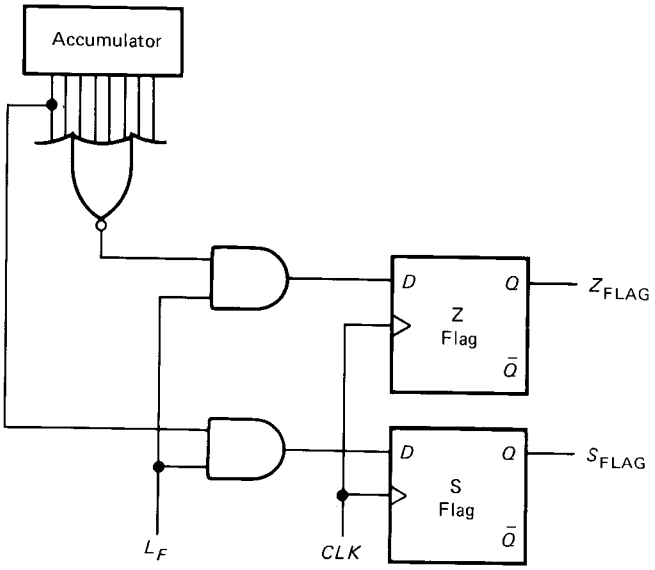


Fig. 11-8 Setting the flags.

Conditional Jumps

As mentioned earlier, the conditional jumps take ten *T* states when the jump occurs but only seven *T* states when no jump take place. Briefly, this is accomplished as follows. During the execution cycle the address ROM sends the computer to the starting address of a conditional-jump microroutine. The initial microinstruction looks at the flags and judges whether or not to jump. If a jump is indicated, the microroutine continues; otherwise, it is aborted and the computer begins a new fetch cycle.

Addressing Modes

The SAP-2 instructions access data in different ways. It is the operand that tells us how the data is to be accessed. For instance, the first instructions discussed were

```
LDA address
STA address
```

These are examples of *direct* addressing because we specify the address where the data is to be found.

Immediate addressing is different. Instead of giving an address for the data, we give the data itself. For instance,

```
MVI A,byte
```

accesses the data to be loaded into the accumulator by using the byte in memory that immediately follows the op code. Table 11-3 shows the other immediate instructions.

An instruction like

```
MOV A,B
```

TABLE 11-3. SAP-2 INSTRUCTION SET

Instruction	Op Code	*T* States	Flags	Addressing	Bytes
ADD B	80	4	S, Z	Register	1
ADD C	81	4	S, Z	Register	1
ANA B	A0	4	S, Z	Register	1
ANA C	A1	4	S, Z	Register	1
ANI byte	E6	7	S, Z	Immediate	2
CALL address	CD	18	None	Immediate	3
CMA	2F	4	None	Implied	1
DCR A	3D	4	S, Z	Register	1
DCR B	05	4	S, Z	Register	1
DCR C	0D	4	S, Z	Register	1
HLT	76	5	None	—	1
IN byte	DB	10	None	Direct	2
INR A	3C	4	S, Z	Register	1
INR B	04	4	S, Z	Register	1
INR C	0C	4	S, Z	Register	1
JM address	FA	10/7	None	Immediate	3
JMP address	C3	10	None	Immediate	3
JNZ address	C2	10/7	None	Immediate	3
JZ address	CA	10/7	None	Immediate	3
LDA address	3A	13	None	Direct	3
MOV A,B	78	4	None	Register	1
MOV A,C	79	4	None	Register	1
MOV B,A	47	4	None	Register	1
MOV B,C	41	4	None	Register	1
MOV C,A	4F	4	None	Register	1
MOV C,B	48	4	None	Register	1
MVI A,byte	3E	7	None	Immediate	2
MVI B,byte	06	7	None	Immediate	2
MVI C,byte	0E	7	None	Immediate	2
NOP	00	4	None	—	1
ORA B	B0	4	S, Z	Register	1
ORA C	B1	4	S, Z	Register	1
ORI byte	F6	7	S, Z	Immediate	2
OUT byte	D3	10	None	Direct	2
RAL	17	4	None	Implied	1
RAR	1F	4	None	Implied	1
RET	C9	10	None	Implied	1
STA address	32	13	None	Direct	3
SUB B	90	4	S, Z	Register	1
SUB C	91	4	S, Z	Register	1
XRA B	A8	4	S, Z	Register	1
XRA C	A9	4	S, Z	Register	1
XRI byte	EE	7	S, Z	Immediate	2

is an example of *register addressing*. The data to be loaded is stored in a CPU register rather than in the memory. Register addressing has the advantage of speed because fewer *T* states are needed for this type of instruction.

Implied addressing means that the location of the data is contained within the op code itself. For instance,

RAL

tells us to rotate the accumulator bits left. The data is in the accumulator; this is why no operand is needed with implied addressing.

Bytes

Each instruction occupies a number of bytes in the memory. SAP-2 instructions are either 1, 2, or 3 bytes long. Table 11-3 shows the length of each instruction. As you see, ADD instructions are 1-byte instructions, ANI instructions are 2-byte instructions, CALLs are 3-byte instructions, and so forth.

EXAMPLE 11-16

SAP-2 has a clock frequency of 1 MHz. This means that each *T* state has a duration of 1 μs. How long does it take to execute the following SAP-2 subroutine?

Label	Mnemonic	Comment
	MVI C,46H	;Preset count to decimal 70
AGAIN:	DCR C	;Count down
	JNZ AGAIN	;Test count
	NOP	;Delay
	RET	

SOLUTION

The MVI is executed once to initialize the count. The DCR is executed 70 times. The JNZ jumps back 69 times and falls through once. With the number of *T* states given in Table 11-3, we can calculate the total execution time of the subroutine as follows:

MVI:	$1 \times 7 \times 1\ \mu s = 7\ \mu s$	
DCR:	$70 \times 4 \times 1\ \mu s = 280$	
JNZ:	$69 \times 10 \times 1\ \mu s = 690$	(jump)
JNZ:	$1 \times 7 \times 1\ \mu s = 7$	(no jump)
NOP:	$1 \times 4 \times 1\ \mu s = 4$	
RET:	$1 \times 10 \times 1\ \mu s = 10$	
	$998\ \mu s \approx 1$ ms	

As you see, the total time needed to execute the subroutine is approximately 1 ms.

A subroutine like this can produce a time delay of 1 ms whenever it is called. There are many applications where you need a delay.

According to Table 11-3, the instructions in the foregoing subroutine have the following byte lengths:

Instruction	MVI	DCR	JNZ	NOP	RET
Bytes	2	1	3	1	1

The total byte length of the subroutine is 8. As part of the SAP-2 software, the foregoing subroutine can be assembled and relocated at addresses F010H to F017H. Hereafter, the execution of a CALL F010H will produce a time delay of 1 ms.

EXAMPLE 11-17

How much time delay does this SAP-2 subroutine produce?

Label	Mnemonic	Comment
	MVI B,0AH	;Preset B counter with decimal 10
LOOP1:	MVI C,47H	;Preset C counter with decimal 71
LOOP2:	DCR C	;Count down on C
	JNZ LOOP2	;Test for C count of zero
	DCR B	;Count down on B
	JNZ LOOP1	;Test for B count of zero
	RET	

SOLUTION

This subroutine has two loops, one inside the other. The inner loop consists of DCR C and JNZ LOOP2. This inner loop produces a time delay of

DCR C:	$71 \times 4 \times 1\ \mu s = 284\ \mu s$	
JNZ LOOP2:	$70 \times 10 \times 1\ \mu s = 700$	(jump)
JNZ LOOP2:	$1 \times 7 \times 1\ \mu s = 7$	(no jump)
	$991\ \mu s$	

When the C count drops to zero, the program falls through the JNZ LOOP2. The B counter is decremented, and the JNZ LOOP1 sends the program back to the MVI C,47H. Then we enter LOOP2 for a second time. Because LOOP2 is inside LOOP1, LOOP2 will be executed 10 times and the overall time delay will be approximately 10 ms.

Here are the calculations for the overall subroutine:

MVI B,0AH:	$1 \times 7 \times 1\ \mu s = 7\ \mu s$	
MVI C,47H:	$10 \times 7 \times 1\ \mu s = 70$	
LOOP2:	$10 \times 991\ \mu s = 9{,}910$	
DCR B:	$10 \times 4 \times 1\ \mu s = 40$	
JNZ LOOP1:	$9 \times 10 \times 1\ \mu s = 90$	(jump)
JNZ LOOP1:	$1 \times 7 \times 1\ \mu s = 7$	(no jump)
RET:	$1 \times 10 \times 1\ \mu s = 10$	
	$10{,}134\ \mu s \approx 10$ ms	

This SAP-2 subroutine has a byte length of

$$2 + 2 + 1 + 3 + 1 + 3 + 1 = 13$$

It can be assembled and located at addresses F020H to F02CH. From now on, a CALL F020H will produce a time delay of approximately 10 ms.

By changing the first instruction to

```
MVI B,64H
```

the B counter is preset with decimal 100. In this case, the inner loop is executed 100 times and the overall time delay is approximately 100 ms. This 100-ms subroutine can be located at addresses F030H to F03CH.

EXAMPLE 11-18

Here is a subroutine with three loops *nested* one inside the other. How much time delay does it produce?

```
Label    Mnemonic      Comment
         MVI A,0AH     ;Preset A counter with
                         decimal 10
LOOP1:   MVI B,64H     ;Preset B counter with
                         decimal 100
LOOP2:   MVI C,47H     ;Preset C counter with
                         decimal 71
LOOP3:   DCR C         ;Count down C
         JNZ LOOP3     ;Test C for zero
         DCR B         ;Count down B
         JNZ LOOP2     ;Test B for zero
         DCR A         ;Count down A
         JNZ LOOP1     ;Test A for zero
         RET
```

SOLUTION

LOOP3 still takes approximately 1 ms to get through. LOOP2 makes 100 passes through LOOP3, so it takes about 100 ms to complete LOOP2. LOOP1 makes 10 passes through LOOP2; therefore, it takes around 1 s to complete the overall subroutine.

What do we have? A 1-s subroutine. It will be located in F040H to F052H. To produce a 1-s time delay, we would use a CALL F040H.

By changing the initial instruction to

```
MVI A,64H
```

LOOP1 will make 100 passes through LOOP2, which makes 100 passes through LOOP3. The resulting subroutine can be located at F060H to F072H and will produce a time delay of 10 s.

Table 11-4 summarizes the SAP-2 time delays. With these subroutines, we can produce delays from 1 ms to 10 s.

TABLE 11-4. SAP-2 SUBROUTINES

Label	Starting Address	Delay	Registers Used
D1MS	F010H	1 ms	C
D10MS	F020H	10 ms	B, C
D100MS	F030H	100 ms	B, C
D1SEC	F040H	1 s	A, B, C
D10SEC	F060H	10 s	A, B, C

EXAMPLE 11-19

The traffic lights on a main road show green for 50 s, yellow for 6 s, and red for 30 s. Bits 1, 2, and 3 of port 4 are the control inputs to peripheral equipment that runs these traffic lights. Write a program that produces time delays of 50, 6, and 30 s for the traffic lights.

SOLUTION

```
Label    Mnemonic      Comment
AGAIN:   MVI A,32H     ;Preset counter with
                         decimal 50
         STA SAVE      ;Save accumulator
                         contents
         MVI A,02H     ;Set bit 1
         OUT 04H       ;Turn on green light
LOOPGR:  CALL D1SEC    ;Call 1-s subroutine
         LDA SAVE      ;Load current A count
         DCR A         ;Decrement A count
         STA SAVE      ;Save reduced A count
         JNZ LOOPGR    ;Test for zero
         MVI A,06H     ;Preset counter with
                         decimal 6
         STA SAVE
         MVI A,04H     ;Set bit 2
         OUT 04H       ;Turn on yellow light
LOOPYE:  CALL D1SEC
         LDA SAVE
         DCR A
         STA SAVE
         JNZ LOOPYE
         MVI A,1EH     ;Preset counter with
                         decimal 30
         STA SAVE
         MVI A,08H     ;Set bit 3
         OUT 04H       ;Turn on red light
LOOPRE:  CALL D1SEC
         LDA SAVE
         DCR A
         STA SAVE
         JNZ LOOPRE
         JMP AGAIN
SAVE:    Data
```

Let's go through the green part of the program; the yellow and red are similar. The green starts with MVI A,32H, which loads decimal 50 into the accumulator. The STA SAVE will store this initial value in a memory location called SAVE. The MVI A,02H sets bit 1 in the accumulator; then the OUT 04H transfers this high bit to port 4. Since this port controls the traffic lights, the green light comes on.

The CALL D1SEC produces a time delay of 1 s. The LDA SAVE loads the accumulator with decimal 50. The DCR A decrements the count to decimal 49. The STA SAVE stores this decimal 49. Then the JNZ LOOPGR takes the program back to the CALL D1SEC for another 1-s delay.

The CALL D1SEC is executed 50 times; therefore, the green light is on for 50 s. Then the program falls through the JNZ LOOPGR to the MVI A,06H. The yellow part of the program then begins and results in the yellow light being on for 6 s. Finally, the red part of the program is executed and the red light is on for 30 s. The JMP AGAIN repeats the whole process. In this way, the program is controlling the timing of the green, yellow, and red lights.

EXAMPLE 11-20

Middle C on a piano has a frequency of 261.63 Hz. Bit 5 of port 4 is connected to an amplifier which drives a loudspeaker. Write a program that sends middle C to the loudspeaker.

SOLUTION

To begin with, the period of middle C is

$$T = \frac{1}{f} = \frac{1}{261.63\ \text{Hz}} = 3{,}822\ \mu\text{s}$$

What we are going to do is send to port 4 a signal like Fig. 11-9. This square wave is high for 1,911 μs and low for 1,911 μs. The overall period is 3,822 μs, and the frequency is 261.63 Hz. Because the signal is square rather than sinusoidal, it will sound distorted but it will be recognizable as middle C.

Here is a program that sends middle C to the loudspeaker.

Label	Mnemonic	Comment
LOOP1:	OUT 04H	;Send bit to speaker
	MVI C,86H	;Preset counter with decimal 134
LOOP2:	DCR C	;Count down
	JNZ LOOP2	;Test count
	CMA	;Reset bit 5
	NOP	;Fine tuning
	NOP	;Fine tuning
	JMP LOOP1	;Go back for next half cycle

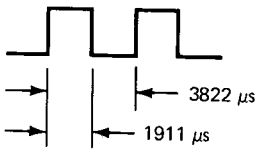


Fig. 11-9 Generating middle C note.

The OUT 04H sends a bit (either low or high) to the loudspeaker. The MVI presets the counter to decimal 134. Then comes LOOP2, the DCR and JNZ, which produces a time delay of 1,866 μs. The program then falls through to the CMA, which complements all bits in the accumulator. The two NOPs add a time delay of 8 μs. The JMP LOOP1 then takes the program back. When the OUT 04H is executed, bit 5 (complemented) goes to the loudspeaker. In this way the loudspeaker is driven into the opposite state. The execution time for both half cycles is 3,824 μs, close enough to middle C.

Here are the calculations for the time delay:

OUT 04H:	1 × 10 × 1 μs =	10 μs
MVI C,86H:	1 × 7 × 1 μs =	7
DCR C:	134 × 4 × 1 μs =	536
JNZ LOOP2:	133 × 10 × 1 μs =	1,330
JNZ LOOP2:	1 × 7 × 1 μs =	7
CMA:	1 × 4 × 1 μs =	4
2 NOPs:	2 × 4 × 1 μs =	8
JMP LOOP1:	1 × 10 × 1 μs =	10
		1,912 μs

This is the half-cycle time. The period is 3,824 μs.

EXAMPLE 11-21

Serial data is sometimes called a serial data stream because bits flow one after another. In Fig. 11-10 a serial data stream drives bit 7 of port 2 at a rate of approximately 600 bits per second. Write a program that inputs an 8-bit character in a serial data stream and stores it in memory location 2100H.

SOLUTION

Since approximately 600 bits are received each second, the period of each bit is

$$\frac{1}{600\ \text{Hz}} = 1{,}667\ \mu\text{s}$$

The idea will be to input a bit from port 2, rotate the accumulator right, wait approximately 1,600 μs, then input another bit, rotate the accumulator right, and so on, until all bits have been received.

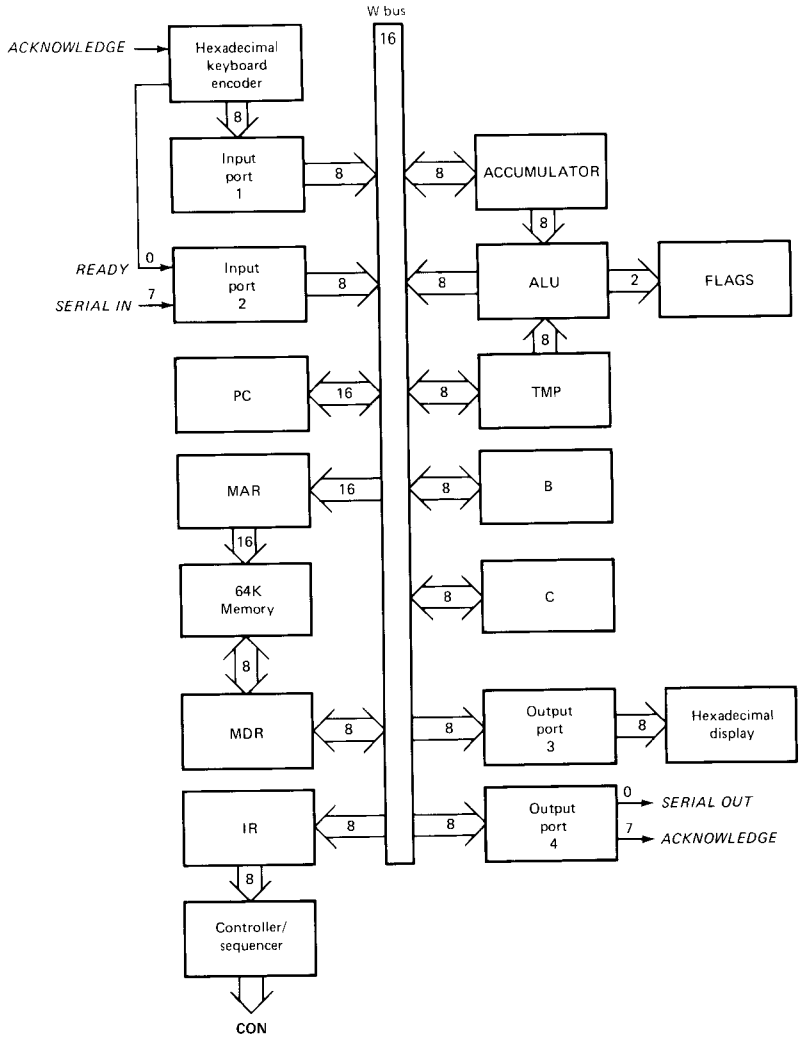


Fig. 11-10

Here is a program that will work:

Label	Mnemonic	Comment
	MVI B,00H	;Load zero into B register
	MVI C,07H	;Preset counter with decimal 7
BIT:	IN 02H	;Input data
	ANI 80H	;Isolate bit 7
	ORA B	;Update character
	RAR	;Move bits right
	MOV B,A	;Save bits in B
	MVI A,73H	;Begin a delay of 1,600 μs
DELAY:	DCR A	;Count down A
	JNZ DELAY	;Test A count for zero
	DCR C	;Count down C
	JNZ BIT	;Test C count for zero
	IN 02H	;Input last bit
	ANI 80H	;Isolate bit 7
	ORA B	
	STA 2100H	;Save character

The first instruction clears the B register. The second instruction loads decimal 7 into the C counter. The IN 02H brings in the data from port 2. The ANI mask isolates bit 7 because this is the *SERIAL IN* bit from port 2. The ORA B does nothing the first time through because B is full of 0s. The RAR moves the accumulator bits to the right. The MOV B,A stores the accumulator contents in the B register.

MVI A,73H presets the accumulator with decimal 115. Then comes a delay loop, DCR A and JNZ DELAY, that takes approximately 1,600 μs to complete.

The DCR C reduces the C count by 1, and the JNZ BIT tests the C count for zero. The program jumps back to the IN 02H to get the next bit from the serial data stream. The ANI mask isolates bit 7, which is then ORed with the contents of the B register; this combines the previous bit with the newly received bit. After another RAR, the two received bits are stored in the B register. Then comes another delay of approximately 1,600 μs.

The program continues to loop and each time a new bit is input from the serial data stream. After 7 bits have been

received, the program will fall through the JNZ BIT instruction.

The last four instructions do the following. The IN 02H brings in the eighth bit. The ANI isolates bit 7. The ORA B combines this new bit with the other seven bits in the B register. At this point, all received bits are in the accumulator. The STA 2100H then stores the byte in the accumulator at 2100H.

A concrete example will help. Suppose the 8 bits being received are 57H, the ASCII code for W. The LSB is received first, the MSB last. Here is how the contents of the B register appear after the execution of the ORA B:

A = 1000 0000	(First pass through loop)
A = 1100 0000	(Second pass)
A = 1110 0000	(Third pass)
A = 0111 0000	(Fourth pass)
A = 1011 1000	(Fifth pass)
A = 0101 1100	(Sixth pass)
A = 1010 1110	(Seventh pass)
A = 0101 0111	(Final contents)

Incidentally, the ASCII code only requires 7 bits; for this reason, the eighth bit (A_7) may be set to zero or used as a parity bit.

GLOSSARY

assembler A program that converts a source program into a machine-language program.

comment Personal notes in an assembly-language program that are not assembled. They refresh the programmer's memory at a later date.

conditional jump A jump that occurs only if certain conditions are satisfied.

direct addressing Addressing in which the instruction contains the address of the data to be operated on.

flag A flip-flop that keeps track of a changing condition during a computer run.

hand assembling Translating a source program into a machine-language program by hand rather than computer.

handshaking Interaction between a CPU and a peripheral device that takes place during an I/O operation. In SAP-2 it involves *READY* and *ACKNOWLEDGE* signals.

immediate addressing Addressing in which the data to be operated on is the byte immediately following the op code of the instruction.

implied addressing Addressing in which the location of the data is contained within the mnemonic.

label A name given to an instruction in an assembly-language program. To jump to this instruction, you can use the label rather than the address. The assembler will work out the correct address of the label and will use this address in the machine-language program.

mask A byte used with an ANI instruction to blank out certain bits.

register addressing Addressing in which the data is stored in a CPU register.

relocate To move a program or subroutine to another part of the memory. In doing this, the addresses of jump instructions must be converted to new addresses.

subroutine A program stored in higher memory that can be used repeatedly as part of a main program.

SELF-TESTING REVIEW

Read each of the following and provide the missing words. Answers appear at the beginning of the next question.

1. The controller-sequencer produces _____ words or microinstructions.
2. (*control*) A flag is a _____ that keeps track of a changing condition during a computer run. The sign flag is set when the accumulator contents go negative. The _____ flag is set when the accumulator contents go to zero.
3. (*flip-flop, zero*) In coding the LDA address and STA address instructions, the _____ byte of the address is stored in lower memory, the _____ byte in upper memory.
4. (*lower, upper*) The JMP instruction changes the program sequence by jumping to another part of the program. With the JM instruction, the jump is executed only if the sign flag is _____. With the JNZ instruction, the jump is executed only if the zero flag is _____.
5. (*set, clear*) Every subroutine must terminate with a _____ instruction. This returns the program to the instruction following the CALL. The CALL instruction is unconditional; it sends the computer to the starting address of a _____.
6. (*RET, subroutine*) An assembler allows you to write programs in mnemonic form. Then the assembler

converts these mnemonics into _____ language. The assembler ignores the _____ following a semicolon and assigns addresses to the labels. Labels can be up to six characters, the first of which must be a _____.

7. (*machine, comments, letter*) Repeated addition is one way to do _____. Programmed multiplication is used in most microprocessors because their ALUs can only add and subtract.
8. (*multiplication*) A parameter is a piece of data passed to a _____. When you call a subroutine, you often need to pass _____ for the subroutine to work properly.
9. (*subroutine, parameters*) A _____ is used to isolate a bit; it does this because the ANI sets all other bits to zero.
10. (*mask*) Handshaking is an interaction between a _____ and a peripheral device. In SAP-2 the _____ bit tells the CPU whether the input data is valid or not. After the data has been transferred into the computer, the CPU sends an _____ bit to the peripheral device.
11. (CPU, *READY, ACKNOWLEDGE*) The SAP-2 computer is microprogrammed with a _____ machine cycle. This means that some instructions take longer than others to execute.
12. (*variable*) The types of addressing covered up to now are direct, immediate, register, and implied.

PROBLEMS

11-1. Write a source program that loads the accumulator with decimal 100, the B register with decimal 150, and the C register with decimal 200.

11-2. Hand-assemble the source program of the preceding problem starting at address 2000H.

11-3. Write a source program that stores decimal 50 at memory location 4000H, decimal 51 at 4001H, and decimal 52 at 4002H.

11-4. Hand-assemble the source program in the preceding problem starting at address 2000H.

11-5. Write a source program that adds decimal 68 and 34, with the answer stored at memory location 5000H.

11-6. Hand-assemble the preceding program starting at address 2000H.

11-7. Here is a program:

Label	Mnemonic
LOOP:	MVI C,78H
	DCR C
	JNZ LOOP
	HLT

a. How many times (decimal) is the DCR C executed?
b. How many times does the program jump to LOOP?
c. How can you change the program to loop 210 times?

11-8. Which of the following are valid labels?
a. G100
b. UPDATE
c. 5TIMES
d. 678RED
e. T
f. REPEAT

11-9. Write a program that multiplies decimal 25 and 7 and stores the answer at 2000H. (Use the multiply subroutine located at F006H.)

11-10. Write a program that inputs a byte from port 1 and determines if the decimal equivalent is even or odd. If the byte is even, the program is to send an ASCII E to port 3; if odd, an ASCII O.

11-11. Modify the foregoing program so that it sends the answer in serial form to bit 0 of port 4.

11-12. Write a program that inputs a byte from port 1 using handshaking. Store the byte at address 4000H.

11-13. Hand assemble the foregoing program starting at address 2000H.

11-14. Write a subroutine that produces a time delay of approximately 500 μs.

11-15. Hand-assemble the preceding program starting at address 2000H.

11-16. Write a subroutine that produces a time delay of approximately 35 ms using a SAP-2 subroutine. Hand-assemble this subroutine and locate it at starting address E000H.

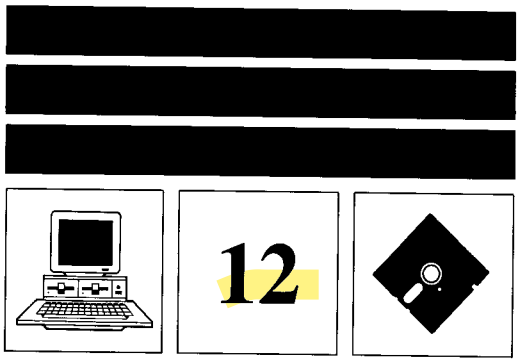
11-17. Write a subroutine that produces a time delay of 50 ms. (Use a SAP-2 subroutine.) Hand-assemble the program at starting address E100H.

11-18. Write a subroutine that produces a delay of 1 min. (Use CALL F060H.)

11-19. Hand-assemble the preceding subroutine at starting addresses F080H.

11-20. The C note one octave above middle C has a frequency of 523.25 Hz. Write a program that sends this note to bit 4 of port 4.

11-21. Hand-assemble the foregoing program starting at address 2000H.



12

SAP-3

The SAP-3 computer is an 8-bit microcomputer that is upward-compatible with the 8085 microprocessor. In this chapter, the emphasis is on the SAP-3 instruction set. This instruction set includes all the SAP-2 instructions of the preceding chapter plus new instructions to be discussed.

Appendix 6 shows the op codes, *T* states, flags, and so forth, for the SAP-3 instructions. In the remainder of this chapter, refer to Appendix 6 as needed.

12-1 PROGRAMMING MODEL

All you need to know about SAP-3 hardware is the programming model of Fig. 12-1. This is a diagram showing the CPU registers needed by a programmer.

Some of the CPU registers are familiar from SAP-2. For instance, the program counter (PC) is a 16-bit register that can count from 0000H to FFFFH or decimal 0 to 65,535. As you know, the program counter sends out the address of the next instruction to be fetched. This address is latched into the MAR.

CPU registers A, B, and C are the same as in SAP-2. These 8-bit registers are used in arithmetic and logic operations. Since the accumulator is only 8 bits wide, the range of unsigned numbers is 0 to 255; the range of signed 2's-complement numbers is −128 to +127.

SAP-3 has additional CPU registers (D, E, H, and L) for more efficient data processing. These 8-bit registers can be loaded with MOV and MVI instructions, the same as the A, B, and C registers. Also notice the F register, which stores flag bits S, Z, and others.

Finally, there is the *stack pointer* (SP), a 16-bit register. This new register controls a portion of memory known as the *stack*. The stack and the stack pointer are discussed later in this chapter.

Figure 12-1 shows all the CPU registers needed to understand the SAP-3 instruction set. With this programming model we can discuss the SAP-3 instruction set, which is upward-compatible with the 8080 and 8085. At the end of this chapter, you will know almost all of the 8080/8085 instruction set.

12-2 MOV AND MVI

The MOV and MVI instructions work the same as in SAP-2. The only difference is more registers to choose from. The format of any move instruction is

MOV reg1, reg2

where reg1 = A, B, C, D, E, H, or L
reg2 = A, B, C, D, E, H, or L

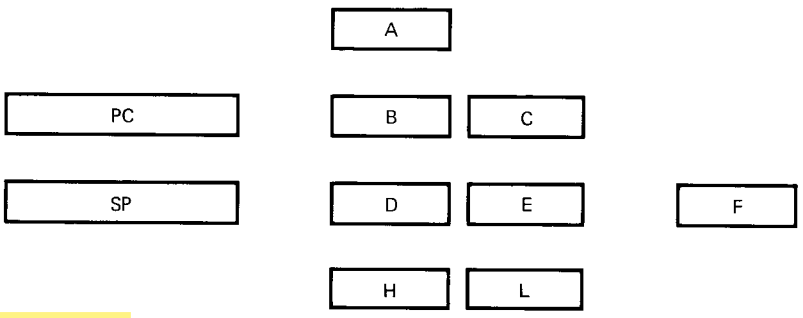


Fig. 12-1 SAP-3 programming model.

The MOV instructions send the data in reg2 to reg1. Symbolically,

$$\text{reg1} \leftarrow \text{reg2}$$

where the arrow indicates that the data in register 2 is copied nondestructively into register 1. At the end of the execution

$$\text{reg1} = \text{reg2}$$

For instance,

```
MOV L,A
```

copies A into L, so that

$$\mathbf{L = A}$$

Similarly,

```
MOV E,H
```

gives

$$\mathbf{E = H}$$

The immediate moves have the format of

```
MVI reg,byte
```

where reg = A, B, C, D, E, H, or L. Therefore, the execution of

```
MVI D,0EH
```

will result in

$$\mathbf{D} = 0\text{EH}$$

Likewise,

```
MVI L,FFH
```

produces

$$\mathbf{L} = \text{FFH}$$

What is the advantage of more CPU registers? As you may recall, MOV and MVI instructions use fewer *T* states than memory-reference instructions (MRIs). The extra CPU registers mean that we can use more MOV and MVI instructions and fewer MRIs. Because of this, SAP-3 programs can run faster than SAP-2 programs; furthermore, having more CPU registers for temporary storage simplifies program writing.

12-3 ARITHMETIC INSTRUCTIONS

Since the accumulator is only 8 bits wide, its contents can represent unsigned numbers from 0 to 255 or signed 2's complement numbers from −128 to +127. Whether signed or unsigned binary numbers are used, the programmer needs to detect *overflows*, sums or differences that lie outside the normal range of the accumulator. This is where the *carry* flag comes in.

Carry Flag

As shown in Fig. 6-7, a 4-bit adder-subtracter produces a sum $S_3S_2S_1S_0$ and a carry. In SAP-1, two 74LS83s (equivalent to eight full adders) produce an 8-bit sum and a carry. In this simple computer, the carry is disregarded. SAP-3, however, takes the carry into account.

Figure 12-2*a* shows the logic circuit used for the SAP-3 adder-subtracter. When *SUB* is low, the circuit adds the **A** and **B** inputs. If a final carry is generated, *CARRY* will be high and *CY* will be high. If there is no final carry, *CY* is low.

On the other hand, when *SUB* is high, the circuit forms the 2's complement of **B**, which is then added to **A.** Because of the final XOR gate, a high *CARRY* out of the last full-adder produces a low *CY*. If no carry occurs, *CY* is high.

In summary,

$$CY = \begin{cases} CARRY & \text{for ADD instructions} \\ \overline{CARRY} & \text{for SUB instructions} \end{cases}$$

During an add operation, *CY* is called a carry. During a subtract operation, *CY* is referred to as a *borrow*.

The 8-bit sum $S_7S_6S_5S_4S_3S_2S_1S_0$ is stored in the accumulator of Fig. 12-2*b*. The carry (or borrow) is stored in a special flip-flop called the *carry flag*, designated CY in Fig. 12-2*b*. This flag acts like the next higher bit of the accumulator. That is,

$$CY = A_8$$

Carry-Flag Instructions

There are two instructions we can use to control the carry flag. The STC instruction will set the CY flag if it is not already set. (STC stands for *set carry*.) So, if

$$CY = 0$$

the execution of a STC instruction produces

$$CY = 1$$

The other carry-flag instruction is the CMC, which stands for *complement the carry*. When executed, a CMC com-

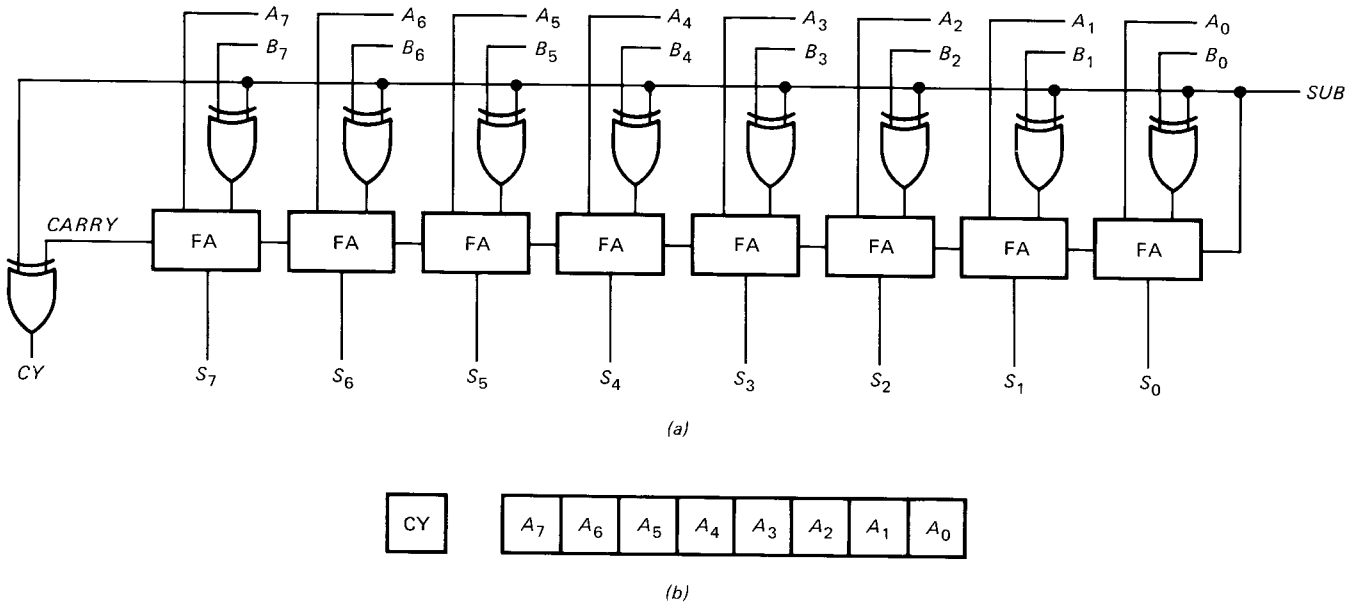


Fig. 12-2 (*a*) SAP-3 adder-subtractor (*b*) carry flag and accumulator.

plements the value of CY. If $CY = 1$, CMC produces a CY of 0. On the other hand, if $CY = 0$, CMC results in a CY of 1.

If you want to reset the carry flag and its current status is unknown, you have to set it, then complement it. That is, execution of

```
STC
CMC
```

guarantees that the final value of CY will be 0 if the initial value of CY is unknown.

ADD Instructions

The format of the ADD instruction is

```
ADD reg
```

where reg = A, B, C, D, E, H, or L. This instruction adds the contents of the specified register to the accumulator contents. The sum is stored in the accumulator and the carry flag is set or reset, depending on whether there is a final carry or not.

For instance, suppose

A = 1111 0001 and **E** = 0000 1000

The instruction

```
ADD E
```

produces the binary addition

```
  1111 0001
+ 0000 1000
-----------
  1111 1001
```

There is no final carry; therefore, at the end of the instruction cycle,

$CY = 0$ and **A** = 1111 1001

As another example, suppose

A = 1111 1111 and **L** = 0000 0001

Then executing an ADD L produces

```
  1111 1111
+ 0000 0001
-----------
 1 0000 0000
```

At the end of the instruction cycle

$CY = 1$ and **A** = 0000 0000

ADC Instructions

The ADC instruction (add with carry) is formatted like this:

```
ADC reg
```

where reg = A, B, C, D, E, H, or L. This instruction adds the contents of the specified register plus the carry flag to the contents of the accumulator. Because it includes the CY flag, the ADC instruction allows us to add numbers outside the unsigned 0 to 255 range or the signed −128 to +127 range.

As an example, suppose

A = 1000 0011
E = 0001 0010

and *CY* = 1

The execution of

ADC E

produces the following addition:

```
  1000 0011
  0001 0010
+        1
-----------
  1001 0110
```

Therefore, the new accumulator and carry flag contents are

CY = 0 **A** = 1001 0110

SUB Instructions

The SUB instruction is formatted as

SUB reg

where reg = A, B, C, D, E, H, or L. This instruction will subtract the contents of the specified register from the accumulator contents; the result is stored in the accumulator. If a final borrow occurs, the CY flag is set. If there is no borrow, the CY flag is reset. In other words, during subtraction the CY flag functions as a borrow flag.

For example, if

A = 0000 1111 and **C** = 0000 0001

then

SUB C

results in

```
  0000 1111
- 0000 0001
-----------
  0000 1110
```

Notice that there is no final borrow. In terms of 2's-complement addition, the foregoing subtraction appears like this:

```
  0000 1111
+ 1111 1111
-----------
1 0000 1110
```

The final *CARRY* is 1, but this is complemented during subtraction to get a *CY* of 0 (Fig. 12-2*a*). This is why the execution of SUB C produces

CY = 0 **A** = 0000 1110

Here is another example. If

A = 0000 1100 and **C** = 0001 0010

then a SUB C produces

```
  0000 1100
- 0001 0010
-----------
1 1111 1010
```

Notice the final borrow. This borrow occurs because the contents of the C register (decimal 18) are greater than the contents of the accumulator (decimal 12). In terms of 2's-complement arithmetic, the foregoing looks like

```
  0000 1100
+ 1110 1110
-----------
0 1111 1010
```

In this case, *CARRY* is 0 and *CY* is 1. The final register and flag contents are

CY = 1 and **A** = 1111 1010

SBB Instructions

SBB stands for *subtract with borrow*. This instruction goes one step further than the SUB. It subtracts the contents of a specified register and the CY flag from the accumulator contents. If

A = 1111 1111
E = 0000 0010

and *CY* = 1

the instruction SBB E starts by combining **E** and *CY* to get 0000 0011 and then subtracts this from the accumulator as follows:

```
  1111 1111
- 0000 0011
-----------
  1111 1100
```

The final contents are

$$CY = 0 \quad \text{and} \quad \mathbf{A} = 1111\ 1100$$

EXAMPLE 12-1

In unsigned binary, 8 bits can represent 0 to 255, whereas 16 bits can represent 0 to 65,535. Show a SAP-3 program that adds 700 and 900, with the final answer stored in the H and L registers.

SOLUTION

Double bytes can represent decimal 700 and 900 as follows:

$$700_{10} = 02BCH = 0000\ 0010\ 1011\ 1100_2$$
$$900_{10} = 0384H = 0000\ 0011\ 1000\ 0100_2$$

Here is how to add 700 and 900:

Label	Instruction	Comment
	MVI A,00H	;Clear the accumulator
	MVI B,02H	;Store upper byte (UB) of 700
	MVI C,BCH	;Store lower byte (LB) of 700
	MVI D,03H	;Store UB of 900
	MVI E,84H	;Store LB of 900
	ADD C	;Add LB of 700
	ADD E	;Add LB of 900
	MOV L,A	;Store partial sum
	MVI A,00H	;Clear the accumulator
	ADC B	;Add UB of 700 with carry
	ADD D	;Add UB of 900
	MOV H,A	;Store partial sum
	HLT	;Stop

The first five instructions initialize registers A through E. The ADD C and ADD E add the lower bytes BCH and 84H; this addition sets the carry flag because

$$\begin{array}{rr} \text{BCH} = & 1011\ 1100_2 \\ +\ 84\text{H} = & 1000\ 0100_2 \\ \hline 1\ 40\text{H} = & 1\ 0100\ 0000_2 \end{array}$$

The sum is stored in the L register and the final carry in the CY flag.

Next, the accumulator is cleared. The ADC B adds the upper byte plus the carry flag to get

$$\begin{array}{rr} 00\text{H} = & 0000\ 0000_2 \\ +\ 02\text{H} = & 0000\ 0010_2 \\ +\ 1\text{H} = & 1_2 \\ \hline 03\text{H} = & 0000\ 0011_2 \end{array}$$

Then the ADD D produces

$$\begin{array}{rr} 03\text{H} = & 0000\ 0011_2 \\ +\ 03\text{H} = & 0000\ 0011_2 \\ \hline 06\text{H} = & 0000\ 0110_2 \end{array}$$

The MOV H,A stores this upper sum in the H register.

So the program ends with the answer stored in the H and L registers as follows:

$$\mathbf{H} = 06\text{H} = 0000\ 0110_2$$
and
$$\mathbf{L} = 40\text{H} = 0100\ 0000_2$$

The complete answer is 0640H, which is equivalent to decimal 1,600.

12-4 INCREMENTS, DECREMENTS, AND ROTATES

This section is about increment, decrement, and rotate instructions. The increment and decrement are similar to those of SAP-2, but the rotates are different because of the carry flag.

Increment

The increment instruction appears as

INR reg

where reg = A, B, C, D, E, H, or L. It works as previously described. Therefore, given

$$\mathbf{L} = 0000\ 1111$$

the execution of INR L produces

$$\mathbf{L} = 0001\ 0000$$

The INR instruction has no effect on the carry flag, but, as before, it does affect the sign and zero flags. For instance, if

$$\mathbf{B} = 1111\ 1111$$

and the initial flags are

$$S = 1 \quad Z = 0 \quad CY = 0$$

then INR B produces

$$\mathbf{B} = 0000\ 0000$$
$$S = 0 \quad Z = 1 \quad CY = 0$$

As you see, the carry flag is unaffected even though the B register overflowed. At the same time, the zero flag has been set and the sign flag reset.

Decrement

The decrement is similar. It looks like

DCR reg

where reg = A, B, C, D, E, H, or L. If

E = 0111 0110

then a DCR E produces

E = 0111 0101

The DCR affects the sign and zero flags but not the carry flag. This is why the initial values may be

E = 0000 0000
$S = 0$ $Z = 1$ $CY = 0$

Executing a DCR E results in

E = 1111 1111
$S = 1$ $Z = 0$ $CY = 0$

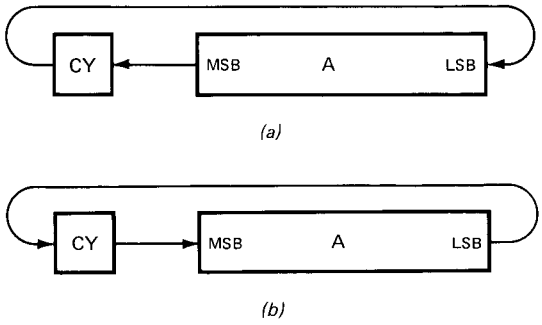


Fig. 12-3 (*a*) RAL; (*b*) RAR.

Rotate All Left

Figure 12-3*a* illustrates the RAL instruction used in SAP-3. The CY flag is included in the rotation of bits. RAL stands for rotate all left, which is a reminder that all bits including the CY flag are rotated to the left.

If the initial values are

$CY = 1$ **A** = 0111 0100

then executing a RAL instruction produces

$CY = 0$ **A** = 1110 1001

As you see, the original CY goes to the LSB position, and the original MSB goes to the CY flag.

Rotate All Right

The rotate-all-right instruction (RAR) rotates all bits including the CY flag to the right, as shown in Fig. 12-3*b*. If

$CY = 1$ **A** = 0111 0100

an RAR will result in

$CY = 0$ **A** = 1011 1010

This time, the original *CY* goes to the MSB position, and the original LSB goes into the CY flag.

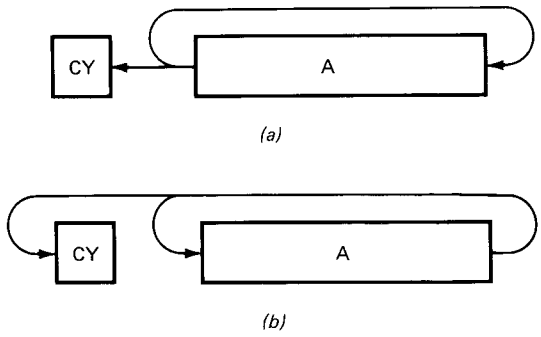


Fig. 12-4 (*a*) RLC; (*b*) RRC.

Rotate Left with Carry

Sometimes you don't want to treat the CY flag as an extension of the accumulator. In other words, you may not want to rotate all bits. Figure 12-4*a* illustrates the RLC instruction. The accumulator bits are rotated left, and the MSB is saved in the CY flag. For instance, given

$CY = 1$ **A** = 0111 0100

executing an RLC produces

$CY = 0$ **A** = 1110 1000

Rotate Right with Carry

Figure 12-4*b* shows how the RRC instruction rotates the bits. In this case, the accumulator bits are rotated right and the LSB is saved in the CY flag. So, given

$CY = 1$ **A** = 0111 0100

an RRC will result in

$CY = 0$ **A** = 0011 1010

Multiply and Divide by 2

Example 11-14 showed a program where the RAR instruction was used in converting from parallel to serial data. Parallel-to-serial conversion, and vice versa, is one of the main uses of rotate instructions.

There is another use for rotate instructions. Rotating has the effect of multiplying or dividing the accumulator contents by a factor of 2. Specifically, with the carry flag reset, an RAL has the effect of multiplying by 2, while the RAR divides by 2. This can be proved algebraically, but it's much easier to examine a few specific examples to see how it works.

Suppose

$$CY = 0 \qquad \mathbf{A} = 0000\ 0111$$

Then an RAL produces

$$CY = 0 \qquad \mathbf{A} = 0000\ 1110$$

The accumulator contents have changed from decimal 7 to decimal 14. The RAL has multiplied by 2.

Likewise, if

$$CY = 0 \qquad \mathbf{A} = 0010\ 0001$$

then an RAL results in

$$CY = 0 \qquad \mathbf{A} = 0100\ 0010$$

In this case, **A** has changed from decimal 33 to 66.

RAR instructions have the opposite effect; they divide by 2. If

$$CY = 0 \qquad \mathbf{A} = 0001\ 1000$$

an RAR gives

$$CY = 0 \qquad \mathbf{A} = 0000\ 1100$$

The decimal contents of the accumulator have changed from decimal 24 to 12.

Remember the basic idea. RAL instructions have the effect of multiplying by 2; RAR instructions divide by 2.

12-5 LOGIC INSTRUCTIONS

The SAP-3 logic instructions are almost the same as in SAP-2. For instance, three of the logic instructions are

ANA reg
ORA reg
XRA reg

where reg = A, B, C, D, E, H, or L. These instructions will AND, OR, or XOR the contents of the specified register with the contents of the accumulator on a bit-by-bit basis.

The only new logic instruction is the CMP, formatted as

CMP reg

where reg = A, B, C, D, E, H, or L. CMP compares the contents of the specified register with the contents of the accumulator. The zero flag indicates the outcome of this comparison as follows:

$$Z = \begin{cases} 1 & \text{if A = reg} \\ 0 & \text{if A} \neq \text{reg} \end{cases}$$

SAP-3 carries out a CMP as follows. The contents of the accumulator are copied in a temporary register. Then the contents of the specified register are subtracted from the contents of the temporary register. Since the ALU does the subtraction, the zero flag is affected. If the 2 bytes being compared are equal, the zero flag is set. If the bytes are unequal, the zero flag is reset. Because the temporary register is used, the accumulator contents are not changed by a CMP instruction.

For example, if

$$\mathbf{A} = \text{F8H}$$
$$\mathbf{D} = \text{F8H}$$
and $$Z = 0$$

executing a CMP D results in

$$\mathbf{A} = \text{F8H}$$
$$\mathbf{D} = \text{F8H}$$
and $$Z = 1$$

CMP has no effect on **A** and **D**; only the flag changes to indicate that **A** and **D** are equal. (If they were not equal, Z would be 0.)

CMP is a powerful instruction because it allows us to compare the accumulator contents with the data in a specified register. By following a CMP with a conditional zero jump, we can control loops in a new way. Later programs will show how this is done.

12-6 ARITHMETIC AND LOGIC IMMEDIATES

So far, we have introduced these arithmetic and logic instructions: ADD, ADC, SUB, SBB, ANA, ORA, XRA, and CMP. Each of these has the accumulator as an implied register; the data comes from a specified register (A, B, C, D, E, H, or L).

The immediate instructions from SAP-2 that carry over to SAP-3 are ANI, ORI, and XRI. As you know, each of these has the format of

ANI byte
ORI byte
XRI byte

where the immediate byte is ANDed, ORed, or XORed with the accumulator byte.

Besides the foregoing, SAP-3 has these immediate instructions:

ADI byte
ACI byte
SUI byte
SBI byte
CPI byte

The ADI adds the immediate byte to the accumulator byte. The ACI adds the immediate byte plus the CY flag to the accumulator byte. The SUI subtracts the immediate byte from the accumulator byte. The SBI subtracts immediate byte and the CY flag from the accumulator byte. The CPI compares the immediate byte with the accumulator byte; if the bytes are equal, the zero flag is set; if not, it is reset.

EXAMPLE 12-2

Show a program that subtracts 700 from 900 and stores the answer in the H and L registers.

SOLUTION

We need double bytes to represent 900 and 700 as follows:

900_{10} = 0384H = $0000\,0011\,1000\,0100_2$
700_{10} = 02BCH = $0000\,0010\,1011\,1100_2$

Here's the program for subtracting 700 from 900:

Label	Instruction	Comment
	MVI A, 84H	;Load LB of 900
	SUI BCH	;Subtract LB of 700
	MOV L,A	;Save lower half answer
	MVI A, 03H	;Load UB of 900
	SBI 02H	;Subtract UB of 700 with borrow
	MOV H,A	;Save upper half answer

The first two instructions subtract the lower bytes as follows:

$$\begin{array}{r} 1000\,0100 \\ -\ 1011\,1100 \\ \hline 1\,1100\,1000 \end{array}$$

At this point,

$$CY = 1 \qquad \mathbf{A} = \text{C8H}$$

The high *CY* flag indicates a borrow.

After saving C8H in the L register, the program loads the upper byte of 900 into the accumulator. The SBI is used instead of a SUI because of the borrow that occurred when subtracting the bytes. The execution of the SBI gives

$$\begin{array}{r} 0000\,0011 \\ -\ 0000\,0010 \\ -\ \quad\quad 1 \\ \hline 0000\,0000 \end{array}$$

This part of the answer is stored in the H register, so that the final contents are

$$\mathbf{H} = 00\text{H} = 0000\,0000_2$$
$$\mathbf{L} = \text{C8H} = 1100\,1000_2$$

12-7 JUMP INSTRUCTIONS

Here are the SAP-2 jump instructions that become part of the SAP-3 instruction set:

JMP address	(Unconditional jump)
JM address	(Jump if minus)
JZ address	(Jump if zero)
JNZ address	(Jump if not zero)

Here are some more SAP-3 jump instructions.

JP

JM stands for *jump if minus*. When the program encounters a JM address, it will jump to the specified address if the sign flag is set.

The JP instruction has the opposite effect. JP stands for *jump if positive* (including zero). This means that

JP address

produces a jump to the specified address if the sign flag is reset.

JC and JNC

The instruction

JC address

means to jump to the specified address if the carry flag is set. In short, JC stands for jump if carry. Similarly,

JNC address

means to jump to the specified address if the carry flag is not set. That is, jump if no carry.

Here is a program segment to illustrate JC and JNC:

```
Label      Instruction      Comment
           MVI A,FEH
REPEAT:    ADI 01H
           JNC REPEAT
           MVI A,C4H
           JC ESCAPE
             .
             .
             .
ESCAPE:    MOV L,A
```

The MVI loads the accumulator with FEH. The ADI adds 1 to get FFH. Since no carry takes place, the JNC takes the program back to the REPEAT point, where a second ADI is executed. This time the accumulator overflows to get contents of 00H with a carry. Since the CY flag is set, the program falls through the JNC. The accumulator is loaded with C4H. Then the JC produces a jump to the ESCAPE point, where the C4H is loaded into the L register.

JPE and JPO

Besides the sign, zero, and carry flag, SAP-3 has a *parity flag* designated P. During the execution of certain instructions (like ADD, INR, etc.), the ALU result is checked for parity. If the result has an even number of 1s, the parity flag is set; if an odd number of 1s, the flag is reset.

The instruction

JPE address

produces a jump to the specified address when the parity flag is set (even parity). On the other hand,

JPO address

results in a jump when the parity flag is reset (odd parity). For instance, given these flags,

$$S = 1 \quad Z = 0 \quad CY = 0 \quad P = 1$$

the program would jump if it encountered a JPE instruction; but it would fall through a JPO instruction.

Incidentally, we now have discussed all the flags in the SAP-3 computer. For upward compatibility with the 8085 microprocessor, these flags are stored in the F register, as shown in Fig. 12-5. For instance, if the contents of the F register are

S	Z	0	0	0	P	0	CY

Fig. 12-5 F register stores flags.

$$\mathbf{F} = 0100\ 0101$$

then we know that the flags are

$$S = 0 \quad Z = 1 \quad P = 1 \quad CY = 1$$

EXAMPLE 12-3

What does the following program segment do?

SOLUTION

```
Label      Instruction      Comment
           MVI E,00H        ;Initialize counter
LOOP:      INR E            ;Increment counter
           MOV A,E          ;Load A with E
           CPI FFH          ;Compare to 255
           JNZ LOOP         ;Go back if not 255
```

The E register is being used as a counter. It starts at 0. The first time the INR and MOV are executed

$$\mathbf{A} = 01H$$

After executing the CPI, the zero flag is 0 because 01H and FFH are unequal. The JNZ then forces the program to return to the LOOP point.

The looping will continue until the INR and MOV have been executed 255 times to get

$$\mathbf{A} = FFH$$

On this pass through the loop, the CPI sets the zero flag because the accumulator byte and the immediate byte are equal. With the zero flag set for the first time, the program falls through the JNZ instruction.

Do you see the point? The computer will loop 255 times before it falls through the JNZ. One use of this program segment is to set up a time delay. Another use is to insert additional instructions inside the loop as follows:

```
Label     Instruction     Comment
          MVI E,00H
LOOP:       .
            .
            .
          INR E
          MOV A,E
          CPI FFH
          JNZ LOOP
```

The instructions at the beginning of the loop (symbolized by dots) will be executed 255 times. If you want to change the number of passes through the loop, modify the CPI instruction as required.

12-8 EXTENDED-REGISTER INSTRUCTIONS

Some SAP-3 instructions use pairs of CPU registers to process 16-bit data. In other words, during the execution of certain instructions, the CPU registers are cascaded, as shown in Fig. 12-6. The pairing is always as shown: B with C, D with E, and H with L. What follows are the SAP-3 instructions that use *register pairs*. Throughout these instructions, you will notice the letter X, which stands for extended register, a reminder that register pairs are involved.

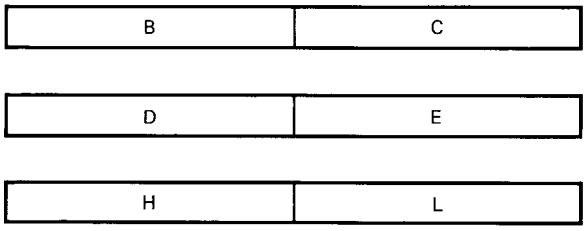


Fig. 12-6 Register pairs.

Load Extended Immediate

Since there are three register pairs (BC, DE, and HL), the LXI instruction can appear in any of these forms:

```
LXI B,dble
LXI D,dble
LXI H,dble
```

where B stands for BC
D stands for DE
H stands for HL
dble stands for double byte

The LXI instruction says to load the specified register pair with the double byte. For instance, if we execute

```
LXI B,90FFH
```

the B and C registers are loaded with the upper and lower bytes to get

B = 90H
C = FFH

Visualizing **B** and **C** paired off as shown in Fig. 12-6, we can write

BC = 90FFH

DAD Instructions

DAD stands for double-add. This instruction has three forms:

```
DAD B
DAD D
DAD H
```

where B stands for BC
D stands for DE
H stands for HL

The DAD instruction adds the contents of the specified register pair to the contents of the HL register pair; the result is then stored in the HL register pair. For instance, given

BC = F521H
HL = 0003H

the execution of a DAD B produces

HL = F524H

As you see, F521H and 0003H are added to get F524H. The result is stored in the HL register pair.

The DAD instruction affects the CY flag. If there is a carry out of the HL register pair, the CY flag is set; otherwise it is reset. As an example, if

DE = 0001H
HL = FFFFH

a DAD D will result in

HL = 0000H
CY = 1

Incidentally, a DAD H has the effect of adding the data in the HL register pair to itself. In other words, a DAD H doubles the value of HL. If

HL = 1234H

a DAD H results in

$$\mathbf{HL} = 2468\text{H}$$

INX and DCX

INX stands for *increment the extended register*, and DCX means *decrement the extended register*. The extended increment instructions are

```
INX B
INX D
INX H
```

where B stands for BC
D stands for DE
H stands for HL

The DCX instructions have a similar format: DCX B, DCX D, and DCX H.

The INX and DCX instructions have no effect on the flags. For instance, if

$$\mathbf{BC} = \text{FFFFH}$$
$$S = 1$$
$$Z = 0$$
$$P = 1$$
$$CY = 0$$

executing an INX B results in

$$\mathbf{BC} = 0000\text{H}$$
$$S = 1$$
$$Z = 0$$
$$P = 1$$
$$CY = 0$$

Notice that all flags are unaffected.

In summary, the extended register instructions are LXI, DAD, INX, and DCX. Of the three register pairs, the HL combination is special. The next section tells you why.

12-9 INDIRECT INSTRUCTIONS

As discussed in Chap. 10, the program counter is an *instruction pointer;* it points to the memory location where the next instruction is stored.

The HL register pair is different; it points to memory locations where data is stored. In other words, SAP-3 has several instructions where the HL register pair acts like a *data pointer*. The following discussion clarifies the idea.

Visualizing the HL Pointer

Figure 12-7*a* shows a 64K memory; it has 65,636 memory registers or memory locations where data is stored. The first memory location is M_{0000H}, the next is M_{0001H}, and so on. The memory location with address HL is M_{HL}.

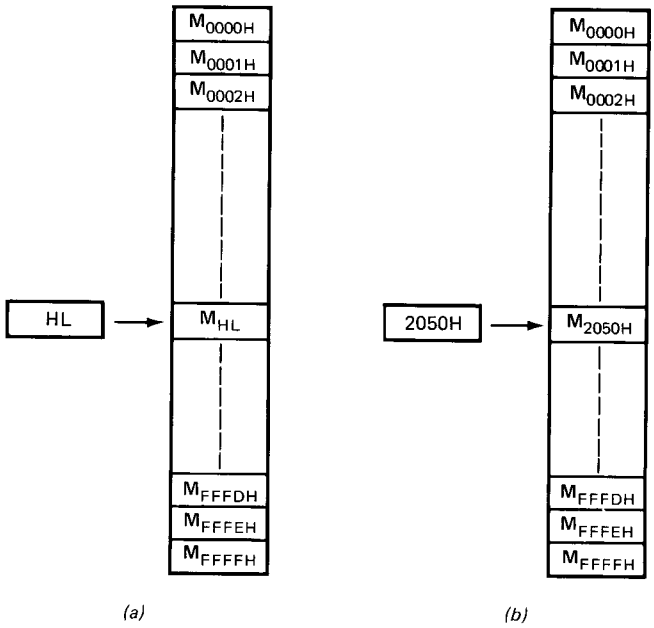


Fig. 12-7 (*a*) HL pointer; (*b*) pointing to 2050H.

With some SAP-3 instructions, the contents of the HL register pair are used as the address for data in memory. That is, the contents of the HL register pair are sent to the MAR, and then a memory read or write is performed. It's as though the HL register pair were pointing to the desired memory location, as shown in Fig. 12-7*a*.

For instance, suppose

$$\mathbf{HL} = 2050\text{H}$$

If HL is acting as a pointer, its contents (2050H) are sent to the MAR during one *T* state. During the next *T* state, the memory location whose address is 2050H undergoes a read or write operation. As shown in Fig. 12-7*b* the HL register pair points to the desired memory location.

Indirect Addressing

With direct addressing like LDA 5000H and STA 6000H, the programmer knows the address of the memory location because the instruction itself directly gives the address. With instructions that use the HL pointer, however, programmers do not know the address; all they know is that the address is stored in the HL register pair. Whenever an instruction uses the HL pointer, the addressing is called *indirect addressing*.

Indirect Read

One of the indirect instructions is

```
MOV reg,M
```

where reg = A, B, C, D, E, H, or L
M = M_{HL}

This instruction says to load the specified register with the data addressed by HL. After execution of this instruction, the designated register contains M_{HL}.

For instance, if

HL = 3000H and $\mathbf{M}_{3000H}$ = 87H

executing a

MOV C,M

produces

C = 87H

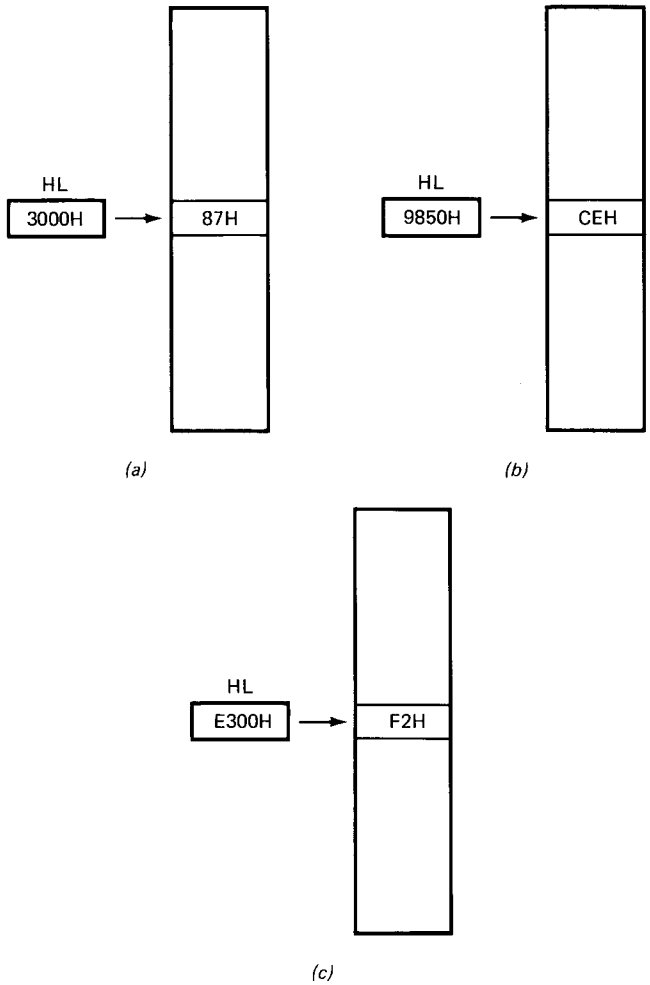


Fig. 12-8 Examples of indirect addressing.

Figure 12-8*a* shows how to visualize the MOV C,M. The HL pointer points to 87H, which is the data to be read into register C.

As another example, if

HL = 9850H and $\mathbf{M}_{9850H}$ = CEH

a MOV A,M results in

A = CEH

Figure 12-8*b* illustrates the MOV A,M. The HL pointer points to CEH, which is the data to be loaded into the A register.

Indirect Write

Here is another indirect MOV instruction:

MOV M,reg

where M = M_{HL}
reg = A, B, C, D, E, H, or L

This says to load the memory location addressed by HL with the contents of the specified register. After execution of this instruction,

$\mathbf{M}_{HL}$ = reg

As an example, if

HL = E300H
B = F2H

the execution of a MOV M,B produces

$\mathbf{M}_{E300H}$ = F2H

Figure 12-8*c* illustrates the idea.

Indirect-Immediate Instructions

Sometimes we want to write immediate data into the memory location addressed by the HL pointer. The instruction to use in this case is

MVI M,byte

Here is an example. If **HL** = 3000H, executing a

MVI M,87H

produces

$\mathbf{M}_{3000H}$ = 87H

Other Pointer Instructions

Here are more instructions using the HL pointer:

```
ADD M
ADC M
SUB M
SBB M
INR M
DCR M
ANA M
ORA M
XRA M
CMP M
```

In each of these, M is the memory location addressed by HL. Think of M as another register where data is stored. Each of the foregoing instructions operates on this data as previously described.

EXAMPLE 12-4

Suppose 256 bytes of data are stored in memory between addresses 2000H and 20FFH. Show a program that will copy these 256 bytes at addresses 3000H to 30FFH.

SOLUTION

Label	Instruction	Comment
	LXI H,1FFFH	;Initialize pointer
LOOP:	INX H	;Advance pointer
	MOV B,M	;Read byte
	MOV A,H	;Load 20H into accumulator
	ADI 10H	;Add offset to get 30H
	MOV H,A	;Offset pointer
	MOV M,B	;Write byte in new location
	SUI 10H	;Subtract offset
	MOV H,A	;Restore H for next read
	MOV A,L	;Prepare for compare
	CPI FFH	;Check for 255
	JNZ LOOP	;If not done, get next byte
	HLT	;Stop

This looping program transfers each successive byte in the 2000H–20FFH area of memory into the 3000H–30FFH area of memory. Here are the details.

The LXI initializes the pointer with address 1FFFH. The first time into the loop, the INX will advance the HL pointer to 2000H. The MOV B,M then reads the first byte into the B register. The next three instructions

```
MOV A,H
ADI 10H
MOV H,A
```

offset the HL pointer to 3000H. Then the MOV M,B writes the first byte into location 3000H. The next two instructions, SUI and MOV, restore the HL pointer to 2000H. The MOV A,L puts 00H into the accumulator. Because the CPI FFH resets the zero flag, the JNZ forces the program to return to the LOOP entry point.

On the second pass through the loop, the computer will read the byte at 2001H and it will store this byte at 3001H. The looping will continue with successive bytes being moved from the 2000H–20FFH section of memory to the 3000H–30FFH area. Since the first byte is read from 2000H, the 256th byte is read from 20FFH. After this byte is stored at 30FFH, the pointer is restored to 20FFH. The MOV A,L then loads the accumulator to get

$$A = FFH$$

This time, the CPI FFH will set the zero flag. Therefore, the program will fall through the JNZ to the HLT.

12-10 STACK INSTRUCTIONS

SAP-2 has a CALL instruction that sends the program to a subroutine. As you recall, before the jump takes place, the program counter is incremented and the address is saved at addresses FFFEH and FFFFH. The addresses FFFEH and FFFFH are set aside for the purpose of saving the return address. At the completion of a subroutine, the RET instruction loads the program counter with the return address, which allows the computer to get back to the main program.

The Stack

A *stack* is a portion of memory set aside primarily for saving return addresses. SAP-2 has a stack because addresses FFFEH and FFFFH are used exclusively for saving the return address of a subroutine call. Figure 12-9*a* shows how to visualize the SAP-2 stack.

SAP-3 is different. To begin with, the programmer decides where to locate the stack and how large to make it. As an example, Fig. 12-9*b* shows a stack between addresses 20E0H and 20FFH. This stack contains 32 memory locations for saving return addresses. Programmers can locate the stack anywhere they want in memory, but once they have set up the stack, they no longer use that portion of memory for program and data. Instead, the stack becomes a special space in memory, used for storing the return addresses of subroutine calls.

Stack Pointer

The instructions that read and write into the stack are called *stack instructions;* these include PUSH, POP, CALL, and

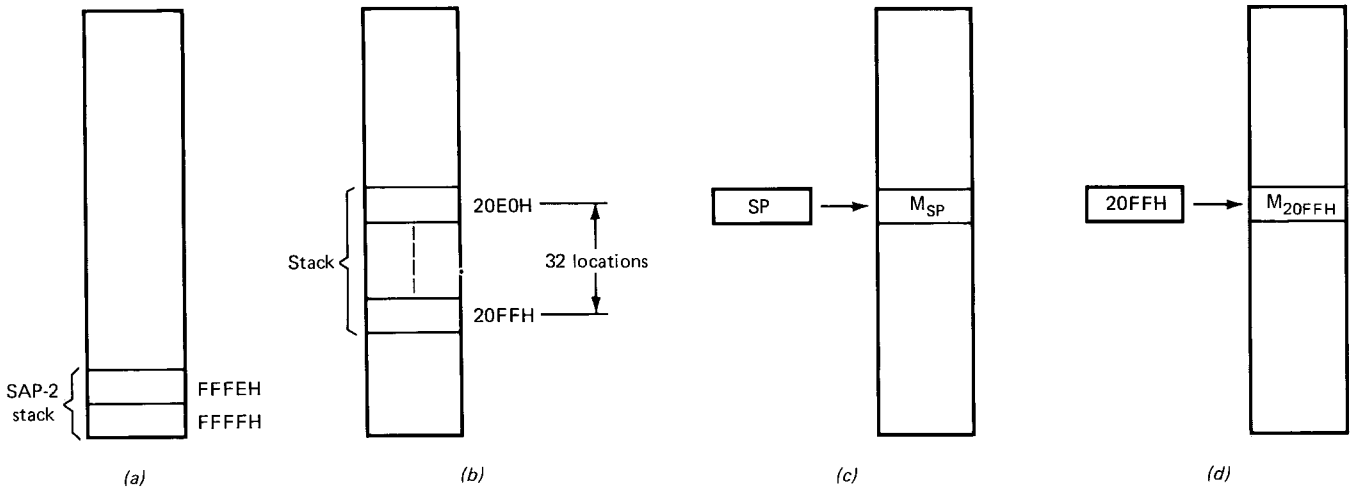


Fig. 12-9 (*a*) SAP-2 stack; (*b*) example of a stack; (*c*) stack pointer addresses the stack; (*d*) SP points to 20FFH.

others to be discussed. Stack instructions use indirect addressing because a 16-bit register called the *stack pointer* (SP) holds the address of the desired memory location. As shown in Fig. 12-9*c*, the stack pointer is similar to the HL pointer because the contents of the stack pointer indicate which memory location is to be accessed. For instance, if

SP = 20FFH

the stack pointer points to memory location M_{20FFH} (see Fig. 12-9*d*). Depending on the stack instruction, a byte is then read from, or written into, this memory location.

To initialize the stack pointer, we can use the immediate load instruction

LXI SP,dble

For instance, if we execute

LXI SP,20FFH

the stack pointer is loaded with 20FFH.

PUSH Instructions

The contents of the accumulator and the flag register are known as the *program status word* (PSW). The format for this word is

PSW = AF

where **A** = contents of accumulator
F = contents of flag register

The accumulator contents are the high byte, and the flag contents the low byte. When calling subroutines, we usually have to save the program status word, so that the main program can resume after the subroutine is executed. We may also have to save the contents of the other registers.

PUSH instructions allow us to save data in a stack. Here are the four PUSH instructions:

PUSH B
PUSH D
PUSH H
PUSH PSW

where B stands for BC
D stands for DE
H stands for HL
PSW stands for program status word

When a PUSH instruction is executed, the following things happen:

1. The stack pointer is decremented to get a new value of SP − 1.
2. The high byte in the specified register pair is stored in M_{SP-1}.
3. The stack pointer is decremented again to get SP − 2.
4. The low byte in the specified register pair is stored in M_{SP-2}.

Here is an example. Suppose

BC = 5612H
SP = 2100H

When a PUSH B is executed,

1. The stack pointer is decremented to get 20FFH.
2. The high byte 56H is stored at 20FFH (Fig. 12-10*a*).
3. The stack pointer is again decremented to get 20FEH.
4. The low byte 12H is stored at 20FEH (Fig. 12-10*b*).

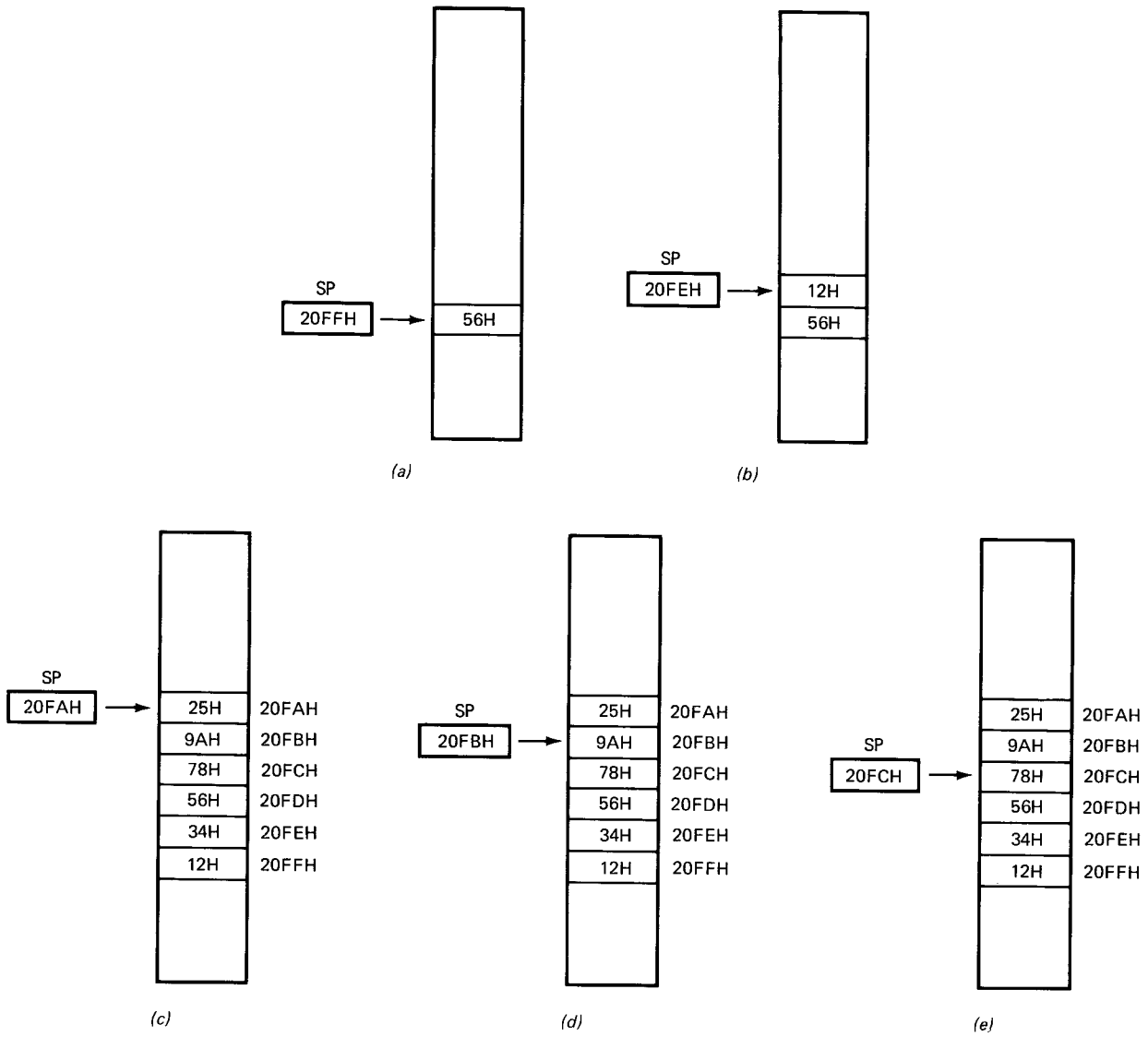


Fig. 12-10 Push operations: (*a*) high byte first; (*b*) low byte second; (*c*) 6 bytes pushed on stack; (*d*) popping a byte off the stack; (*e*) incrementing stack pointer.

Here's another example. Suppose

SP = 2100H
AF = 1234H
DE = 5678H
HL = 9A25H

then executing

```
PUSH PSW
PUSH D
PUSH H
```

loads the stack as shown in Fig. 12-10*c*. The first PUSH stores 12H at 20FFH and 34H at 20FEH. The next PUSH stores 56H at 20FDH and 78H at 20FCH. The last PUSH stores 9AH at 20FBH and 25H at 20FAH. Notice how the stack builds. Each new PUSH shoves data onto the stack.

POP Instructions

Here are four POP instructions:

```
POP B
POP D
POP H
POP PSW
```

where B stands for BC
D stands for DE
H stands for HL
PSW stands for program status word

When a POP is executed, the following happens:

1. The low byte is read from the memory location addressed by the stack pointer. This byte is stored in the lower half of the specified register pair.
2. The stack pointer is incremented.
3. The high byte is read and stored in the upper half of the specified register pair.
4. The stack pointer is incremented.

Here's an example. Suppose the stack is loaded as shown in Fig. 12-10*c* with the stack pointer at 20FAH. Then execution of POP B does the following:

1. Byte 25H is read from 20FAH (Fig. 12-10*c*) and stored in the C register.
2. The stack pointer is incremented to get 20FBH. Byte 9AH is read from 20FBH (Fig. 12-10*d*) and stored in the B register. The BC register pair now contains

$$\mathbf{BC} = 9A25H$$

3. The stack pointer is incremented to get 20FCH (Fig. 12-10*e*).

Each time we execute a POP, 2 bytes come off the stack. If we were to execute a POP PSW and a POP H in Fig. 12-10*e*, the final register contents would be

$$\mathbf{AF} = 5678H$$
$$\mathbf{HL} = 1234H$$

and the stack pointer would contain

$$\mathbf{SP} = 2100H$$

CALL and RET

The main purpose of the SAP-3 stack is to save return addresses automatically when using CALLs. When a

CALL address

is executed, the contents of the program counter are pushed onto the stack. Then the starting address of the subroutine is loaded into the program counter. In this way, the next instruction fetched is the first instruction of the subroutine. On completion of the subroutine, a RET instruction pops the return address off the stack into the program counter.

Here is an example:

Address	Instruction
2000H	LXI SP,2100H
2001H	
2002H	
2003H	CALL 8050H
2004H	
2005H	
2006H	MVI A,0EH
.	.
.	.
.	.
20FFH	HLT
.	
.	
.	
8050H	.
.	.
.	.
8059H	RET

To begin with, LXI and CALL instructions take 3 bytes each when assembled: 1 byte for the op code and 2 for the data. This is why the LXI instruction occupies 2000H to 2002H and the CALL occupies 2003H to 2005H.

The LXI loads the stack pointer with 2100H. During the execution of CALL 8050H, the address of the next instruction is saved in the stack. This address (2006H) is pushed onto the stack in the usual way; the stack pointer is decremented and the high byte 20H is stored; the stack pointer is decremented again, and the low byte 06H is stored (see Fig. 12-11*a*). The program counter is then loaded with 8050H, the starting address of the subroutine.

When the subroutine is completed, the RET instruction takes the computer back to the main program as follows. First, the low byte is popped from the stack into the lower half of the program counter; then the high byte is popped from the stack into the upper half of the program counter.

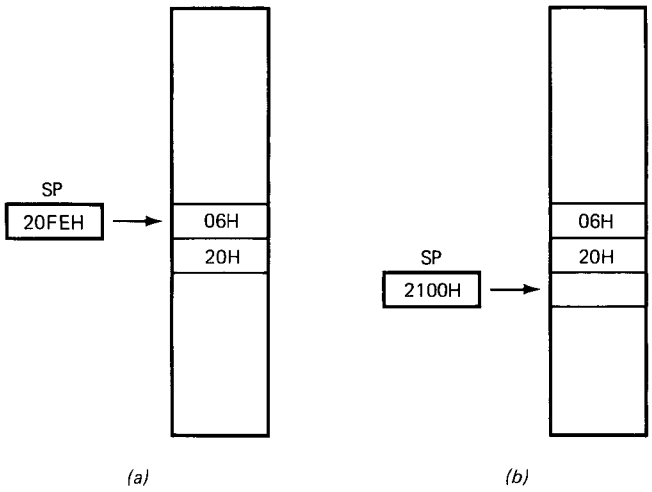


Fig. 12-11 (*a*) Saving a return address during a subroutine call; (*b*) popping the return address during a RET.

After the second increment, the stack pointer is back at 2100H, as shown in Fig. 12-11*b*.

The stack operation is automatic during CALL and RET instructions. All we have to do is initialize the setting of the stack pointer; this is purpose of the LXI SP,dble instruction. It sets the upper boundary of the stack. Then a CALL automatically pushes the return address onto the stack, and a RET automatically pops this return address off the stack.

Conditional Calls and Returns

Here is a list of the SAP-3 conditional calls:

CNZ address
CZ address
CNC address
CC address
CPO address
CPE address
CP address
CM address

They are similar to the conditional jumps discussed earlier. The CNZ branches to a subroutine only if the zero flag is reset, the CZ branches only if the zero flag is set, the CNC branches only if the carry flag is reset, and so forth.

The return from a subroutine may also be conditional. Here is a list of the conditional returns:

RNZ
RZ
RNC
RC
RPO
RPE
RP
RM

The RNZ will return only if the zero flag is reset, the RZ returns only when the zero flag is set, the RNC returns only if the carry flag is reset, and so on.

EXAMPLE 12-5

SAP-3 has a clock frequency of 1 MHz, the same as SAP-2. Write a program that provides a time delay of approximately 80 ms.

SOLUTION

Label	Mnemonic	Comment
	LXI SP,E000H	;Initialize stack pointer
	MVI E,08H	;Initialize counter
LOOP:	CALL F020H	;Delay for 10 ms
	DCR E	;Count down
	JNZ LOOP	;Test for 8 passes
	HLT	

You almost always use subroutines in complicated programs; this means that the stack will be used to save return addresses. For this reason, one of the first instructions in any program should be a LXI SP to initialize the stack pointer.

The 80-ms time delay program shown here starts with a LXI SP,E000H. This implies that the stack grows from address DFFFH toward lower memory. In other words, the stack pointer is decremented before the first push operation; this means that the stack begins at DFFFH.

The remainder of the program is straightforward. The E register is used as a counter. The program calls the 10-ms time delay 8 times. Therefore, the overall time delay is approximately 80 ms.

GLOSSARY

data pointer Another name for the HL register pair because some instructions use its contents to address the memory.
extended register A pair of CPU registers that act like a 16-bit register with certain instructions.
indirect addressing Addressing in which the address of data is contained in the HL register pair.
overflow A sum or difference that lies outside the normal range of the accumulator.
pop To read data from the stack.
push To save data in the stack.
stack A portion of memory reserved for return addresses and data.
stack pointer A 16-bit register that addresses the stack. The stack pointer must be initialized by an LXI instruction before calling subroutines.

SELF-TESTING REVIEW

Read each of the following and provide the missing words. Answers appear at the beginning of the next question.

1. An _____ is a sum or difference that lies outside the normal range of the accumulator. One way to detect an overflow is with the _____ flag.
2. (*overflow, carry*) To reset the carry flag, you may use an _____ followed by a CMC. STC stands for _____ the carry flag.
3. (*STC, set*) The ADC instruction adds the _____ flag and the contents of the specified register to the contents of the _____. SBB stands for subtract with _____.
4. (*carry, accumulator, borrow*) The RAL rotates all bits to the _____ with CY going to the LSB. RRC rotates the accumulator bits to the right with the LSB going to the carry flag.
5. (*left*) The CMP instruction compares the contents of the designated register with the contents of the accumulator. If the two are equal, the zero flag is _____. The CPI compares an immediate byte to the contents of the _____.
6. (*set, accumulator*) JM stands for jump if _____ . The program will branch to a new address if the _____ flag is set. JNZ means jump if not zero. With this instruction, the program branches only if the _____ flag is reset.
7. (*minus, sign, zero*) The LXI instruction is used to load register pairs. B is paired off with C, D with E, and H with _____. The HL register pair acts like a _____ pointer with some instructions. This type of addressing is called _____.
8. (*L, data, indirect*) The stack is a portion of memory reserved primarily for return addresses. The stack pointer is a 16-bit register that addresses the stack. It is necessary to initialize the stack pointer before calling any subroutines.

PROBLEMS

12-1. Write a program that adds decimal 345 and 753. (Use immediate bytes for the data.)

12-2. Write a program that subtracts decimal 456 from 983. (Use immediate data.)

12-3. Suppose that 1,024 bytes of data are stored between addresses 5000H and 53FFH. Write a program that copies these bytes at addresses 9000H to 93FFH.

12-4. Show a program that provides a delay of approximately 35 ms. If you use the SAP subroutines of Chap. 11, start your program with LXI SP,E000H.

12-5. Write a program that sends 1, 2, 3, . . . , 255 to port 22 with a time delay of 1 ms between OUT 22 instructions. (Use a LXI SP,E000H and a CALL F010H.)

12-6. Bytes arrive a port 21H at a rate of approximately 1 per millisecond. Write a program that inputs 256 bytes and stores them at addresses 8000H to 80FFH. (Use CALL F010H.)

12-7. Suppose that 512 bytes of data are stored at addresses 6000H to 61FFH and write a program that outputs these bytes to port 22H at a rate of approximately 100 bytes per second. (Use CALL F020H.)

12-8. A peripheral device is sending serial data to bit 7 of port 21H at a rate of 1,000 bits per second. Write a program that converts any 8 bits in the serial data stream to an 8-bit parallel word, which is then sent to port 22H. (Use CALL F010H.)

12-9. Suppose that 256 bytes are stored at addresses 5000H to 50FFH and write a program that converts each of these bytes into a serial data stream at bit 0 of port 22H. Output the data at a rate of approximately 1,000 bits per second. (Use CALL F010H.)