7

Program-Flow Control Instructions

Objectives

- List and classify the jump instructions
- Compare different unconditional jump instructions along with their range.
- Show how unconditional instructions are coded
- Describe the conditions used for bit and byte-level conditional jump instructions
- Introduce the looping technique
- Illustrate the call and return operations
- Develop the subroutines using call instructions
- Discuss the stack initialization and overflow
- Understand the time delay generation using software

Key Terms

- Call
- Context Saving/Retrieving
- Counter
- Destination Address
- Jump
- Loops
- Page
- Pointer
- Relocatability
- Return
- Return Address
- Stack Overflow
- Stack Pointer
- Subroutine
- Time Delay

The instructions discussed thus far were performing sequential operations, i.e. they were executed one by one in a sequence in the order that they appear in a program memory. Each instruction performs a single and simple operation. Many times we need to change the order of the execution of the instructions to other memory location, based on either certain conditions existing at the time or even without any condition. The program flow control instructions allow the microcontroller to alter the sequence of the program execution. These instructions make the program more flexible and versatile as required by real world applications. These instructions are also referred as *Branch instructions* or simply Jump instructions. The Jump and Call instructions in the 8051 have capability to alter the program flow.

7.1 JUMP INSTRUCTIONS

A jump instruction changes the content of program counter with the new program address usually referred as *destination address* or *target address*. This causes program execution to begin at the destination address. Two types of jumps are supported by the 8051: unconditional and conditional jumps.

7.1.1 Unconditional Jumps

The unconditional jump does not test any condition and jump is always taken. The 8051 supports three types of unconditional jumps: short (relative) jump, absolute jump and long (direct) jump. These jumps differ in range (in terms of bytes) over which the jump can be taken.

1. Short Jump

We know that the PC always contains the address of the next instruction (with respect to instruction that is being executed). Using a short jump, a program may only jump to instructions within 127 bytes in forward direction (+127) or 128 bytes in reverse direction (–128) with respect to contents of the PC (next instruction). A short jump is called *relative jump* because the destination address that is specified in the instruction (and then placed in the program counter) is relative to the address where the jump instruction is written. The advantage offered by relative jump is that it allows relocation, i.e. a program that is written using relative jump instructions can be placed (loaded) anywhere in the entire program space without reassembling. The second advantage is that only 1 byte is required to specify relative address of the destination location which saves the program bytes and increases the speed of execution. The instruction for short jump is SJMP and its format is,

SJMP *rel* // jump to relative destination address *rel*

SJMP is two-byte instruction: the first byte is op-code and second byte is relative address.

Calculating Relative Address from the Actual Destination Address

The relative address is relative to the value of the PC and to calculate relative address, the value of PC is subtracted from actual destination address. Consider the following program given in Figure 7.1.

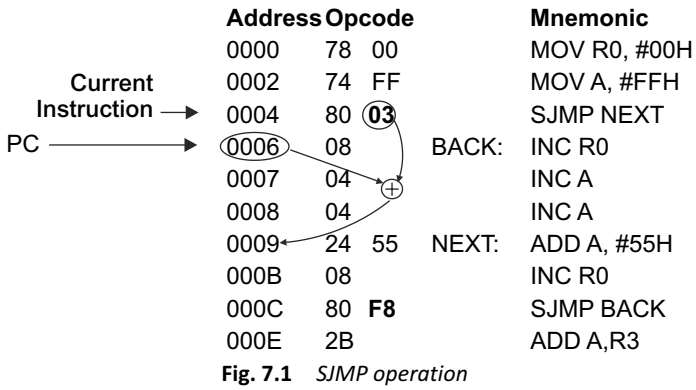


Fig. 7.1 *SJMP operation*

Consider the instruction "SJMP NEXT". It is written at address 0004H, so during the execution of this instruction, PC = 0006H, i.e. the PC points to the next instruction. Now destination instruction is "NEXT: ADD A, #55H" is located at address 0009H, so the relative address of destination instruction is 0009H-0006H = 03H. Therefore, relative address 03H is to be specified as a second byte of the instruction "SJMP NEXT" (see op-codes in the Figure 7.1). Conversely, if relative address is known, we can calculate actual address of the destination instruction. To calculate destination address, the second byte of SJMP instruction is added with the PC. For same example, add relative address into value of PC, i.e. add 03 with 0006H to get 0009H (actual address). The jump described in the above example is *forward jump* because destination address is ahead with respect to jump instruction.

Consider in the above example, the instruction "SJMP BACK". As can be seen, it is a *backward jump* because destination address is at lower address. Here, relative address is specified in 2's complement negative format. While

execution of the instruction "SJMP BACK", the PC = 000E and destination address is 0006H; therefore, displacement is 000EH-0006H=08H. Since displacement 08 is in backward direction, it is specified in 2's complement which is F8H. To get the actual destination address from the relative address, add displacement with value of PC, i.e. 0E+F8=1**06 (**Carry is neglected).

Fortunately, the programmer does not have to calculate the relative address; it is automatically calculated by an assembler, and programmer should use only labels.

The disadvantage of relative addressing is its limited range, i.e. within -128 to 127 bytes with respect to PC. If jump beyond this range is required then a jump can be made to address containing another jump instruction until desired address is reached.

Discussion Question What will be the second byte of the SJMP instruction (relative address) for following program fragment?

```
1000H          SJMP NEXT
…              …
1025H  NEXT:   ADD R1, #05H
```

Answer During the execution of the SJMP instruction, the PC is pointing to the next instruction, i.e. 1002H because SJMP is a two-byte instruction. The relative address will be difference between destination address and PC, therefore, relative address will be 1025H-1002H= 0023H= 23H, Note that only lower byte is considered.

Example 7.1

Assume that 8 switches are connected to port 1 pin and 8 LEDs are connected to port2 pins, write instructions to read status of all switches and send it to LEDs continuously.

Solution:

```
          MOV P1, #0FFH     // configure P1 as input port
REPEAT:   MOV A, P1         // read status of switches
          MOV P2, A         // send status of switches to LEDs
          SJMP REPEAT       // repeat task continuously and unconditionally
```

Example 7.2

Illustrate the common uses of SJMP instruction with suitable example.

Solution:

(i) SJMP is commonly used to repeat a part of a program (or whole program) indefinitely without checking any condition. The structure of such a program is shown below,

```
          …
REPEAT:   …               // repeat following instructions forever
          …
          …
          SJMP REPEAT
          …
```

(ii) SJMP is also used to skip part of the program as shown below,

```
          …
          SJMP SKIP       // continue program execution at label 'SKIP' and skip
          …               // following instructions
          …
          …
SKIP:     …
          …
```

(iii) Third common use of SJMP instruction is to stop program execution. For detailed explanation, refer topic 'How to stop program execution in the 8051?' at the end of Section 7.1.1 in this chapter.

2. Absolute Jump

The instruction for absolute jump is AJMP and its format is,

AJMP *add11* // jump to absolute destination address *add11*

AJMP instruction logically divides entire program memory into 32 pages of 2K Bytes each. The address range of each page is shown in Table 7.1.

Table 7.1 *Address range for pages in 8051*

Page No.	Address range (Hex)	Page No.	Address range (Hex)	Page No.	Address range (Hex)
00	0000- 07FF	0B	5800- 5FFF	16	B000- B7FF
01	0800- 0FFF	0C	6000- 67FF	17	B800- BFFF
02	1000- 17FF	0D	6800- 6FFF	18	C000- C7FF
03	1800- 1FFF	0E	7000- 77FF	19	C800- CFFF
04	2000- 27FF	0F	7800- 7FFF	1A	D000- D7FF
05	2800- 2FFF	10	8000- 87FF	1B	D800- DFFF
06	3000- 37FF	11	8800- 8FFF	1C	E000- E7FF
07	3800- 3FFF	12	9000- 97FF	1D	E800- EFFF
08	4000- 47FF	13	9800- 9FFF	1E	F000- F7FF
09	4800- 4FFF	14	A000- A7FF	1F	F800- FFFF
0A	5000- 57FF	15	A800- AFFF		

As can be seen from the Table, the upper five bits of an address in each page remains constant and represents a page number. For example, consider page 02, its address range is 1000H-17FFH. The upper five bits "**0001 0**000 0000 0000B – **0001 0**111 1111 1111B" represent the page number 02, and throughout the page, these bits are same.

AJMP instruction can jump within a page of 2K. Since upper five bits are same for each page, they need not be specified for destination address. Lower 11 bits hold address within a page and need to be specified. An absolute destination address is formed by taking page number (first five bits) of instruction following the AJMP (page number of address in the PC) and attaching 11 bits of destination address. This can be understood by Example 7.3.

Example 7.3

Show how the destination address in AJMP instruction is specified.

Solution:

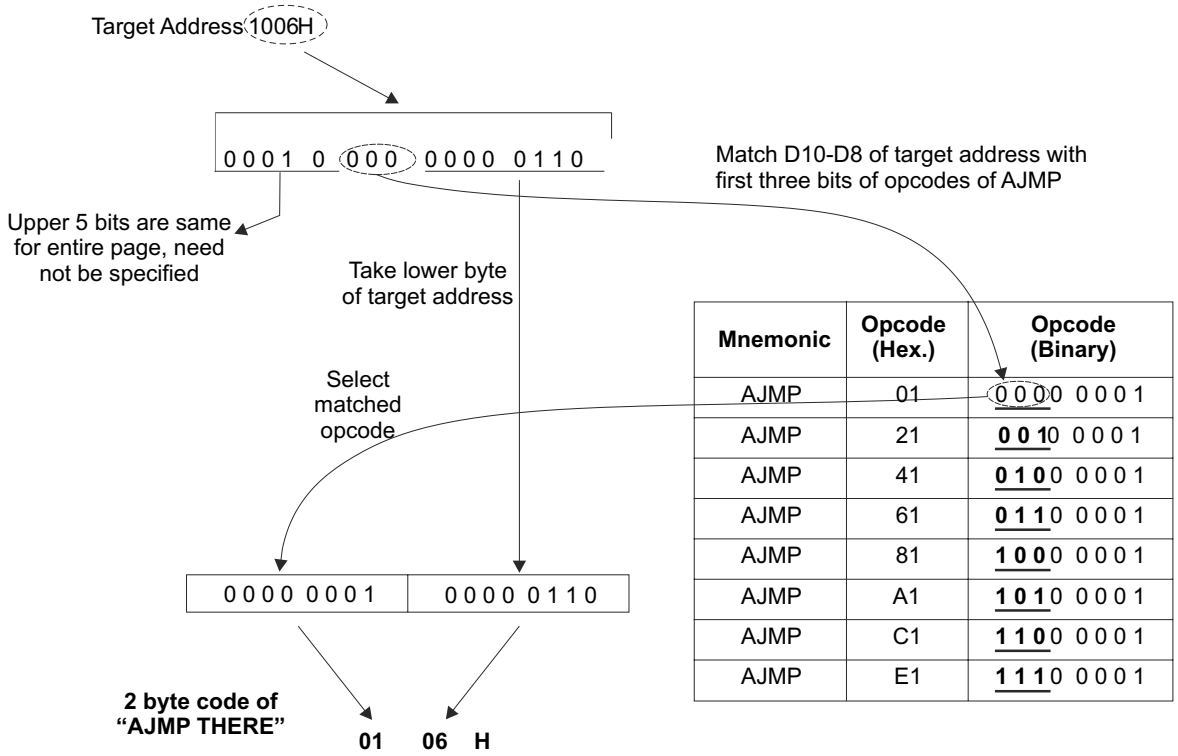
Consider the following program:

Address	Opcode	Mnemonic
1000	78 00	MOV R0,#00H
1002	**01 06**	AJMP THERE
1004	04	INC A
1005	08	INC R0
10**06**	24 55	THERE: ADD A,#55H
1007	08	INC R0

The label "THERE" represents absolute destination address (1006H) for the instruction "AJMP THERE". Since instructions "AJMP THERE" as well as "THERE: ADD A,#55H" belongs to same page (Page 2), we need not to specify upper five bits of destination address, remaining 11 bits are specified as follows.

Observe the target (destination) address, i.e. address 1006H,

The lower 8 bits of destination address (*add11*) are specified directly as a second byte of the AJMP instruction (06 in the example). The remaining 3 bits are specified indirectly by choosing one out of the eight different op-codes of AJMP instruction as shown in Figure 7.2. Match these three bits (bit D10, D9 and D8 of destination address) with the first three bits of the op-codes of the AJMP. Choose the op-code for which these three bits are same. In our example bits, D10-D8 are 000 which matches with the first three bits of op-code 01H (**000**0 0001), therefore choose op-code 01H for AJMP. This way, instruction "AJMP THERE" is coded as "01 06 H".



Mnemonic	Opcode (Hex.)	Opcode (Binary)
AJMP	01	0 0 0 0 0 0 0 1
AJMP	21	0 0 1 0 0 0 0 1
AJMP	41	0 1 0 0 0 0 0 1
AJMP	61	0 1 1 0 0 0 0 1
AJMP	81	1 0 0 0 0 0 0 1
AJMP	A1	1 0 1 0 0 0 0 1
AJMP	C1	1 1 0 0 0 0 0 1
AJMP	E1	1 1 1 0 0 0 0 1

Fig. 7.2 *Opcode generation process for instruction AJMP*

As another example, let us find the op-code for same instruction "AJMP THERE" if the destination address is 1208H.

The upper 5 bits are not required to be specified because they are directly taken from the PC; lower 8 bits are taken from lower byte of the destination address, i.e. 08H. The bits D10-D8 are 010 for the destination address which matches with first three bits of op-code 41H, therefore the two-byte code for instruction "AJMP THERE" is "41 08". If assembler is used to generate the machine codes, this process will be automatically done by an assembler.

When AJMP instruction occurs at page boundary, i.e. at address x7FFH or xFFFH (or x7FEH or xFFEH), the next instruction starts at address x801 or x001 (or x800 or x000), which places destination address on the next page, however, this page change does not cause any problem when there is a forward jump, but causes trouble when the jump is backward. The assembler should report this error while assembling the program, so necessary action (of writing such instructions at other address) can be taken by the programmer to resolve this problem.

The advantage of using AJMP instruction is similar to SJMP, i.e. only two-byte machine code, also jump range is anywhere within 2K page and program is relocatable provided that relocated code begins at the start of a page.

Note: AJMP (and ACALL) are the special instructions for which there are eight op-codes. This is an example of compromise between number of instructions possible for the 8051 and number of bytes required for AJMP (and ACALL) instructions.

THINK BOX 7.1

We know that AJMP and ACALL instructions each have eight op-codes. Can we use any of the op-code any time?
No. It is decided by destination address specified in an instruction.

3. Long Jump

The instruction for long jump is LJMP and its format is,

LJMP *add16* // jump to direct destination address *add16*

LJMP instruction allows a jump to any location within entire program address space, i.e. any where from 0000H to FFFFH. It is a 3-byte instruction, first byte is the op-code for the instruction and the second and third byte represents directly 16-bit destination address. Therefore long jump is also referred as direct jump. Advantage of LJMP instruction is its address range. The disadvantage is that it is three byte instruction and programs using it are not relocatable. LJMP is normally used in larger programs. The jump ranges of all unconditional jumps are illustrated in Figure 7.3.

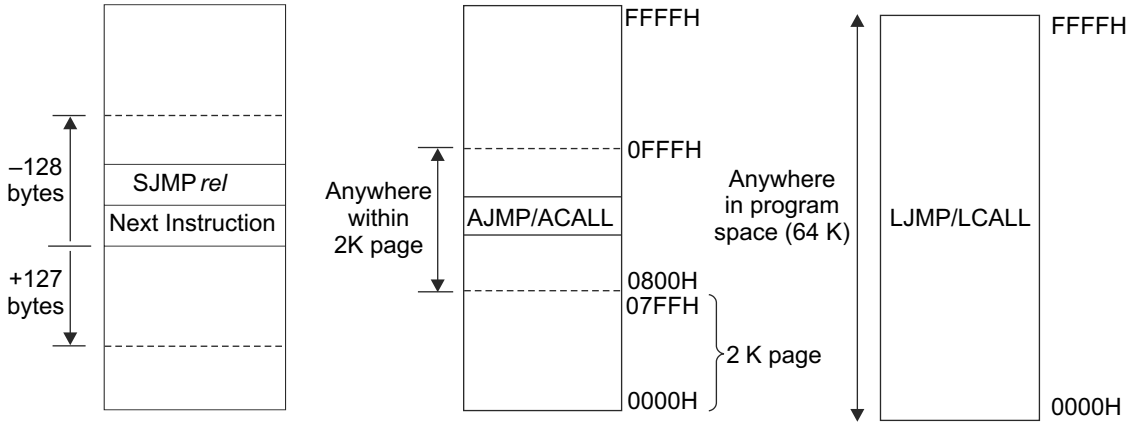


Fig. 7.3 *Range of SJMP, AJMP/ACALL and LJMP/LCALL (not to the scale)*

Discussion Question Compare SJMP, AJMP and LJMP instructions with respect to

(i) Jump range
(ii) Number of bytes of instructions
(iii) Speed of execution
(iv) Relocatability

Answer:

(i) **Jump range**
SJMP: –128 to +127 bytes with respect to PC
AJMP: Anywhere within the current page of 2Kbytes
LJMP: Anywhere within the entire 64Kbytes of program memory

(ii) **Number of bytes of instructions**
SJMP: 2 bytes
AJMP: 2 bytes
LJMP: 3 bytes

(iii) **Speed of execution**
SJMP: 2 machine cycles
AJMP: 2 machine cycles
LJMP: 2 machine cycles

(iv) **Relocatability**
SJMP: Fully relocatable, because relative distance between SJMP instruction and destination address will remain same irrespective of starting address of a program.
AJMP: Relocatable, as long as relocated code begins at the start of a page.
LJMP: Not relocatable, because this instruction uses exact address in the instruction, therefore when program is reloaded at different address, the destination address also changes. So we need to reassemble the code.

Discussion Question When is it preferred to use

(i) SJMP instruction
(ii) LJMP instruction

Answer

(i) SJMP is preferred in a program when distance between jump instruction and destination address is less, i.e. within –128 to +127 bytes, and when available program memory is restricted.
(ii) LJMP is preferred in larger programs where required jump range is more.

How can you stop the Program Execution in the 8051?

The 8051 does not have an instruction to stop the program execution. Therefore, once it is powered on, it always executes some instructions (except for power saving modes).

If a task is completed (or a task have to wait for external event or command), keep the microcontroller busy in an infinite loop using unconditional jump instruction. Jump to the same instruction is commonly used to implement the infinite loop. The SJMP instruction is commonly used for this purpose as shown in the following instruction.

'HERE: SJMP HERE' or 'HERE: AJMP HERE'

These instructions jump to the same instruction forever, which is as good as stopping further program execution.

7.1.2 Conditional Jumps

Conditional jump instructions first test the condition specified in an instruction op-code. If the condition is true then the jump is taken by modifying the value of the PC, otherwise program execution continues to the next sequential instruction. All conditional jumps are short relative jumps. The conditional jumps are categorized as bit and byte jumps.

1. Bit Jumps

Bit jump instructions check the status of addressable bit specified as a part of instruction. The jump is taken to specified relative address if the condition (for specified bit) is satisfied (or true), otherwise program execution proceeds to the next instruction. The destination address is calculated by adding the relative address with the PC. These instructions are used to monitor the status of specified bit and to take the decision based on its value. Bit jump instructions are listed below.

JC *rel*	// jump to relative address *rel* if C=1 (Jump if Carry)
JC BACK	// jump to (continue program execution from) label BACK if C=1
JNC *rel*	// jump to relative address *rel* if C=0 (Jump if No Carry)
JNC NEXT	// jump to (continue program execution from) label NEXT if C=0
JB *bit*, *rel*	// jump to relative address *rel* if *bit* =1 (Jump if Bit)
JB 00H, TMP	// jump to label TMP if bit 00H is set (content of 00H is 1)
JNB *bit*, *rel*	// jump to relative address *rel* if *bit* =0 (Jump if No Bit)
JNB 05H, MP	// jump to label MP if bit 05H is clear (content of bit address 05H is 0)
JBC *bit*, *rel*	// jump to relative address *rel* if *bit* = 1, and then clear *bit* (Jump if Bit, then Clear)
JBC 50H, XY	// jump to label XY if bit 50H is set (content of bit address50H is 1), then clear bit 50H

THINK BOX 7.2

How is 'JBC bit, rel' instruction more efficient than other bit jump instructions?

It combines bit test, clear bit and jump operation in a single instruction.

Example 7.4

Illustrate the use of JNC, JC JNB, JB and JBC instructions.

Solution:

(i) Use of JNC

```
        CLR C
        MOV A, #10H
        MOV R0, A
AGAIN:  ADD A, R0
        JNC AGAIN     // repeat addition of A with R0 until carry flag is set
        …
```

(ii) Use of JC

```
                …
                SETB C          // set carry for illustration
                JC AHEAD        // since C=1, continue program execution from label
                …               // AHEAD
                …
AHEAD:          MOV A, P1
                …
```

(iii) Use of JNB and JB

Assume that a pushbutton switch is connected to pin P1.0 and, when a switch is pressed, the logic high is given to the pin. Otherwise, it remains at low logic. Write instructions to toggle the status of pin P2.0 every time the switch is pressed.

```
        SETB P1.0       // configure pin P1.0 as an input
WAIT:   JNB P1.0, WAIT  // wait until switch is pressed
        CPL P2.0        // complement P2.0
        SJMP WAIT       // repeat the operation
```

Alternatively, JB can be used as shown below:

```
        SETB P1.0       // configure pin P1.0 as an input
WAIT:   JB P1.0, COMP   // If a switch is pressed, jump to COMP
        SJMP WAIT       // repeat the operation if switch is not pressed
COMP:   CPL P2.0        // complement P2.0
        SJMP WAIT       // repeat the operation
```

Assume that a pushbutton switch is connected to pin P1.0. When switch is pressed, logic low is given to the pin; otherwise it remains at high logic. Write instructions to toggle the status of pin P2.0 every time the switch is pressed.

```
        SETB P1.0       // configure pin P1.0 as an input
WAIT:   JB P1.0, WAIT   // wait until switch is pressed
        CPL P2.0        // complement P2.0
        SJMP WAIT       // repeat the operation
```

Alternatively, JNB can be used as shown below:

```
        SETB P1.0       // configure pin P1.0 as an input
WAIT:   JNB P1.0, COMP  // If switch is pressed jump to COMP
        SJMP WAIT       // repeat the operation if switch is not pressed
COMP:   CPL P2.0        // complement P2.0
        SJMP WAIT       // repeat the operation
```

(iv) **Use of JBC:** It is illustrated in **Example 16.12** (in the more efficient method part of example).

---

2. Byte Jumps

Byte jump instructions check bytes of data to make a decision whether to jump to destination address or continue to the next instruction. The byte jump instructions are listed below.

```
JZ rel                  // jump to relative address rel if A is 0 (Jump if Zero)
JNZ rel                 // jump to relative address rel if A is not 0(Jump if Not Zero)
CJNE A, direct, rel     // compare A with contents of address direct and jump to relative
                        // address rel if they are not equal, if A is less than contents of address
                        // direct, set carry flag to 1, otherwise clear to 0
                        // CJNE means Compare and Jump if Not Equal)
CJNE A, #data, rel      // compare A with immediate value data and jump to relative address
                        // rel if they are not equal, if A is less than immediate value data, set
                        // carry flag to 1, otherwise clear to 0
```

```
CJNE Rn, #data, rel      // compare Rn with immediate value data and jump to relative address
                         // rel if they are not equal, if Rn is less than immediate value data, set
                         // carry flag to 1, otherwise clear to 0
CJNE @Ri,#data, rel      // compare contents of address in Ri with immediate value data and jump to relative
                         // address rel if they are not equal, if contents of address in Rn is less than immediate
                         // value data, set carry flag to 1, otherwise clear to 0
DJNZ Rn, rel             // decrement register Rn by 1 and jump to relative address rel if
                         // content of Rn is not zero after decrement operation, no flags are affected
                         // DJNZ means Decrement and Jump if Not Zero
DJNZ direct, rel         // decrement contents of address direct by 1 and jump to relative address rel if content of
                         // address direct is not zero after decrement operation, no flags are affected
```

For all the byte jump instructions, the jump is taken to specified relative address if condition is satisfied (true), otherwise program execution proceeds to the next instruction. The destination address is calculated by adding the relative address with the PC. These instructions are used to monitor the value of specified byte and to take the decision based on its value. It should be noted that there is no zero flag in the 8051, the instructions JZ and JNZ checks the Accumulator for zero. The DJNZ instruction decrements the specified operand first and then checks operand for zero. The CJNE instruction does not affect any of its operands.

Example 7.5

Write instructions to monitor the status of port 2 continuously until it is 55H.

Solution:

```
        MOV P2, #0FFH          // configure P2 as input
REPEAT: MOV A, P2              // read status of P2 in to A
        CJNE A, #55H, REPEAT   // repeat until P2 status is 55H
        …
```

Example 7.6

(i) Write instructions to increment contents of R5 until it becomes 50H.

(ii) Modify above program fragment to increment R5 until it is equal to contents of address 30H.

Solution:

```
(i)          …
     REPEAT: INC R5                 // increment R5 until its value is 50H
             CJNE R5, #50H, REPEAT
             …
(ii) REPEAT: INC R5                 // increment R5 until its value is equal to content of address 30H
             MOV A, R5
             CJNE A, 30H, REPEAT
             …
```

The jump instructions are widely used in looping, which is discussed in detail in the next topic.

7.2 | LOOPS

In many cases, we have to repeat a single task for several times, and the best way to do this is by looping. The *looping* is a programming technique used to repeat the sequence of instructions several times until certain conditions are met. The repeatability is the key feature and reason for popularity of the microcontroller (or computer) based systems. The looping allows us to develop concise and efficient programs. The most common requirement for loops is to specify loop

count, which determines the number of times a task has to be repeated. The loop count is loaded into some register or memory location before the loop is started. The register or memory location that holds the loop count is usually referred as a counter for that task or a program. After each iteration, a loop counter is decremented by one and a test is made to check if the loop counter is zero, if it is zero then the loop is terminated otherwise the task is repeated. There are two basic types of loops: unconditional and conditional loops. Unconditional loops repeat the task indefinitely without checking any condition until system is reset or power down. Conditional loops repeat the task until certain condition exists.

In the 8051, the instructions DJNZ and CJNE are used to repeat the loop for fixed number of times, while the instructions JZ/JNZ and all bit jump instructions are used to repeat the loop until a flag or bit is set to desired state. The typical structure of loop using the DJNZ instruction is shown below:

```
        MOV R4, #10      // load the count (number of times loop to be repeated)
LOOP:   ...              // begin loop
        ...
        ...
        ...              // end loop
        DJNZ R4, LOOP    // check whether loop is repeated required times, if not repeat it, otherwise, exit from the
                         // loop and continue program execution at next instruction
        ...
```

Let us consider a few simple examples to understand the looping.

Example 7.7

Add 5 to A register ten times.

Solution:

The given task can be performed in many ways; three simple ways are given here.

(i) A counter is initialized with 10 and the counter is decremented after every addition and checked for zero using the DJNZ instruction, if it is not zero then addition is repeated, otherwise the loop has already been repeated 10 times and program execution will come out of the loop and continue to the next task.

```
        CLR A            // clear A
        CLR C            // clear carry flag
        MOV R3, #10      // loop counter R3=10
REPEAT: ADD A, #05       // add 05 to A
        DJNZ R3, REPEAT  // repeat addition operation 10 times
```

(ii) A counter is initialized with 10 and the counter is decremented after every addition using the DEC instruction and compared with zero using the CJNE instruction; if it is not zero then addition is repeated, otherwise program execution will come out of the loop and continue to the next task.

```
        CLR A                  // clear A
        CLR C                  // clear carry flag
        MOV R3, #10            // loop counter R3=10
REPEAT: ADD A, #05             // add 05 to A
        DEC R3                 // decrement the loop count
        CJNE R3, #00, REPEAT   // repeat addition operation 10 times
```

(iii) A counter is initialized with 00 and the counter is incremented after every addition using the INC instruction and compared with 10 using the CJNE instruction; if it is not equal to 10, then addition is repeated, otherwise program execution will come out of the loop and continue to the next task

```
        CLR A                  // clear A
        CLR C                  // clear carry flag
        MOV R3, #00            // loop counter R3=00
REPEAT: ADD A, #05             // add 05 to A
        INC R3                 // decrement the loop count
        CJNE R3, #10, REPEAT   // repeat addition operation 10 times
```

Example 7.8

Write a program to add contents of ten memory locations from 20H onwards.

Solution:

The result of addition may be greater than 8 bits, therefore we need to store result in two 8-bit locations. After each addition carry flag is checked; if it is set, then content of address 41H is incremented which was initialized with value 00H. This will correct the 9th or higher bits of the result.

```
        MOV R2, #0AH        // counter for addition of 10 numbers
        MOV R0, #20H        // initialize pointer to first memory address
        CLR C
        MOV 41H, #00H       // store higher byte of result at 41H
        CLR A               // clear contents of A
NEXT:   ADD A,@R0           // add number pointed by R0 with A
        JC AHEAD
        SJMP SKIP
AHEAD:  INC 41H             // If carry is generated increment contents of
                            // address 41H
SKIP:   INC R0              // point to next number
        DJNZ R2, NEXT       // repeat addition 10 times
        MOV 40H, A          // store LSByte of result at address 40H
```

Nested Loops

By using byte jumps we can repeat the loop for a maximum of 256 times (FFH). If we want to repeat the loop more than 256 times then we have to use a loop inside loop which is referred as *nested loop*. In such a case, we use two (or more) loop counters, refer Example 7.9.

Example 7.9

Read port P0 and send its value on port P2 five hundred times.

Solution:

The number 500 is greater than 256; therefore, we have to use nested loops, each with separate loop count. Let us take 50 as the loop count for outer loop and 10 for inner loop. The total loop count is, therefore, 50 x 10 = 500.

```
        MOV R1, #10         // outer loop count
OUTER:  MOV R2, #50         // inner loop count
INNER:  MOV A, P0           // read the value of P0 and send value to P2
        MOV P2, A
        DJNZ R2, INNER      // inner loop repeat 50 times
        DJNZ R1, OUTER      // outer loop repeat 10 times
```

One important application of loops is in generating time delay using software. Delay generation using software is discussed in detail in Section 7.5 (Time-delay generation using timers is discussed in Chapter 14).

THINK BOX 7.3

Consider the following loop structure.

```
        MOV R2, #COUNT
LOOP:   …
        ..
        DJNZ R2, LOOP
```

What should be the value of COUNT to have maximum iterations of the above loop?

00H. Because DJNZ instruction first decrements the value of specified register (R2 in this case) and then check it for zero.

7.3 | CALLS AND SUBROUTINES

While developing larger programs, many times, we may require it to perform a task (or subtask) repeatedly. Instead of writing group of instructions for this task repeatedly, we can write these instructions as subprograms separately from the main program. This group of instructions is called *subroutine*. The subroutines are used by the main program many times as and when required. When a subroutine is required to be executed, a jump is made to the first instruction of the subroutine. The jump to the subroutine is more commonly referred as a *call*. Upon completion of the subroutine, another jump is made to the calling program (main) to resume the operation. This jump back is referred as *return*. The return is always made to instruction immediately next to the instruction for call.

A subroutine offers following advantages:

1. They simplify program-development process, they allow larger programs to be divided into smaller modules, these modules may be developed independently to speed up the development process; module may contain single subroutine or more than one subroutines.
2. Subroutines may be reused and therefore they save memory space.
3. Modular approach makes debugging and testing of the program easier and therefore saves time and money required for the development.

The subroutines are also referred as *routines* or *procedures*. The only disadvantage of using subroutine is that they reduce the speed of execution of a program because extra time is required in switching between main program and subroutines.

In the 8051, there are two instructions for calling a subroutine, LCALL (long call) and ACALL (absolute call), and RET instruction for return. The formats of these instructions are given below:

LCALL *add16*	// call the subroutine at address *add16* located anywhere in the entire
	// program memory space of 64KB, also save (push) the address of the
	// next instruction following LCALL (return address) on to the stack

LCALL DISPLAY	// call a subroutine DISPLAY, save return address on the stack

ACALL *add11*	// call the subroutine at address *add11* located on same page as the next
	// instruction, also save the address of the next instruction following
	// ACALL (return address) on to the stack

ACALL COMPARE	// call a subroutine COMPARE, save return address on the stack
RET	// return to the calling program by retrieving (pop) the return address from the stack

Before we understand the above instructions, we need to understand the relation of these instructions with the stack because these instructions use the stack.

1. Relation of Calls and Stack

A call (ACALL or LCALL) causes a jump to address where called subroutine is located. After completing a subroutine, the main program (calling program) execution should resume at instruction next to the call instruction. The stack automatically keeps track of where microcontroller is supposed to return after executing the subroutine. When we call a subroutine, the address of the instruction next to the call instruction is automatically saved on to the stack; this address is called return address. At the end of a subroutine, the return (RET) instruction will load return address from the stack to PC to resume the execution of the calling program. The stack pointer (SP) register is used to access the stack. Stack pointer always points to top of the stack, i.e. last memory address accessed.

The process of calling a subroutine and returning from it to resume the operation of the calling program using ACALL instruction, is described in the following steps.

(a) ACALL instruction will save the return address (PC) on the stack using two push operations: lower byte at address SP+1, and higher byte at SP+2. (Stack pointer is automatically incremented before pushing each byte).
(b) Address of subroutine is placed in the PC.
(c) Subroutine is completed.
(d) A RET instruction at the end of the subroutine will retrieve the return address to the PC from the stack using two pop operations. (Stack pointer is automatically decremented after each pop operation).
(e) Calling (main) program resumes its operation from the next instruction after ACALL.

The operation of ACALL instruction is illustrated in Figure 7.4.

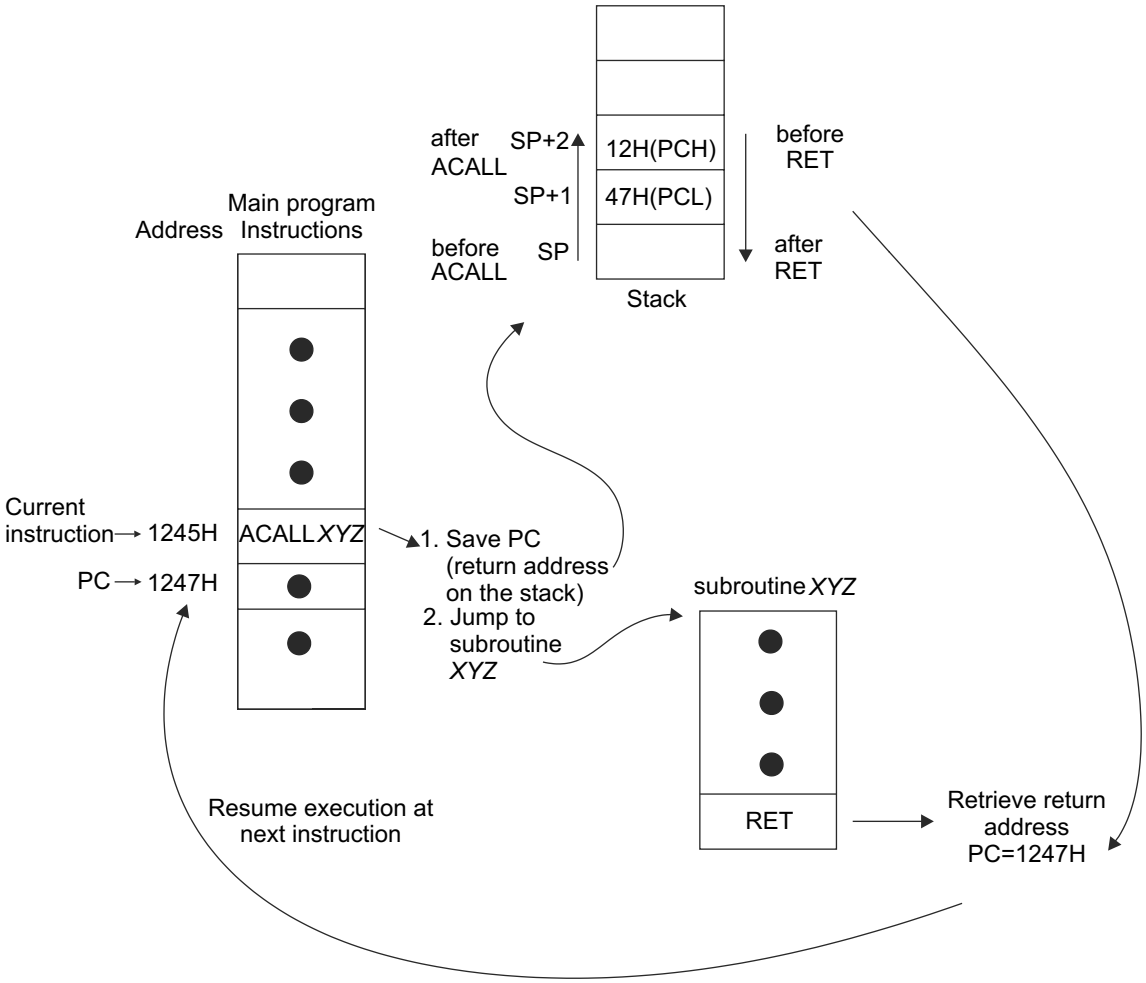


Fig. 7.4 *Operation of ACALL instruction*

The operation of LCALL is exactly same as the ACALL instruction. The only difference it has is that it can call subroutine anywhere in the entire program space of 64Kbytes, while ACALL can call the subroutine within a page of 2Kbytes in which instruction following ACALL is located.

ACALL is a 2-byte instruction, while LCALL is a 3 byte instruction. The way 11-bit address is specified for ACALL instruction is exactly similar to the AJMP instruction (refer Example 7.3)

Example 7.10

Illustrate the use of ACALL instruction to use the subroutine in a program.

Solution:

Assume that we want to write a very simple subroutine to add two numbers stored at addresses 40H and 41H, and place the result at address 42H.

```
MOV SP, #50H            // initialize SP
ACALL ADDITION          // call subroutine ADDITION
…
…
```

```
ADDITION: PUSH ACC          // save the registers (A) used in the subroutine
          PUSH PSW          // save PSW because carry may be affected
          MOV A, 40H        // add numbers
          ADD A, 41H
          MOV 42H, A        // save result at address 42H
          POP PSW           // retrieve the saved registers
          POP ACC
          RET               // return to calling program
```

Note the use of PUSH and POP instructions in above program. They are used to save the registers or memory locations used by a subroutine to perform its internal operations and retrieve the saved contents before returning back to the calling program. In given program, A is saved on stack because it is used by the subroutine, which may contain data of main program and PSW is saved because addition operation may change carry flag. Once all the tasks of subroutine are completed, the saved data is retrieved in same registers. It is done by POPing data in a reverse sequence than a PUSHing because stack is Last In First Out (LIFO) memory.

These extra instructions should always be written to make sure that subroutine does not modify accidently any important data (in register or memory location) used by a main program. Therefore, it is recommended to write subroutines which are transparent to main program, i.e. everything is same before and after a subroutine is called (and executed), this will be very much helpful when a larger program is developed in modules and each module is developed by a different person because the other person should be able to use the subroutine directly without knowing inner details of the subroutine.

Example 7.11

Illustrate the use of LCALL instruction to use subroutine in a program.

Solution:

(i) Assume that we want to send numbers 00H to FFH in increment of one to P2 after every 1 second, assume subroutine 'DELAY' is already available which generates a delay of 1 second.

```
          …
BACK:     MOV P2, A
          LCALL DELAY       // call subroutine 'DELAY'
          INC A
          SJMP BACK
DELAY:    …
          …
          RET               // return to main program
```

(ii) The rectangular wave of 25% duty cycle is to be generated on port pin P1.0. Assume that delay subroutine is available.

```
BACK:  SETB P1.0            // set P1.0 pin, High portion of rectangular wave
       LCALL DELAY          // call DELAY subroutine
       CLR P1.0             // clear P1.0 pin low portion of rectangular wave
       LCALL DELAY          // call DELAY subroutine 3 consecutive times to get
                            // low portion 3 times larger than high time to get 25% duty cycle
       LCALL DELAY
       LCALL DELAY
       SJMP BACK            // repeat operation forever
```

Example 7.12

Write a subroutine to an convert 8-bit binary number stored in the Accumulator into an equivalent BCD number.

Solution:

Refer Example 9.1 for the explanation of conversion process of binary to BCD.

```
            …
            MOV A, #30H       // binary number to be converted
            ACALL BIN2BCD     // call conversion routine
            …
            MOV A, #85H       // other number to be converted
            ACALL BIN2BCD     // call again conversion routine
            SJMP HERE         // skip subroutine
BCD2BIN:    MOV B, #64H       // divisor (100)
            DIV AB            // A=00 quotient, B=30H remainder (these values are for first number –30H)
            MOV R1, A         // store 100's digit in R1 (unpacked BCD digit)
            MOV A, B          // copy remainder in to A for next division
            MOV B, #0AH       // next divisor (10)
            DIV AB            // A=4, B= 8 (these values are for first number –30H)
            MOV R2, A         // 10's digit (unpacked BCD digit)
            MOV R3, B         // 1's digit (unpacked BCD digit)
            RET               // return to main program
HERE:       SJMP HERE         // end of program; loop forever
```

Note that the result obtained by the above program is unpacked BCD digits. We may convert this unpacked digits into packed digits by rotate and OR operations if required.

THINK BOX 7.4

Which op-code is undefined in the 8051?

A5.

The other examples of subroutines are given throughout remainder of the chapters.

2. Cautions while Developing Subroutines

Subroutines allow powerful and efficient way of developing modular programs; however they require more efforts from programmer/system designer during a program development. There are few simple cautions that have to be considered by a programmer; they are listed below.

(a) Always terminate subroutine with RET instruction: The RET instruction will resume operation in a main program. The program execution will not return to the main program when RET instruction is omitted and, therefore, the program will give an erroneous result.

(b) Parameter passing and context saving: Save registers (or memory locations) that are modified (used) by the subroutine/s for its internal operations. It is very common to forget to save the registers that are used by subroutines. This may overwrite or modify the important data in a main program resulting in strange behavior of the program. It is preferred to save all the registers used in the subroutine and PSW just at the beginning of the subroutine (unless application demands to change the specific register), and at the end of a subroutine, retrieve all the saved registers. To speed up the process of context saving (or switching), switch the register bank, which relieves the program from saving the registers R0 to R7. This is one of the major reasons for providing register banks in the 8051.

(c) Context retrieving: Another common error is to save registers onto the stack and then forget to retrieve them all off the stack before exiting the subroutine. An unequal number of save and retrieval of registers will result in return at wrong address. Therefore always make sure that equal number of PUSH and POP instructions are used.

The program of Example 7.12 is rewritten with proper context saving and retrieving in Example 7.13.

Example 7.13

Write a subroutine with context saving and retrieving to convert 8-bit binary number into equivalent BCD number.

Solution:

The binary number to be converted is placed into internal RAM address 10H before calling the subroutine. The three unpacked BCD digits will be stored in R1, R2and R3 of bank 3.

```
            MOV SP, #60H          // initialize the stack pointer at higher address
            …
            …
            MOV 10H, #30H         // binary number to be converted
            ACALL BIN2BCD         // call the conversion routine
            …
            MOV 10H, #85H         // other number to be converted
            ACALL BIN2BCD         // call again conversion routine
            SJMP HERE             // skip subroutine
BCD2BIN:    PUSH ACC              // save A on the stack
            PUSH B                // save B on the stack
            PUSH PSW              // save PSW on the stack
            SETB D3               // switch to register bank 3
            SETB D4
            MOV A, 10H            // get number to be converted
            MOV B, #64H           // divisor (100)
            DIV AB                // A=00 quotient, B=30 H remainder (these values are for first number –30H)
            MOV R1, A             // store 100's digit in R1 (unpacked BCD digit)
            MOV A, B              // copy remainder into A for next division
            MOV B, #0AH           // next divisor (10)
            DIV AB                // A=4. B= 8 (these values are for first number –30H)
            MOV R2, A             // 10's digit (unpacked BCD digit)
            MOV R3, B             // 1's digit (unpacked BCD digit)
            POP PSW               // retrieve PSW (switch back to original register bank)
            POP B                 // retrieve B
            POP ACC               // retrieve A
            RET                   // return to main program
HERE:       SJMP HERE             // end of program; loop forever
```

THINK BOX 7.5

Who do you think should save the registers: a main program (caller) or subroutine (callee) when calling a subroutine (or ISR)?
Both may save the registers. When a caller is saving the registers, it may save the registers which are needed by it or it may save the registers changed by the subroutine, but this requires the knowledge of internal details of subroutine (which is time consuming and difficult). Moreover, this approach will not work with ISRs.

When a subroutine is saving the registers (used by it), it will work for both subroutines and ISRs. In this approach, the subroutine need not have details of the main program and the instructions to save the registers are required to be written only once. Since all registers used in a subroutine are required to be saved, some registers may be unnecessarily saved. Thus, this approach is better and easy.

THINK BOX 7.6

What should be minimum size of the subroutine if it is to be called four times in a program and it is to save the program size?

Assuming that LCALL instruction is used for calling the subroutine. Each time the subroutine is called, 3 bytes are required for calling the subroutine and one byte for return, therefore 13 bytes are occupied if the subroutine is called four times (4x 3 for LCALL + 1 for RET). If the subroutine is of 4 or more bytes (excluding RET), it will occupy 16 bytes (or more) if called 4 times. Therefore minimum size of the subroutine is 4 bytes if is to save the program size for the given condition.

7.4 STACK INITIALIZATION AND OVERFLOW

The reset value of the stack pointer (SP) is 07H, therefore, the first location used for the stack operation is 08H (because SP is automatically incremented by 1 before saving data on the stack). We can use internal RAM addresses 08H to 7FH for the stack, however addresses 08H to 1FH are used by register banks and addresses 20H to 2FH are bit addressable, therefore we should not use this area for the stack (if they are being used by a program). Therefore addresses 30H to 7FH may be used for the stack. The initial portion of this memory area is normally used for storing data, therefore the usual practice is to initialize SP above data area. For example, it may be initialized with address 50H when programmer has reserved the addresses 30H to 50H for storing data.

The 8051 has limited size of the stack, i.e. maximum size of the stack may be 128 bytes (00H to 7FH if these locations are not used for other purpose), therefore make sure that value of SP never exceeds 7FH, otherwise stack will use area reserved for SFRs and overwrite their contents, moreover, all locations between 80H to FFH are not physically present, and data will be lost and program may reach irrecoverable state.

When the value of SP is greater than 7FH, it is referred as *stack overflow* and programmer must make sure that it is always avoided.

THINK BOX 7.7

Internal RAM location with address FFH does not exist physically in the 8051. What will be the contents of A after execution of the instruction MOV A, 0FFH?

Since address FFH does not exist physically, the contents of A are unpredictable.

THINK BOX 7.8

What is the limitation of using only internal RAM as a stack memory in the 8051?

Since internal RAM is only 128 bytes, theoretically maximum size of the stack can only be 128 bytes (practically even less) which is very small.

More care has to be taken by programmer to avoid stack overflow.

THINK BOX 7.9

Can we use SFRs for the stack?

No. Because all the locations between 80H to FFH do not exist physically. Moreover, general data should not be written in the SFRs because it may disturb the operations of the peripheral devices.

7.5 TIME-DELAY GENERATION USING SOFTWARE

Execution of each instruction requires certain number of machine cycles and each machine cycle in the 8051 requires 12 clock cycles (12 oscillator cycles). Time period of a machine cycle depends on frequency of crystal connected to the system (or frequency of external clock signal when on-chip oscillator is not used to generate the clock signal). For example, if 12 MHz crystal is connected to on-chip oscillator, the time period of one clock pulse is 1/12MHz = 0.08333 μs and time period of one machine cycle is 12 × 0.08333 μs = 1 μs. The desired time delay can be generated by wasting the time of microcontroller by executing group of instructions. Before we proceed further to develop the delays using software, let us discuss NOP instruction because it is often used in delay generation.

NOP (No Operation)

NOP is a 1-byte instruction and requires one machine cycle execution time. No operation is performed by this instruction. This instruction only updates the PC to point to the next instruction after execution. It is generally used to waste microcontroller's time in generating time delays using software.

Consider the following instructions:

Instructions	Machine cycles	Execution time (Crystal freq. =12 MHz)
MOV A, R0	1	1 μs
MOV R0, #00H	1	1 μs
NOP	1	1 μs
MOV 10H, 20H	2	2 μs
Total	**5**	**5 μs**

As shown above, the total time required to execute all instructions is 5 μs. Therefore, we can say that these instructions have generated a delay of 5 μs. To generate larger delays, the instructions can be repeated using loops. For example, consider the following instructions:

```
        MOV R0, #0FFH
HERE:   DJNZ, R0, HERE
```

The first instruction requires 1 machine cycle and is executed only once, the second instruction requires 2 machine cycles and it is executed 255 times; therefore, total machine cycles required to complete above two instructions are 1 + (255 × 2) = 511. Assuming crystal frequency equal to 12 MHz, the time required will be 511 μs (0.5 ms approx). To generate even higher delays more instructions can be repeated in a loop or nested loops can be used. It is illustrated in Example 7.14.

Example 7.14

Find the time required to execute (or delay generated by) the following instructions. Assume crystal frequency is 12 MHz.

```
        MOV R0, #249
THERE:  NOP
        NOP
        DJNZ R0, THERE
        NOP
        NOP
        NOP
```

Solution:

The time required by instructions can be calculated as follows.

	Instructions	Execution time (μs)
	MOV R0, #249	1
THERE:	NOP	249 (1 x 249)
	NOP	249 (1 x 249)
	DJNZ R0, THERE	498 (2 x 249)
	NOP	1
	NOP	1
	NOP	1
	Total	1000 μs.

It requires 1000 μs or 1 ms time.

Note that NOP is useful for making adjustments to get exact time delays.

Example 7.15

How much time will be required to execute instructions in Example 7.14 if crystal frequency is 6 MHz?

Solution:

Since crystal frequency is half, it will require double time to execute, i.e. 2000 μs = 2 ms.

Example 7.16

Find time delay generated by the following instructions. Assume crystal frequency is 12 MHz.

```
        MOV R0, # 20
THERE:  MOV R1, # 250
```

```
HERE:  DJNZ R1, HERE
       DJNZ R0, THERE
```

Solution:

It will generate a delay of 1+ 20 x ((1+ (2 x 250)) + 2) =10601 μs.

Example 7.17

Modify the program in Example 7.16 to get an exact delay of 10 ms.

Solution:

```
                MOV R0, # 20
THERE:          MOV R1, # 248
HERE:           DJNZ R1, HERE
                DJNZ R0, THERE
                MOV R0, #09
HERE1:          DJNZ R0, HERE1
```

The delay generated will be exactly 10 ms. Verify using the following expression.

[1+ 20 x ((1+ (2 x 248)) + 2)]+ [1+ (2 x 9)]

Summary of Program-Flow Control Instructions

Unconditional and conditional jumps (byte jumps) are summarized in Tables 7.2 and 7.3 respectively.

Table 7.2 *Unconditional jump instructions with examples*

Mnemonics	Operation	Addressing Modes			
		Direct	Indirect	Register	Immediate
SJMP **rel**	Jump to **rel**	SJMP **rel** SJMP **ABC**			
AJMP **addr11**	Jump to **addr11**	AJMP addr11 AJMP **PQR**			
LJMP **addr16**	Jump to **addr16**	LJMP **addr16** LJMP **AGAIN**			
JMP @A + **DPTR**	Jump to **A+ DPTR**		JMP @**A+ DPTR** JMP @**A+ DPTR**		
ACALL **addr11**	Call subroutine at addr11	ACALL **addr11** ACALL **ROUTINE**			
LCALL **addr16**	Call subroutine at addr16	LCALL **addr16** LCALL **AGAIN**			
RET	Return from subroutine	RET			
RETI	Return from interrupt	RETI			
NOP	No operation	NOP			

Table 7.3 *Conditional jump (byte jumps) instructions with examples*

Mnemonics	Operation	Addressing Modes			
		Direct	Indirect	Register	Immediate
JZ **rel**	Jump if A= 0	Accumulator only			
JNZ **rel**	Jump if A≠ 0	Accumulator only			
DJNZ <BYTE>, **rel**	Decrement & jump if not zero	DJNZ direct, **rel** DJNZ 10, **NOW**		DJNZ Rn,**rel** DJNZ R0,**NEXT**	
CJNE A, **<BYTE>, rel**	Jump if A≠ **<BYTE>**	CJNE A, **direct, rel** CJNE A, **12, ABC**			CJNE A,**#data,rel** CJNE A,**#10H, AB**
CJNE <BYTE>, **#data, rel**	Jump if <BYTE> ≠ #data		CJNE @ Ri,**#data,rel** CJNE @ R1,**#20H, AB**	CJNE Rn,**#data,rel** CJNE R2,**#10H,NEXT**	

POINTS TO REMEMBER

- The program-flow control instructions make the program more flexible and versatile as required by real-world applications.
- A jump instruction changes the content of program counter with a new program address usually referred as destination address. This causes program execution to begin at destination address.
- The unconditional jump does not test any condition and jump is always taken.
- The 8051 support three types of unconditional jumps: short, absolute and long jump. These jumps differ in range over which the jump can be taken.
- The advantage offered by the relative jump is that it allows relocation and requires only one-byte address to be specified as a part of an instruction op-code. The limitation is that destination address must be within +127 to –128 bytes with respect to the PC.
- AJMP and ACALL are the special instructions for which there are eight op-codes.
- All conditional jumps are short relative jumps.
- There is no zero flag in the 8051, the instructions JZ and JNZ checks the Accumulator for the zero.
- The instruction DJNZ decrements first, then checks for zero and it does not affect any of its operands.
- In the 8051, the instructions DJNZ and CJNE are used to repeat the loop for a fixed number of times, while instruction JZ/JNZ and all bit jump instructions are used to repeat the loop until a flag is set to desired state.
- Use of subroutines (modular approach) makes debugging and testing of a program easier.
- PUSH and POP instructions are used to save the registers or memory locations used by a subroutine to perform its internal operations and retrieve the saved contents before returning back to the calling program.
- The number of PUSH and POP instructions in subroutines should be equal and their sequence must be opposite because the stack is last-in first-out memory.
- The desired time delay can be generated by wasting microcontroller time by executing a group of instructions.

OBJECTIVE QUESTIONS

1. The jump instruction with short absolute address occupies _____ bytes in memory.
   (a) 1 byte (b) 2 bytes (c) 3 bytes (d) 4 bytes
2. The jump instruction with long address occupies _____ bytes in memory.
   (a) 1 byte (b) 2 bytes (c) 3 bytes (d) 4 bytes
3. The addressing mode used in the AJMP instruction is,
   (a) absolute addressing (b) long addressing
   (c) indexed addressing (d) relative addressing
4. The call (ACALL/LCALL) instruction stores the return address,
   (a) on the stack (b) in the stack pointer
   (c) in the program counter (d) in the DPTR register
5. Which of the following registers are modified by the jump instructions?
   (a) A (b) DPTR (c) PC (d) all
6. Which of the following registers are modified by the POP E0 instruction?
   (a) SP (b) A (c) PC (d) all
7. A program flow control instruction always modify,
   (a) PC (b) SP (c) DPTR (d) all
8. An instruction is required to provide a backward jump to a location at offset of 100 bytes and with respect to address of jump instruction, it is preferred to use,
   (a) AJMP (b) SJMP (c) LJMP (d) none
9. The destination address for AMP instruction must be,
   (a) within the same page of 2 Kbytes (b) within the same page of 4 Kbytes
   (c) within the same page of 8 Kbytes (d) within the same page of 16 Kbytes

10. The operation performed by JB bit. rel instruction is,
    (a) if bit=1, PC=PC+*rel* (b) if bit=0, PC=PC+*rel*
    (c) if bit=1, PC=PC+2 (d) if bit=0, PC=PC+3
    (PC is pointing to the next instruction)
11. The operation performed by JNC rel instruction is,
    (a) if c=1, PC=PC+*rel* (b) if c=0, PC=PC+*rel*
    (c) if c=1, PC=PC+2 (d) if c=0, PC=PC+2
    (PC is pointing to the next instruction)
12. JZ rel checks the_____ for decision making.
    (a) result of last operation (b) result of last arithmetic function only
    (c) A (d) zero flag
13. JBC bit, rel instruction may affect,
    (a) CY flag (b) OV flag (c) P flag (d) all of the above
14. CJNE A, 00H, NEXT will always modify,
    (a) Accumulator (b) R0 (c) CY (d) none
15. How many times will the following loop be repeated?
    REPEAT: MOV A, #02H
    JNZ REPEAT
    (a) 0 (b) 1 (c) 2 (d) infinite
16. The following program code will read data from port 1 and copy it to port 2, and it will stop looping when bit 7 of port 2 is set
    REPEAT: MOV A, P1
    MOV P2, A
    JB P2.7, REPEAT
    (a) true (b) false
17. Subroutines are usually written in,
    (a) stack memory (b) code memory (c) internal data memory (d) external data memory
18. PUSH instruction,
    (a) increments SP by 1 (b) increments SP by 2 (c) decrement SP by 1 (d) decrement SP by 2
19. ACALL instruction,
    (a) increments SP by 1 (b) increments SP by 2 (c) decrement SP by 1 (d) decrement SP by 2
20. LCALL instruction,
    (a) increments SP by 1 (b) increments SP by 2 (c) increments SP by 3 (d) do not affect SP

Answers to Objective Questions

1. (b)	2. (c)	3. (a)	4. (a)	5. (c)	6. (d)	7. (a)
8. (b)	9. (a)	10. (a), (d)	11. (b), (c)	12. (c)	13. (d)	14 (d)
15. (d)	16. (b)	17. (b)	18. (a)	19. (b)	20. (b)	

REVIEW QUESTIONS WITH ANSWERS

1. What is meant by destination address?

A. It is an address where the program execution will start after execution of jump or call instruction.

2. What is meant by unconditional jump? List the unconditional jump instructions of the 8051.

A. When program execution is altered by instruction without checking any condition, it is referred as unconditional jump, i.e. the unconditional jump does not test any condition and jump is always taken. SJMP, AJMP and LJMP are the unconditional jump instructions of the 8051.

3. Discuss the advantages offered by relative jump instructions.

A. The program written using relative jumps is relocatable, i.e. a program that is written using relative jumps can be placed (loaded) anywhere in the program address space without reassembling. Second advantage is that only 1 byte is required to specify the relative address of the destination location which saves the program bytes and increases the speed of execution.

4. The backward relative addresses are specified in 2's complement in the 8051. True/False.

A. True.

5. What is special about AJMP and ACALL instructions in the 8051 instruction set?

A. They both have 8 different op-codes, while other instructions have only one op-code.

6. Where is JMP @A+DPTR instruction commonly used?

A. It is commonly used to implement the jump tables.

7. How does the 8051 support JZ and JNZ instructions though it has no zero flag?

A. JZ and JNZ instructions check the contents of Accumulator for zero.

8. Which conditional jump instructions are based on the contents of Accumulator?

A. JZ and JNZ

9. What is meant by mnemonic ACALL and LCALL?

A. ACALL: Absolute call (jump can be taken anywhere in the 2K page)
LCALL: Long call (jump can be taken anywhere in the entire program memory of 64K)

10. How many bytes are required by SJMP, AJMP and LJMP instructions?

A. 2 bytes for SJMP and AJMP, 3 bytes for LJMP instruction.

11. What is meant by relative address?

A. Relative address means how far (in terms of bytes) the destination address is, with respect to address where the jump instruction occurs, i.e. with respect to next instruction after the jump instruction (address of the PC).

12. Upon execution, the jump instructions modify the contents of program counter. True/False.

A. True.

13. What is a loop?

A. The looping is a programming technique used to repeat the sequence of instructions several times until certain conditions are met (or to repeat forever without checking any condition)

14. What are the common requirements of the loops?

A. A counter which specifies the number of times a loop should be repeated and pointers, which will point to different data in each iteration.

15. Stack is a LILO (last in last out) memory. True/False.

A. False, it is LIFO (last in first out) memory.

16. All conditional jump instructions are relative jumps. True/False.

A. True.

17. What is the size of the logical page in the 8051? What is the address range of page 3?

A. Page size is 2Kbytes. Address range of page 3 is 1800H to 1FFFH.

18. List the instructions used to call the subroutines.

A. ACALL and LCALL are instructions used to call the subroutines.

19. "JNC HERE" is a _____ byte instruction.

A. 2.

20. What is the status of specified bit after executing JBC instruction?

A. The status of specified bit is 0.

21. "JNZ HERE" instruction, checks value of ___.

A. Accumulator.

22. What is the key difference between POP and RET instructions?

A. POP retrieves one byte from a stack while RET retrieves two bytes.

EXERCISE

1. Compare execution of ACALL and LCALL instructions.
2. What is meant by return address?
3. Which flag is affected after executing JC instruction?
4. List the advantages and disadvantages of using subroutines.
5. What does the mnemonics CJNE and DJNZ stand for?
6. What is meant by conditional jump? List the conditional jump instructions of the 8051.
7. Show with suitable example how relative address is calculated in the SJMP instruction.
8. What is limitation of the SJMP instruction compared to the AJMP instruction?
9. Show with suitable example how absolute address is calculated in the AJMP instruction.
10. What care has to be taken by a programmer when the AJMP instruction occurs at page boundary?
11. Explain how the ACALL and LCALL modify the contents of the program counter.
12. Define the term *subroutine*. How is it useful?
13. Describe the operation of the RET instruction.
14. Discuss the role of the stack in execution of the ACALL and LCALL instructions.
15. How program resumes its operation after completion of the subroutine?
16. Write a program to place value FFH in to internal RAM addresses 10H to 20H using loop.
17. What is meant by nested loop? Explain with a suitable example.
18. How does the microcontroller know where to return to after executing the subroutine?
19. Find the number of times the following loop is repeated.

```
        MOV R2, #100
THERE:  MOV R3, # 50
HERE:   DJNZ R3, HERE
        DJNZ R2, THERE
```

20. Modify the above instructions to repeat loop for 5000 times.
21. Find the number of times the following loop is repeated.

```
THERE:  MOV R2, #100
        MOV R3, # 50
HERE:   DJNZ R3, HERE
        DJNZ R2, THERE
```

22. Mention the address range of all type of jump instructions.
23. Discuss the advantages and disadvantages of the relative addressing.
24. Compare the SJMP and AJMP instructions.
25. Write a subroutine to generate delay of 1 second. Assume crystal frequency is 12 MHz.
26. Modify above subroutine if crystal frequency is changed to 11.0592 MHz.
27. Numbers of PUSH and POP instructions in a subroutine should be equal. Justify.
28. Discuss how CJNE and DJNZ instructions are executed. Where are they commonly used?
29. Write a subroutine which converts BCD number stored in A to equivalent binary number. Store the result into A.
30. Modify the above subroutine if BCD number is stored at location pointed by R0 and store the result at location pointed by R1.
31. Write a subroutine to count the number of 1's in a byte. Main program reads the byte from port 1. Save the result into top of the stack.
32. Illustrate with a suitable example how parameters can be passed to the subroutine using pointers.
33. Illustrate with a suitable example how parameters can be passed to the subroutine using stack.
34. What is meant by stack overflow? Discuss the consequences of it and how it can be avoided?
35. What is meant by context saving and retrieving?