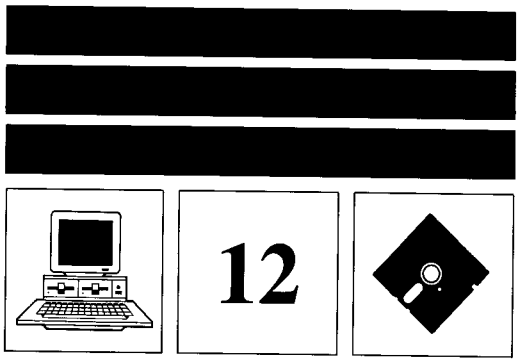

SAP-3

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

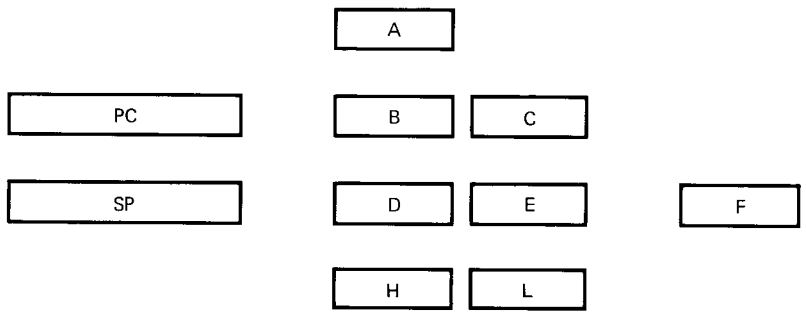

Fig. 12-1 SAP-3 programming model.

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-

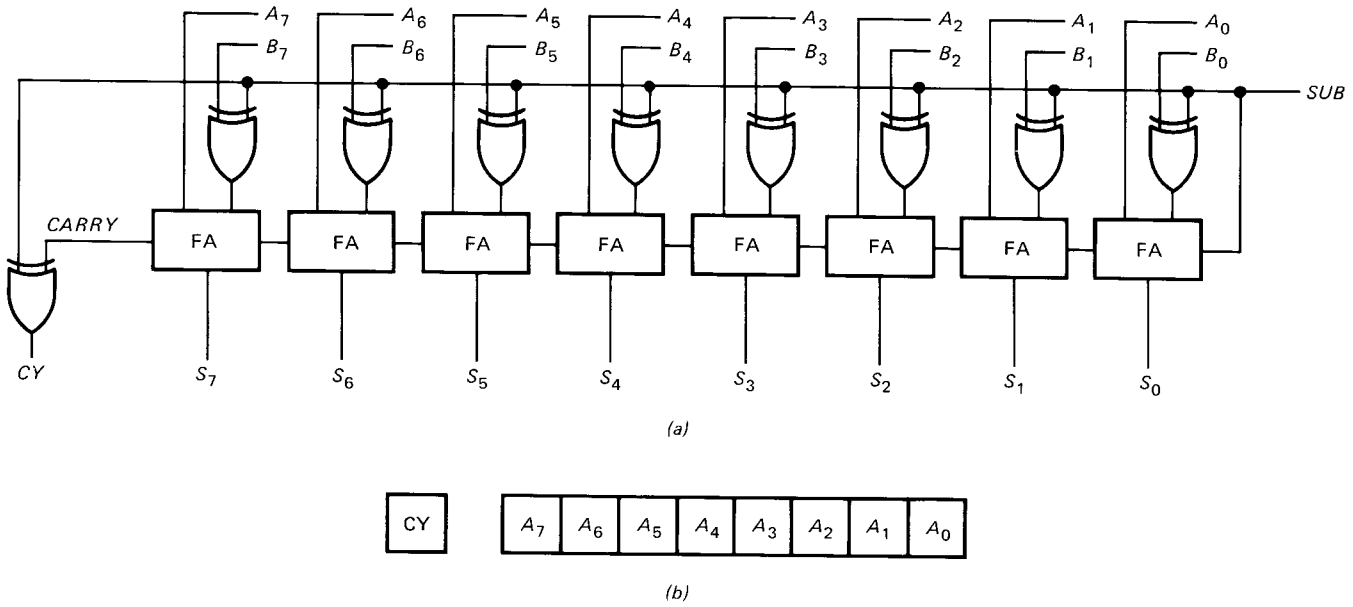

Fig. 12-2 (*a*) SAP-3 adder-subtractor (*b*) carry flag and accumulator.

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:

$$\begin{array}{r} 1000\ 0011 \\ 0001\ 0010 \\ +\qquad\quad 1 \\ \hline 1001\ 0110 \end{array}$$

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

$$\begin{array}{r} 0000\ 1111 \\ -\ 0000\ 0001 \\ \hline 0000\ 1110 \end{array}$$

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

$$\begin{array}{r} 0000\ 1111 \\ +\ 1111\ 1111 \\ \hline 1\ 0000\ 1110 \end{array}$$

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

$$\begin{array}{r} 0000\ 1100 \\ -\ 0001\ 0010 \\ \hline 1\ 1111\ 1010 \end{array}$$

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

$$\begin{array}{r} 0000\ 1100 \\ +1110\ 1110 \\ \hline 0\ 1111\ 1010 \end{array}$$

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:

$$\begin{array}{r} 1111\ 1111 \\ -\ 0000\ 0011 \\ \hline 1111\ 1100 \end{array}$$

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

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

Then the ADD D produces

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

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:

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

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
$S = 1$ $Z = 0$ $CY = 0$

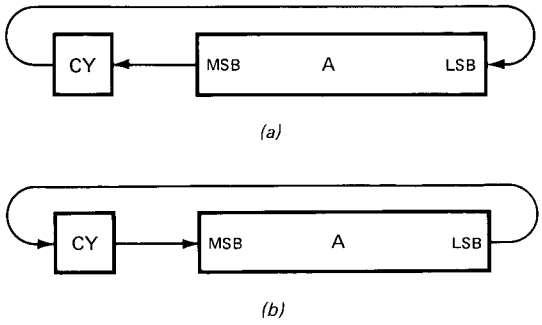

Fig. 12-3 (*a*) RAL; (*b*) RAR.

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.

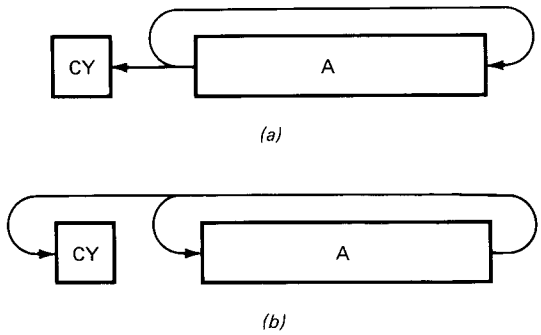

Fig. 12-4 (*a*) RLC; (*b*) RRC.

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

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

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

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

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

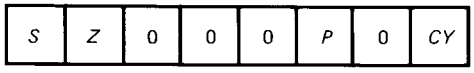

Fig. 12-5 F register stores flags.

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

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.

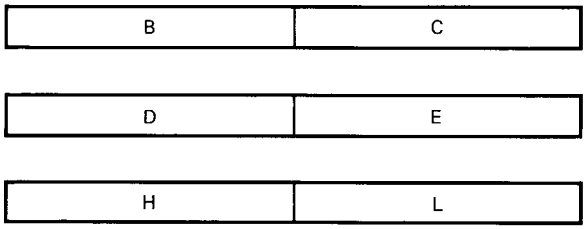

Fig. 12-6 Register pairs.

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}.

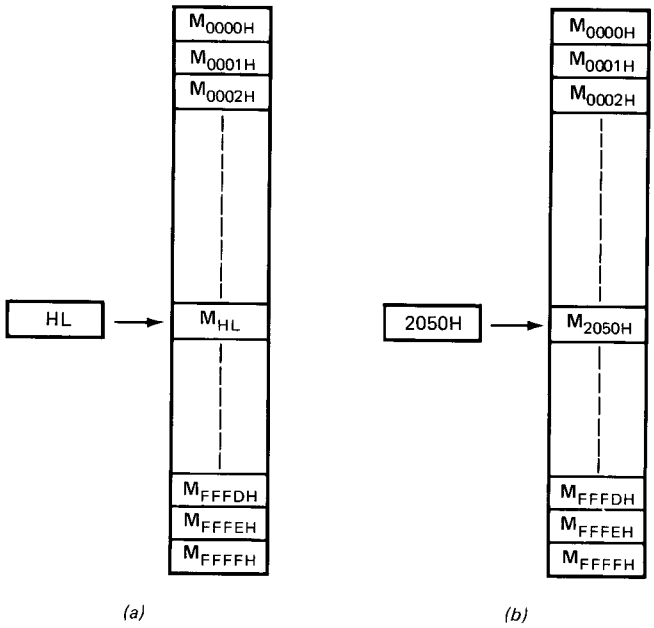

Fig. 12-7 (*a*) HL pointer; (*b*) pointing to 2050H.

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

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

executing a

MOV C,M

produces

$$\mathbf{C} = 87H$$

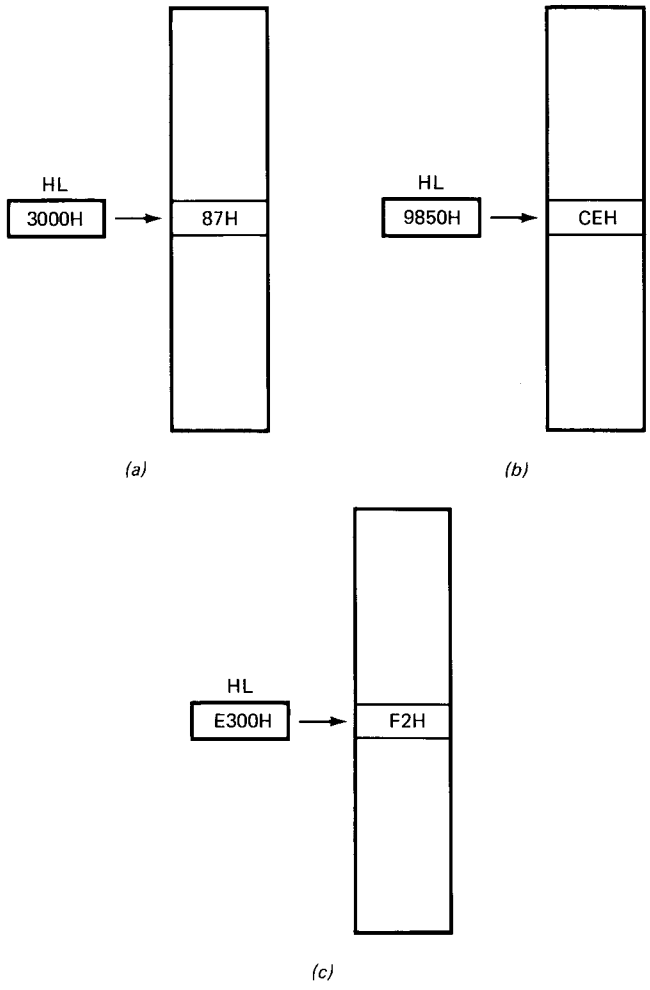

Fig. 12-8 Examples of indirect addressing.

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

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

a MOV A,M results in

$$\mathbf{A} = CEH$$

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,

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

As an example, if

$$\mathbf{HL} = E300H$$
$$\mathbf{B} = F2H$$

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

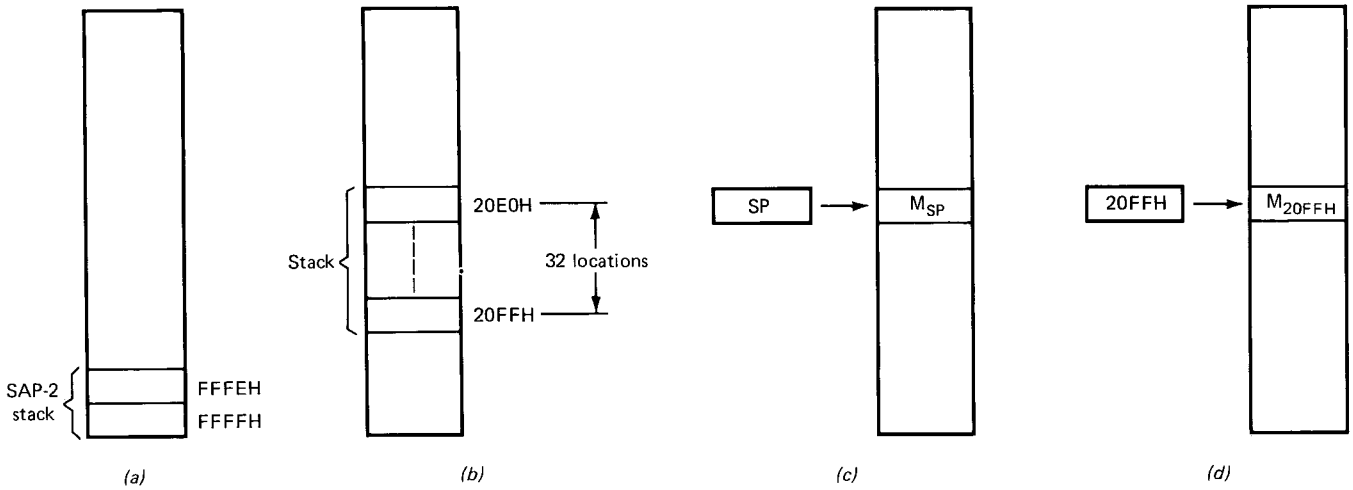

Fig. 12-9 (*a*) SAP-2 stack; (*b*) example of a stack; (*c*) stack pointer addresses the stack; (*d*) SP points to 20FFH.

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

$$\mathbf{SP} = 20FFH$$

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

$$\mathbf{PSW} = \mathbf{AF}$$

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*).

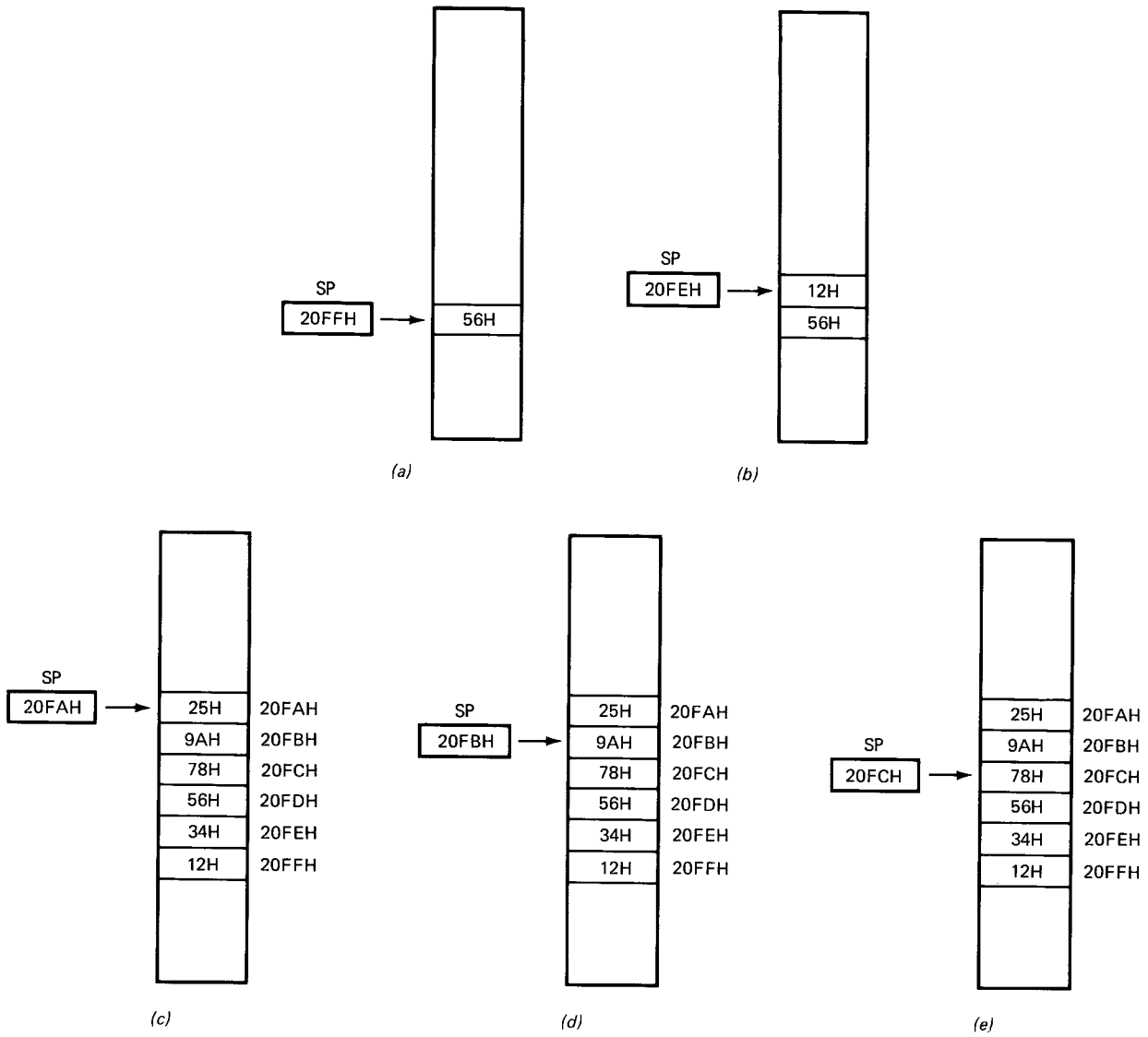

Fig. 12-10 Push operations: (*a*) high byte first; (*b*) low byte second; (*c*) 6 bytes pushed on stack; (*d*) popping a byte off the stack; (*e*) incrementing stack pointer.

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.

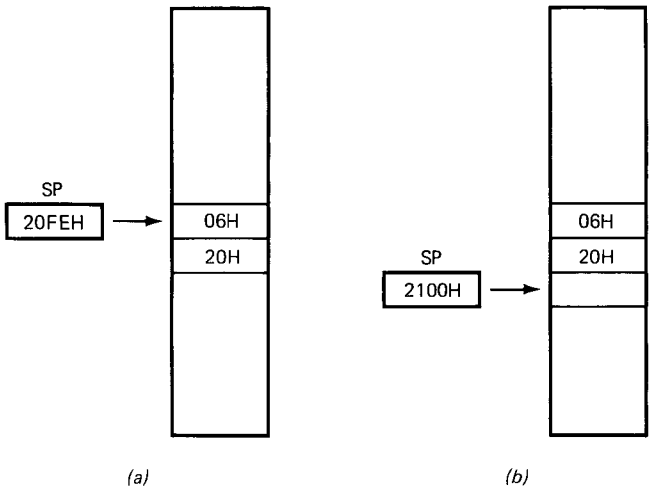

Fig. 12-11 (*a*) Saving a return address during a subroutine call; (*b*) popping the return address during a RET.

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.)