CHAPTER 3

JUMP, LOOP, AND CALL INSTRUCTIONS

OBJECTIVES

Upon completion of this chapter, you will be able to:

- **Code 8051 Assembly language instructions using loops**
- **Code 8051 Assembly language conditional jump instructions**
- **Explain conditions that determine each conditional jump instruction**
- **Code long jump instructions for unconditional jumps**
- **Code short jump instructions for unconditional short jumps**
- **Calculate target addresses for jump instructions**
- **Code 8051 subroutines**
- **Describe precautions in using the stack in subroutines**
- **Discuss crystal frequency versus machine cycle**
- **Code 8051 programs to generate a time delay**

In the sequence of instructions to be executed, it is often necessary to transfer program control to a different location. There are many instructions in the 8051 to achieve this. This chapter covers the control transfer instructions available in 8051 Assembly language. In the first section we discuss instructions used for looping, as well as instructions for conditional and unconditional jumps. In the second section we examine CALL instructions and their uses. In the third section, time delay subroutines are described.

SECTION 3.1: LOOP AND JUMP INSTRUCTIONS

In this section we first discuss how to perform a looping action in the 8051 and then talk about jump instructions, both conditional and unconditional.

Looping in the 8051

Repeating a sequence of instructions a certain number of times is called a *loop*. The loop is one of most widely used actions that any microprocessor performs. In the 8051, the loop action is performed by the instruction "`DJNZ reg, label`". In this instruction, the register is decremented; if it is not zero, it jumps to the target address referred to by the label. Prior to the start of the loop the register is loaded with the counter for the number of repetitions. Notice that in this instruction both the register decrement and the decision to jump are combined into a single instruction.

Example 3-1

Write a program to
(a) clear ACC, then
(b) add 3 to the accumulator ten times.

Solution:

```
;This program adds value 3 to the ACC ten times

          MOV   A,#0       ;A=0, clear ACC
          MOV   R2,#10     ;load counter R2=10
AGAIN:    ADD   A,#03      ;add 03 to ACC
          DJNZ  R2,AGAIN   ;repeat until R2=0(10 times)
          MOV   R5,A       ;save A in R5
```

In the program in Example 3-1, the R2 register is used as a counter. The counter is first set to 10. In each iteration the instruction DJNZ decrements R2 and checks its value. If R2 is not zero, it jumps to the target address associated with label "AGAIN". This looping action continues until R2 becomes zero. After R2 becomes zero, it falls through the loop and executes the instruction immediately below it, in this case the "`MOV R5,A`" instruction.

Notice in the DJNZ instruction that the registers can be any of R0 - R7. The counter can also be a RAM location as we will see in Chapter 5.

Example 3-2

What is the maximum number of times that the loop in Example 3-1 can be repeated?

Solution:

Since R2 holds the count and R2 is an 8-bit register, it can hold a maximum of FFH (255 decimal); therefore, the loop can be repeated a maximum of 256 times.

Loop inside a loop

As shown in Example 3-2, the maximum count is 256. What happens if we want to repeat an action more times than 256? To do that, we use a loop inside a loop, which is called a *nested loop*. In a nested loop, we use two registers to hold the count. See Example 3-3.

Example 3-3

Write a program to (a) load the accumulator with the value 55H, and (b) complement the ACC 700 times.

Solution:

Since 700 is larger than 255 (the maximum capacity of any register), we use two registers to hold the count. The following code shows how to use R2 and R3 for the count.

```
        MOV  A,#55H     ;A=55H
        MOV  R3,#10     ;R3=10, the outer loop count
NEXT:   MOV  R2,#70     ;R2=70, the inner loop count
AGAIN:  CPL  A          ;complement A register
        DJNZ R2,AGAIN   ;repeat it 70 times (inner loop)
        DJNZ R3,NEXT
```

In this program, R2 is used to keep the inner loop count. In the instruction "DJNZ R2,ACAIN", whenever R2 becomes 0 it falls through and "DJNZ R3,NEXT" is executed. This instruction forces the CPU to load R2 with the count 70 and the inner loop starts again. This process will continue until R3 becomes zero and the outer loop is finished.

Other conditional jumps

Conditional jumps for the 8051 are summarized in Table 3-1. More details of each instruction are provided in Appendix A. In Table 3-1, notice that some of the instructions, such as JZ (jump if A = zero) and JC (jump if carry), jump only if a certain condition is met. Next we examine some conditional jump instructions with examples.

JZ (jump if A = 0)

In this instruction the content of register A is checked. If it is zero, it jumps to the target address. For example, look at the following code.

```
        MOV   A,R0          ;A=R0
        JZ    OVER          ;jump if A = 0
        MOV   A,R1          ;A=R1
        JZ .  OVER          ;jump if A = 0
        ...
OVER:
```

In this program, if either R0 or R1 is zero, it jumps to the label OVER. Notice that the JZ instruction can be used only for register A. It can only check to see whether the accumulator is zero, and it does not apply to any other register. More importantly, you don't have to perform an arithmetic instruction such as decrement to use the JNZ instruction. See Example 3-4.

Table 3-1: 8051 Conditional Jump Instructions

Instruction	Action
JZ	Jump if A = 0
JNZ	Jump if A ≠ 0
DJNZ	Decrement and jump if A ≠ 0
CJNE A,byte	Jump if A ≠ byte
CJNE reg,#data	Jump if byte ≠ #data
JC	Jump if CY = 1
JNC	Jump if CY = 0
JB	Jump if bit = 1
JNB	Jump if bit = 0
JBC	Jump if bit = 1 and clear bit

Example 3-4

Write a program to determine if R5 contains the value 0. If so, put 55H in it.

Solution:

```
        MOV   A,R5        ;copy R5 to A
        JNZ   NEXT        ;jump if A is not zero
        MOV R5,#55H
NEXT:   ...
```

JNC (jump if no carry, jumps if CY = 0)

In this instruction, the carry flag bit in the flag (PSW) register is used to make the decision whether to jump. In executing "`JNC label`", the processor looks at the carry flag to see if it is raised (CY = 1). If it is not, the CPU starts to fetch and execute instructions from the address of the label. If CY = 1, it will not jump but will execute the next instruction below JNC.

It needs to be noted that there is also a "`JC label`" instruction. In the JC instruction, if CY = 1 it jumps to the target address. We will give more examples of these instructions in the context of applications in future chapters.

There is also a JB (jump if bit is high) and JNB (jump if bit is low). These are discussed in Chapters 4 and 8 when bit manipulation instructions are discussed.

Example 3-5

Find the sum of the values 79H, F5H, and E2H. Put the sum in registers R0 (low byte) and R5 (high byte).

Solution:

```
        MOV   A,#0        ;clear A(A=0)
        MOV   R5,A        ;clear R5
        ADD   A,#79H      ;A=0+79H=79H
        JNC   N_1         ;if no carry, add next number
        INC   R5          ;if CY=1, increment R5
N_1:    ADD   A,#0F5H     ;A=79+F5=6E and CY=1
        JNC   N_2         ;jump if CY=0
        INC   R5          ;If CY=1 then increment R5(R5=1)
N_2:    ADD   A,#0E2H     ;A=6E+E2=50 and CY=1
        JNC   OVER        ;jump if CY=0
        INC   R5          ;if CY=1, increment 5
OVER:MOV  R0,A            ;Now R0=50H, and R5=02
```

All conditional jumps are short jumps

It must be noted that all conditional jumps are short jumps, meaning that the address of the target must be within −128 to +127 bytes of the contents of the program counter (PC). This very important concept is discussed at the end of this section.

Unconditional jump instructions

The unconditional jump is a jump in which control is transferred unconditionally to the target location. In the 8051 there are two unconditional jumps: LJMP (long jump) and SJMP (short jump). Each is discussed below.

LJMP (long jump)

LJMP is an unconditional long jump. It is a 3-byte instruction in which the first byte is the opcode, and the second and third bytes represent the 16-bit address of the target location. The 2-byte target address allows a jump to any memory location from 0000 to FFFFH.

Remember that although the program counter in the 8051 is 16-bit, thereby giving a ROM address space of 64K bytes, not all 8051 family members have that much on-chip program ROM. The original 8051 had only 4K bytes of on-chip ROM for program space; consequently, every byte was precious. For this reason there is also a SJMP (short jump) instruction which is a 2-byte instruction as opposed to the 3-byte LJMP instruction. This can save some bytes of memory in many applications where memory space is in short supply. SJMP is discussed next.

SJMP (short jump)

In this 2-byte instruction, the first byte is the opcode and the second byte is the relative address of the target location. The relative address range of 00 - FFH

is divided into forward and backward jumps; that is, within -128 to +127 bytes of memory relative to the address of the current PC (program counter). If the jump is forward, the target address can be within a space of 127 bytes from the current PC. If the target address is backward, the target address can be within -128 bytes from the current PC. This is explained in detail next.

Calculating the short jump address

In addition to the SJMP instruction, all conditional jumps such as JNC, JZ, and DJNZ are also short jumps due to the fact that they are all two-byte instructions. In these instructions the first byte is the opcode and the second byte is the relative address. The target address is relative to the value of the program counter. To calculate the target address, the second byte is added to the PC of the instruction immediately below the jump. To understand this, look at Example 3-6.

Example 3-6

Using the following list file, verify the jump forward address calculation.

```
Line  PC    Opcode          Mnemonic  Operand
01    0000                  ORG       0000
02    0000  7800            MOV       R0,#0
03    0002  7455            MOV       A,#55H
04    0004  6003            JZ        NEXT
05    0006  08              INC       R0
06    0007  04      AGAIN:  INC       A
07    0008  04              INC       A
08    0009  2477    NEXT:   ADD       A,#77h
09    000B  5005            JNC       OVER
10    000D  E4              CLR       A
11    000E  F8              MOV       R0,A
12    000F  F9              MOV       R1,A
13    0010  FA              MOV       R2,A
14    0011  FB              MOV       R3,A
15    0012  2B      OVER:   ADD       A,R3
16    0013  50F2            JNC       AGAIN
17    0015  80FE    HERE:   SJMP      HERE
18    0017                  END
```

Solution:

First notice that the JZ and JNC instructions both jump forward. The target address a forward jump is calculated by adding the PC of the following instruction to the s ond byte of the short jump instruction, which is called the relative address. In line 4 instruction "JZ NEXT" has opcode of 60 and operand of 03 at the addresses of 0004 a 0005. The 03 is the relative address, relative to the address of the next instruction I R0, which is 0006. By adding 0006 to 3, the target address of the label NEXT, whic 0009, is generated. In the same way for line 9, the "JNC OVER" instruction has opc and operand of 50 and 05 where 50 is the opcode and 05 the relative address. Therefo 05 is added to 000D, the address of instruction "CLR A", giving 12H, the address label OVER.

Example 3-7

Verify the calculation of backward jumps in Example 3-6.

Solution:

In that program list, "JNC AGAIN" has opcode 50 and relative address F2H. When the relative address of F2H is added to 15H, the address of the instruction below the jump, we have 15H + F2H = 07 (the carry is dropped). Notice that 07 is the address of label AGAIN. Look also at "SJMP HERE", which has 80 and FE for the opcode and relative address, respectively. The PC of the following instruction, 0017H, is added to FEH, the relative address, to get 0015H, address of the HERE label (17H + FEH = 15H). Notice that FEH is -2 and 17H + (-2) = 15H. For further discussion of the addition of negative numbers, see Chapter 6.

Jump backward target address calculation

While in the case of a forward jump, the displacement value is a positive number between 0 to 127 (00 to 7F in hex), for the backward jump the displacement is a negative value of 0 to -128 as explained in Example 3-7.

It must be emphasized that regardless of whether the SJMP is a forward or backward jump, for any short jump the address of the target address can never be more than -128 to +127 bytes from the address associated with the instruction below the SJMP. If any attempt is made to violate this rule, the assembler will generate an error stating the jump is out of range.

Review Questions

1. The mnemonic DJNZ stands for _____.
2. True or false. "DJNZ R5, BACK" combines a decrement and a jump in a single instruction.
3. "JNC HERE" is a 2 -byte instruction.
4. In "JZ NEXT", which register's content is checked to see if it is zero? (A)
5. LJMP is a 3 -byte instruction.

SECTION 3.2: CALL INSTRUCTIONS

Another control transfer instruction is the CALL instruction, which is used to call a subroutine. Subroutines are often used to perform tasks that need to be performed frequently. This makes a program more structured in addition to saving memory space. In the 8051 there are two instructions for call: LCALL (long call) and ACALL (absolute call). Deciding which one to use depends on the target address. Each instruction is explained next.

LCALL (long call)

In this 3-byte instruction, the first byte is the opcode and the second and third bytes are used for the address of the target subroutine. Therefore, LCALL can be used to call subroutines located anywhere within the 64K byte address space of

the 8051. To make sure that after execution of the called subroutine the 8051 knows where to come back to, it automatically saves on the stack the address of the instruction immediately below the LCALL. When a subroutine is called, control is transferred to that subroutine, and the processor saves the PC (program counter) on the stack and begins to fetch instructions from the new location. After finishing execution of the subroutine, the instruction RET (return) transfers control back to the caller. Every subroutine needs RET as the last instruction. See Example 3-8.

The following points should be noted for the program in Example 3-8.

1. Notice the DELAY subroutine. Upon executing the first "LCALL DELAY", the address of the instruction right below it, "MOV A, #0AAH", is pushed onto the stack, and the 8051 starts to execute instructions at address 300H.
2. In the DELAY subroutine, first the counter R5 is set to 255 (R5 = FFH); therefore, the loop is repeated 256 times. When R5 becomes 0, control falls to the RET instruction which pops the address from the stack into the program counter and resumes executing the instructions after the CALL.

Example 3-8

Write a program to toggle all the bits of port 1 by sending to it the values 55H and AAH continuously. Put a time delay in between each issuing of data to port 1. This program will be used to test the ports of the 8051 in the next chapter.

Solution:

```
        ORG     0
BACK:   MOV     A,#55H      ;load A with 55H
        MOV     P1,A        ;send 55H to port 1
        LCALL   DELAY       ;time delay
        MOV     A,#0AAH     ;load A with AA (in hex)
        MOV     P1,A        ;send AAH to port 1
        LCALL   DELAY
        SJMP    BACK        ;keep doing this indefinitely
;-------this is the delay subroutine
        ORG     300H        ;put time delay at address 300H
DELAY:  MOV     R5,#0FFH    ;R5=255(FF in hex),the counter
AGAIN:  DJNZ    R5,AGAIN    ;stay here until R5 becomes 0
        RET                 ;return to caller (when R5 = 0)
        END                 ;end of asm file
```

The amount of time delay in Example 3-8 depends on the frequency of the 8051. How to calculate the exact time will be explained in detail in Chapter 4. However you can increase the time delay by using a nested loop as shown below.

```
DELAY:                      ;nested loop delay
        MOV   R4,#255       ;R4=255(FF in hex)
NEXT:   MOV   R5,#255       ;R5=255(FF in hex)
AGAIN:  DJNZ  R5,AGAIN      ;stay here until R5 becomes 0
        DJNZ  R4,NEXT       ;decrement R4
                            ;keep loading R5 until R4=0
        RET                 ;return (when R4 = 0)
```

CALL instruction and the role of the stack

The stack and stack pointer were covered in the last chapter. To understand the importance of the stack in microcontrollers, we now examine the contents of the stack and stack pointer for Example 3-8. This is shown in Example 3-9.

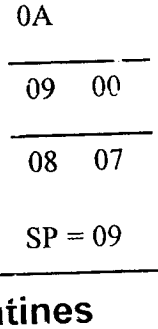
Example 3-9

Analyze the stack contents after the execution of the first LCALL in the following.

Solution:

```
001   0000                      ORG   0
002   0000 7455   BACK:         MOV   A,#55H   ;load A with 55H
003   0002 F590                 MOV   P1,A     ;send 55H to port 1
004   0004 120300               LCALL DELAY    ;time delay
005   0007 74AA                 MOV    A,#0AAH;load A with AAH
006   0009 F590                 MOV   P1,A     ;send AAH to port 1
007   000B 120300               LCALL DELAY
008   000E 80F0                 SJMP  BACK      ;keep doing this
009   0010
010   0010 ;----------this is the delay subroutine
011   0300                      ORG 300H
012   0300          DELAY:
013   0300 7DFF                 MOV   R5,#0FFH  ;R5=255
014   0302 DDFE   AGAIN:        DJNZ  R5,AGAIN  ;stay here
015   0304 22                   RET             ;return to caller
016   0305                      END             ;end of asm file
```

When the first LCALL is executed, the address of the instruction "MOV A,#0AAH" is saved on the stack. Notice that the low byte goes first and the high byte is last. The last instruction of the called subroutine must be a RET instruction which directs the CPU to POP the top bytes of the stack into the PC and resume executing at address 07. The diagram shows the stack frame after the first LCALL.



Use of PUSH and POP instructions in subroutines

Upon calling a subroutine, the stack keeps track of where the CPU should return after completing the subroutine. For this reason, we must be very careful in any manipulation of stack contents. The rule is that the number of PUSH and POP instructions must always match in any called subroutine. In other words, for every PUSH there must be a POP. See Example 3-10.

Calling subroutines

In Assembly language programming it is common to have one main program and many subroutines that are called from the main program. This allows you to make each subroutine into a separate module. Each module can be tested separately and then brought together with the main program. More importantly, in a large program the modules can be assigned to different programmers in order to shorten development time.

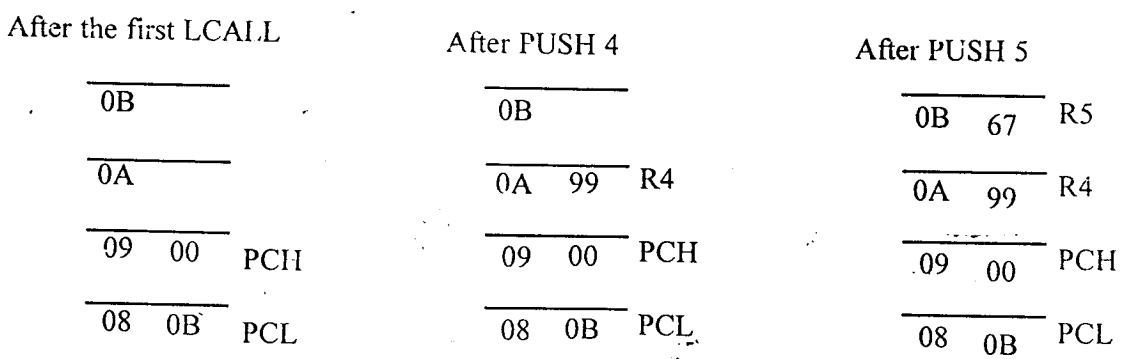
Example 3-10

Analyze the stack for the first LCALL instruction in the following program.

```
01 0000                ORG   0
02 0000 7455 BACK:     MOV   A,#55H      ;load A with 55H
03 0002 F590           MOV   P1,A        ;send 55H to port 1
04 0004 7C99           MOV   R4,#99H
05 0006 7D67           MOV   R5,#67H
06 0008 120300         LCALL DELAY       ;time delay
07 000B 74AA           MOV   A,#0AAH     ;Load A with AA
08 000D F590           MOV   P1,A        ;send AAH to port 1
09 000F 120300         LCALL DELAY
10 0012 80EC           SJMP  BACK        ;keep doing this
11 0014               ;--------this is the delay subroutine
12 0300                ORG 300H
13 0300 C004   DELAY:PUSH 4                ;PUSH R4
14 0302 C005           PUSH 5              ;PUSH R5
15 0304 7CFF           MOV  R4,#0FFH       ;R4=FFH
16 0306 7DFF   NEXT:   MOV  R5,#0FFH       ;R5=255
17 0308 DDFE   AGAIN:DJNZ R5,AGAIN
18 030A DCFA           DJNZ R4,NEXT
19 030C D005           POP  5              ;POP INTO R5
20 030E D004           POP  4              ;POP INTO R4
21 0310 22             RET                 ;return to caller
22 0311                END                 ;end of asm file
```

Solution:

First notice that for the PUSH and POP instructions we must specify the direct address of the register being pushed or popped. Here is the stack frame.



It needs to be emphasized that in using LCALL, the target address of the subroutine can be anywhere within the 64K bytes memory space of the 8051. This is not the case for the other call instruction, ACALL, which is explained next.

```
;MAIN program calling subroutines
          ORG  0
MAIN:     LCALL     SUBR_1
          LCALL     SUBR_2
          LCALL     SUBR_3

HERE:     SJMP      HERE
;---------end of MAIN
;
SUBR_1:   ....
          ....
          RET
;---------end of subroutine 1
;
SUBR_2:   ....
          ....
          RET
;---------end of subroutine 2

SUBR_3:   ....
          ....
          RET
;---------end of subroutine 3
          END       ;end of the asm file
```

Figure 3-1. 8051 Assembly Main Program That Calls Subroutines

ACALL (absolute call)

ACALL is a 2-byte instruction in contrast to LCALL, which is 3 bytes. Since ACALL is a 2-byte instruction, the target address of the subroutine must be within 2K bytes address because only 11 bits of the 2 bytes are used for the address. There is no difference between ACALL and LCALL in terms of saving the program counter on the stack or the function of the RET instruction. The only difference is that the target address for LCALL can be anywhere within the 64K byte address space of the 8051 while the target address of ACALL must be within a 2K-byte range. In many variations of the 8051 marketed by different companies, on-chip ROM is as low as 1K bytes. In such cases, the use of ACALL instead of LCALL can save a number of bytes of program ROM space.

Example 3-11

A developer is using the Atmel AT89C1051 microcontroller chip for a product. This chip has only 1K bytes of on-chip flash ROM. Which of the instructions LCALL and ACALL is most useful in programming this chip?

Solution:
The ACALL instruction is more useful since it is a 2-byte instruction. It saves one byte each time the call instruction is used.

Of course in addition to using compact instructions, we can program effeciently by having a detailed knowledge of all the instructions supported by a given microprocessor, and using them wisely. Look at Example 3-12.

Example 3-12

Rewrite Example 3-8 as efficiently as you can.

Solution:

```
          ORG   0
          MOV   A,#55H      ;load A with 55H
BACK:     MOV   P1,A        ;issue value in reg A to port 1
          ACALL DELAY       ;time delay
          CPL   A           ;complement reg A
          SJMP  BACK        ;keep doing this indefinitely

;----------this is the delay subroutine
DELAY:
          MOV   R5,#0FFH    ;R5=255(FF in hex),the counter
AGAIN:    DJNZ  R5,AGAIN    ;stay here until R5 becomes 0
          RET               ;return to caller
          END               ;end of asm file
```

Notice in this program that register A is set to 55H. By complementing 55H, we hav AAH; and by complementing AAH we have 55H. Why? "01010101" in binary (55H becomes "10101010" in binary (AAH) when it is complemented; and "10101010 becomes "01010101" if it is complemented.

Review Questions

1. What do the mnemonics "LCALL" and "ACALL" stand for?
2. True or false. In the 8051, control can be transferred anywhere within the 64K bytes of code space if using the LCALL instruction.
3. How does the CPU know where to return to after executing the RET instruction?
4. Describe briefly the function of the RET instruction.
5. The LCALL instruction is a ___-byte instruction.

SECTION 3.3: TIME DELAY GENERATION AND CALCULATION

In the last section we used the DELAY subroutine. How to generate various time delays and calculate exact delays is discussed in this section.

Machine cycle

For the CPU to execute an instruction takes a certain number of clock cycles. In the 8051 family, these clock cycles are referred to as *machine cycles*. Appendix A.2 provides the list of 8051 instructions and their machine cycles. To calculate a time delay, we use this list. In the 8051 family, the length of the machine cycle depends on the frequency of the crystal oscillator connected to the

8051 system. The crystal oscillator, along with on-chip circuitry, provide the clock source for the 8051 CPU (see Chapter 4). The frequency of the crystal connected to the 8051 family can vary from 4 MHz to 30 MHz, depending on the chip rating and manufacturer. Very often the 11.0592 MHz crystal oscillator is used to make the 8051-based system compatible with the serial port of the IBM PC (see Chapter 10). In the 8051, one machine cycle lasts 12 oscillator periods. Therefore, to calculate the machine cycle, we take 1/12 of the crystal frequency, then take its inverse, as shown in Example 3-13.

Example 3-13

The following shows crystal frequency for three different 8051-based systems. Find the period of the machine cycle in each case.
(a) 11.0592 MHz (b) 16 MHz (c) 20 MHz

Solution:

(a)11.0592/12 = 921.6 kHz; machine cycle is 1/921.6 kHz = 1.085 μs (microsecond)
(b) 16 MHz/12 = 1.333 MHz; machine cycle (MC) = 1/.1.333 MHz = 0.75 μs
(c) 20 MHz/12 = 1.66 MHz; MC = 1/1.66 MHz = 0.60 μs

Example 3-14

For an 8051 system of 11.0592 MHz, find how long it takes to execute each of the following instructions.

(a) `MOV R3,#55` (b) `DEC R3` (c) `DJNZ R2,target`
(d) `LJMP` (e) `SJMP` (f) `NOP` (no operation)
(g) `MUL AB`

Solution:

The machine cycle for a system of 11.0592 MHz is 1.085 μs as shown in Example 3-13. Table A-1 in Appendix A shows machine cycles for each of the above instructions. Therefore, we have:

Instruction	*Machine cycles*	*Time to execute*
(a) `MOV R3,#55`	1	1×1.085 μs = 1.085 μs
(b) `DEC R3`	1	1×1.085 μs = 1.085 μs
(c) `DJNZ R2,target`	2	2×1.085 μs = 2.17 μs
(d) `LJMP`	2	2×1.085 μs = 2.17 μs
(e) `SJMP`	2	2×1.085 μs = 2.17 μs
(f) `NOP`	1	1×1.085 μs = 1.085 μs
(g) `MUL AB`	4	4×1.085 μs = 4.34 μs

Delay calculation

As seen in the last section, a delay subroutine consists of two parts: (1) setting a counter, and (2) a loop. Most of the time delay is performed by the body of the loop, as shown in Example 3-15.

Example 3-15

Find the size of the delay in the following program, if the crystal frequency is 11.0592 MHz.

```
            MOV  A,#55H
AGAIN:      MOV  P1,A
            ACALL DELAY
            CPL  A
            SJMP AGAIN
;----Time delay
DELAY:      MOV  R3,#200
HERE:       DJNZ R3,HERE
            RET
```

Solution:

From Table A-1 in Appendix A, we have the following machine cycles for each instruction of the DELAY subroutine.

```
                                Machine Cycle
DELAY:      MOV  R3,#200            1
HERE:       DJNZ R3,HERE            2
            RET                     1
```

Therefore, we have a time delay of $[(200 \times 2) + 1 + 1] \times 1.085\ \mu s = 436.17\ \mu s$.

Very often we calculate the time delay based on the instructions inside the loop and ignore the clock cycles associated with the instructions outside the loop.

In Example 3-15, the largest value the R3 register can take is 255; therefore, one way to increase the delay is to use NOP instructions in the loop. NOP, which stands for "no operation," simply wastes time. This is shown in Example 3-16.

Loop inside loop delay

Another way to get a large delay is to use a loop inside a loop, which is also called a *nested loop*. See Example 3-17.

Example 3-16

Find the time delay for the following subroutine, assuming a crystal frequency of 11.0592 MHz.

```
                                  Machine Cycle
DELAY:    MOV  R3,#250              1

HERE:     NOP                       1
          NOP                       1
          NOP                       1
          NOP                       1
          DJNZ R3,HERE              2

          RET                       1
```

Solution:

The time delay inside the HERE loop is [250 (1+1+1+1+2)] × 1.085 μs = 1500 × 1.085 μs = 1627.5 μs. Adding the two instructions outside the loop we have 1627.5 μs + 2 × 1.085 μ s = 1629.67 μs.

Example 3-17

For a machine cycle of 1.085 μs, find the time delay in the following subroutine.

```
DELAY:                            Machine Cycle
          MOV  R2,#200              1
AGAIN:    MOV  R3,#250              1
HERE:     NOP                       1
          NOP                       1
          DJNZ R3,HERE              2
          DJNZ R2,AGAIN             2
          RET                       1
```

For the HERE loop, we have (4 × 250) 1.085 μs = 1085 μs. The AGAIN loop repeats the HERE loop 200 times; therefore, we have 200 × 1085 μs = 217000, if we do not include the overhead. However, the instructions "MOV R3,#250" and "DJNZ R2,AGAIN" at the beginning and end of the AGAIN loop add (3 × 200 × 1.085 μs) = 651 μs to the time delay. As a result we have 217000 + 651 = 217651 μs = 217.651 milliseconds for total time delay associated with the above DELAY subroutine. Notice that in the case of a nested loop, as in all other time delay loops, the time is approximate since we have ignored the first and last instructions in the subroutine.

Review Questions

1. True or false. In the 8051, the machine cycle lasts 12 clock cycles of the crystal frequency.
2. The minimum number of machine cycles needed to execute an 8051 instruction is _____.
3. For Question 2, what is the maximum number of cycles needed, and for which instructions?
4. Find the machine cycle for a crystal frequency of 12 MHz.
5. Assuming a crystal frequency of 12 MHz, find the time delay associated with the loop section of the following DELAY subroutine.

```
DELAY:
            MOV   R3,#100
HERE:       NOP
            NOP
            NOP
            DJNZ  R3,HERE
            RET
```

SUMMARY

The flow of a program proceeds sequentially, from instruction to instruction, unless a control transfer instruction is executed. The various types of control transfer instructions in Assembly language include conditional and unconditional jumps, and call instructions.

The looping action in 8051 Assembly language is performed using a special instruction which decrements a counter and jumps to the top of the loop if the counter is not zero. Other jump instructions jump conditionally, based on the value of the carry flag, the accumulator, or bits of the I/O port. Unconditional jumps can be long or short, depending on the relative value of the target address. Special attention must be given to the effect of LCALL and ACALL instructions on the stack.

PROBLEMS

SECTION 3.1: LOOP AND JUMP INSTRUCTIONS

1. In the 8051, looping action with instruction "`DJNZ Rx,rel address`" is limited to ____ iterations.
2. If a conditional jump is not taken, what is the next instruction to be executed?
3. In calculating the target address for a jump, a displacement is added to the contents of register _____.
4. The mnemonic SJMP stands for _____ and it is a ___-byte instruction.
5. The mnemonic LJMP stands for _____ and it is a ___-byte instruction.
6. What is the advantage of using SJMP over LJMP?
7. True or false. The target of a short jump is within -128 to +127 bytes of the current PC.
8. True or false. All 8051 jumps are short jumps.

9. Which of the following instructions is (are) not a short jump?
   (a) JZ (b) JNC (c) LJMP (d) DJNZ
10. A short jump is a ___-byte instruction. Why?
11. True or false. All conditional jumps are short jumps.
12. Show code for a nested loop to perform an action 1000 times.
13. Show code for a nested loop to perform an action 100,000 times.
14. Find the number of times the following loop is performed.

```
        MOV  R6,#200
BACK:   MOV  R5,#100
HERE:   DJNZ R5,HERE
        DJNZ R6,BACK
```

15. The target address of a jump backward is a maximum of _____ bytes from the current PC.
16. The target address of a jump forward is a maximum of _____ bytes from the current PC.

SECTION 3.2: CALL INSTRUCTIONS

17. LCALL is a ___-byte instruction.
18. ACALL is a ___-byte instruction.
19. The ACALL target address is limited to ____ bytes from the present PC.
20. The LCALL target address is limited to ____ bytes from the present PC.
21. When LCALL is executed, how many bytes of the stack are used?
22. When ACALL is executed, how many bytes of the stack are used?
23. Why do the number of PUSH and POP instructions in a subroutine need to be equal?
24. Describe the action associated with the POP instruction.
25. Show the stack for the following code.

```
000B 120300          LCALL DELAY
000E 80F0            SJMP BACK     ;keep doing this
0010
0010 ;---------this is the delay subroutine
0300                 ORG 300H
0300       DELAY:
0300 7DFF            MOV  R5,#0FFH  ;R5=255
0302 DDFE  AGAIN:    DJNZ R5,AGAIN  ;stay here
0304 22              RET            ;return
```

26. Reassemble Example 3-10 at ORG 200 (instead of ORG 0) and show the stack frame for the first LCALL instruction.

SECTION 3.3:TIME DELAY GENERATION AND CALCULATION

27. Find the system frequency if the machine cycle = 1.2 μs.
28. Find the machine cycle if crystal frequency is 18 MHz.
29. Find the machine cycle if crystal frequency is 12 MHz.
30. Find the machine cycle if crystal frequency is 25 MHz.

31. True or false. LJMP and SJMP instructions take the same amount of time to execute even though one is a 3-byte instruction and the other one is a 2-byte instruction.

32. Find the time delay for the delay subroutine shown to the right, if the system frequency is 11.0592 MHz.

```
DELAY:    MOV  R3,#150
HERE:     NOP
          NOP
          NOP
          DJNZ R3,HERE
          RET
```

33. Find the time delay for the delay subroutine shown to the right, if the system frequency is 16 MHz.

```
DELAY:    MOV  R3,#200
HERE:     NOP
          NOP
          NOP
          DJNZ R3,HERE
          RET
```

34. Find the time delay for the delay subroutine shown to the right, if the system frequency is 11.0592 MHz.

```
DELAY:    MOV  R5,#100
BACK:     MOV  R2,#200
AGAIN:    MOV  R3,#250
HERE:     NOP
          NOP
          DJNZ R3,HERE
          DJNZ R2,AGAIN
          DJNZ R5,BACK
          RET
```

35. Find the time delay for the delay subroutine shown to the right, if the system frequency is 16 MHz.

```
DELAY:    MOV  R2,#150
AGAIN:    MOV  R3,#250
HERE:     NOP
          NOP
          NOP
          DJNZ R3,HERE
          DJNZ R2,AGAIN
          RET
```

ANSWERS TO REVIEW QUESTIONS

SECTION 3.1: LOOP AND JUMP INSTRUCTIONS

1. Decrement and jump if not zero 2. True 3. 2 4. A 5. 3

SECTION 3.2: CALL INSTRUCTIONS

1. Long CALL and Absolute CALL 2. True
3. The address of where to return is in the stack.
4. Upon executing the RET instruction, the CPU pops off the top two bytes of the stack into the program counter (PC) register and starts to execute from this new location.
5. 3

SECTION 3.3: TIME DELAY GENERATION AND CALCULATION

1. True 2. 1 3. MUL and DIV each take 4 machine cycles.
4. 12 MHz / 12 = 1 MHz, and MC = 1/1 MHz = 1 µs.
5. [100 (1+1+1+2)] × 1 µs = 500 µs = 0.5 milliseconds.