Instructor's Manual and PowerPoints

to accompany

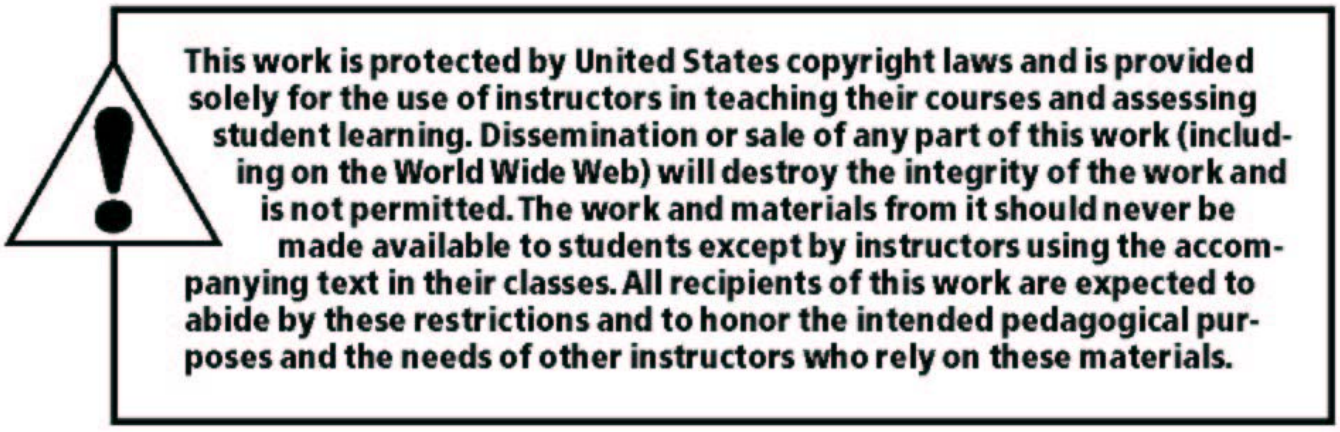
THE 8051 MICROCONTROLLER

Fourth Edition

I. Scott MacKenzie

Raphael C.-W. Phan

Upper Saddle River, New Jersey
Columbus, Ohio



---

10 9 8 7 6 5 4 3 2 1

PEARSON Prentice Hall

ISBN 0-13-060386-4

THE 8051 MICROCONTROLLER

Fourth Edition

Instructor's Manual

This manual contains solutions to the problems at the end of the chapters in The 8051 Microcontroller (4th edition). Additional materials are provided that should prove useful for instructors delivering a lecture + lab course on the 8051 microcontroller. These include the following:

- Discussions on the solutions
- Laboratory project suggestions

Discussions on solutions are provided to assist instructors in discussing with students the solutions to problems.

Laboratory project suggestions are provided with selected problems which lend themselves to further exploration in a laboratory setting. Instructors may wish to distribute the initial solution (given in this manual) to assist students in getting started. Extensions to the basic problem are given in the form of defined tasks to be solved using software and/or hardware. The tasks are defined in a manner that facilitates demonstration in the laboratory.

Courses based on the 8051 will require a single-board computer for laboratory projects. Although many 8051 SBCs are available (sources are provided in Appendix H in the text), the SBC-51 described in Chapter 10 is a logical choice for the initial laboratory project. A section is provided at the end of this manual to facilitate construction and testing of the SBC-51.

This solutions manual has been updated with a total of 72 new questions have been added, for a total of 200.

I. Scott MacKenzie
Raphael C.-W. Phan
August 2003

Chapter 1 - Introduction to Microcontrollers

1. (a) The first widely used microprocessor was the 8080.
   (b) The 8080 was introduced in 1970 by Intel Corp.

2. MOS Technology was responsible for the 6502 microprocessor, Zilog for the Z80.

3. (a) The 8051 was introduced in 1980.
   (b) The predecessor of the 8051 was the 8048, introduced in 1976.

4. (a) RAM (random access memory) and ROM (read-only memory).
   (b) ROM retains its contents even when powered-off.
   (c) The term "non-volatile" describes this property of ROM.

5. (a) The program counter
   (b) The program counter contains the address of the next instruction to be executed.

6. (a) The address bus contains the content of the program counter. The data bus contains the opcode of the instruction.
   (b) The information on the address bus is output, originating from the CPU. The information on the data bus is input, originating from the RAM.

7. $2^{18} = 2^8 \times 2^{10} = 256$K bytes.

8. The phrase "16-bit computer" refers to a computer system with 16 lines on its data bus.

9. Online storage is directly accessible through software, whereas archival storage is "off-line" and must be loaded onto a system by a human operator before it can be accessed by software.

10. Optical disks are also used for archival storage.

11. Human factors is a field of engineering which seeks to match the characteristics of people with (computing) machines, to achieve a safe, comfortable, and efficient working environment.

12. Input devices: joystick, light pen, mouse, and microphone
    Output device: loudspeaker

13. (a) The lowest level of software is the input/output subroutines.

(b) These subroutines directly access to the system's hardware for input/output operations.

14. (a) An actuator is an output device, whereas a sensor is an input device.
(b) A relay is an actuator, a thermistor is a sensor.

15. (a) Firmware is software stored in ROM or EPROM.
(b) Microcontrollers rely more heavily on firmware than microprocessors.
(c) Microcontrollers usually have only a small amount of RAM and they lack a disk drive from which to load programs into RAM.

16. Microcontrollers include instructions to operate on and manipulate bits. These bits are sometimes 1-bit I/O ports on the microcontroller chip that are directly addressable through simple instructions.

17. Five possible products that are likely to use microcontrollers include a hand-held video game, a telephone answering machine, an electronic fish finder, a remote-controlled toy car, and a video camera.

Chapter 2 - Hardware Summary

1. Fujitsu, Siemens, Advanced Micro Devices, Philips

2. The most likely choice is the 8052 because it includes 8K bytes of on-chip ROM.

3.
```
        SETB    28H
```

   The bit address is found in Figure 2-6 in the text. The figure shows the relationship between addressable bits and the byte addresses where the bits are located.

4.
```
        MOV     C,00H
        ORL     C,01H
        MOV     02H,C
```

   All logical operations on bits must use the carry flag — the Boolean accumulator — as one of the bits in the operation. An initial "MOV C,bit" instruction is usually necessary before the logical operation can be performed. The result must be written to the destination address using a "MOV bit,C" instruction.

5.
```
        MOV     C,P0.0
        MOV     P3.0,C
```

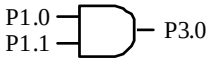
6.
```
        MOV     C,P1.0
        ANL     C,P1.1
        MOV     P3.0,C
```

7.
```
        MOV     C,P1.0
        JNB     P1,1,SKIP
        CPL     C
SKIP:   MOV     P3.0,C
```

8.
```
        MOV     C,P1.0
        ORL     C,P1.1
        CPL     C
        MOV     P3.0,C
```

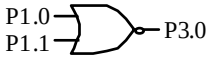
9.

For Problem 6:



For Problem 7:

For Problem 8:



10. (a) bits 31H, 32H, 35H
    (b) bits 31H, 33H, 34H-36H
    (c) bits EOH, E1H, E4H, DOH (Note: P bit in PSW set)
    (d) bits 78H-7FH
    (e) bit 91H
    (f) bits B2H, B3H]

11.

```
MOV       A,#55H
```

12.

```
          MOV     A,#0ABH
          MOV     DPTR,#9A00H
          MOV     @DPTR,A
```

13. 26

14. 07H (see Table 2-6)

15. (a)

```
          MOV     SP,#3FH
```

    (b)

```
          MOV     SP,#0BFH
```

16 (a)

```
          MOV     SP,#5FH
```

    (b)

```
          MOV     SP,#0DFH
```

17.

```
SUB:        PUSH    PSW         ;save previous status
            SETB    RS0         ;enable register bank 3
            SETB    RS1         ; RS0 = RS1 = 1
            ...                 ;execute subroutine
            POP     PSW         ;restore previous status
            RET                 ;return to main program
```

The subroutine is given the name "SUB". "PSW" is a pre-defined assembler symbol equivalent to "0D0H" (see Figure 2-2); so, PUSH PSW is the same as

```
            PUSH    0D0H
```

The PUSH and POP instructions only exist in the following forms:

```
            PUSH    direct
            POP     direct
```

So, registers can only be pushed on the stack or popped from the stack using the corresponding direct address or, as illustrated in this example, the equivalent pre-defined assembler symbol.

The SETB instructions could be replaced with

```
            MOV     PSW,#30H
```

which, in a single instruction, sets the RS0 and RS1 bits, thus activating register bank 3. The savings is one byte (3 bytes vs. 4 bytes); so, the latter method, despite being less "readable", may be preferred if conserving code memory is important.

18. (a) register bank 3
    (b) register bank 1
    (c) register bank 1

19. (a) register bank 1
    (b) register bank 0
    (c) register bank 2

20. 2.67 MHz

The "cycle" frequency of the 8051 is one twelfth the crystal frequency. However, ALE pulses twice per cycle (see Figure 2-9); therefore, the ALE frequency is one sixth the crystal frequency, or

16 / 6 = 2.67 MHz

21. 3 μs

A machine cycle lasts twelve periods of the crystal clock. At 4 MHz, this is 1/4 × 12 μs = 3 μs.

22. 1.67 MHz

ALE pulses twice per machine cycle, or once every 6 periods of the crystal clock. At 10 MHz, ALE pulses every 1/10 × 6 = 0.6 μs, for a frequency of 10/6 = 1.67 MHz.

23. 0.33 or 33%

As seen in Figure 2-9, ALE is high for four of the twelve phases in a machine cycle. 4/12 = 0.33.

24. (a) 3.0 μs

At 8 MHz, a machine cycle lasts 1/8 × 12 μs = 1.5 μs. Two machine cycles take 3.0 μs.)

(b) 5.68 ms

The calculation uses the formula for the charging of a capacitor with an applied voltage and a series resistor:

$$V_{RST} = V_{CC}(1 - e^{-1(t/RC)})$$
$$2.5 = 5.0(1 - e^{-1(t/(10uF*8.4K))})$$

and solving for t yields

t = 5.68 ms

25. 4

26. $\overline{\texttt{PSEN}}$ selects external EPROM. $\overline{\texttt{RD}}$ and $\overline{\texttt{WR}}$ select external RAMs.

27. 2FH

Figure 2-4 shows the addressable bit locations and the corresponding byte addresses.

28. 7BH

29.

Signal	Bit Address	Pin Number
P0.0	80H	39
P0.1	81H	38
P0.2	82H	37
P0.3	83H	36
P0.4	84H	35
P0.5	85H	34
P0.6	86H	33
P0.7	87H	32
P1.0	90H	1
P1.1	91H	2
P1.2	92H	3
P1.3	93H	4
P1.4	94H	5
P1.5	95H	6
P1.6	96H	7
P1.7	97H	8

Signal	Bit Address	Pin Number
P2.0	A0H	21
P2.1	A1H	22
P2.2	A2H	23
P2.3	A3H	24
P2.4	A4H	25
P2.5	A5H	26
P2.6	A6H	27
P2.7	A7H	28
P3.0	B0H	10
P3.1	B1H	11
P3.2	B2H	12
P3.3	B3H	13
P3.4	B4H	14
P3.5	B5H	15
P3.6	B6H	16
P3.7	B7H	17

30. (a) bit 7 in byte address 26H
    (b) bit 7 in byte address 2EH
    (c) bit 7 in byte address F0H

31. (a) bit 0 in byte address A8H
    (b) bit 4 in byte address 80H
    (c) bit 3 in byte address 2CH

32. SETB ACC.0

SETB exists in two forms:

```
SETB    C
```

a one-byte instruction which sets the carry flag (implicitly specified in the opcode), and

```
SETB    bit
```

a two-byte instruction which sets any bit-addressable location. The latter form requires the direct address of the bit. The solution is shown using the "dot operator", which allows a bit to be specified using the byte address of a bit-addressable location, followed by a period (or dot), followed by the bit position within the byte. The assembler converts this to the corresponding bit address.

Note that all bit-addressable special function registers have byte addresses with the least-significant three bits clear, or

```
aaaaa000
```

Substituting the bit position (specified in binary) into these three bits gives the correct bit address.

The answer shown above is equivalent to "SETB 0E0H" which explicitly provides the address of the least-significant bit in the Accumulator.

33. (a) P = 0
    (b) P = 1
    (c) P = 0

34. (a) P = 0
    (b) P = 0
    (c) P = 1

35.
```
MOV     DPTR,#0100H
MOV     A,R7
MOVX    @DPTR,A
```

The only instruction that writes to external data memory is MOVX @DPTR,A. Values written to external data memory, therefore, must be transferred to the accumulator first.

36.
```
MOV     DPTR,#08F5H
MOVX    A,@DPTR
MOV     0F0H,A
```

37. 08H (PC low-byte) and 09H (PC high-byte)

The 8051's Stack Pointer is set to 07H upon reset. Also, the SP is pre-incremented for push operations and post-decremented for pop operations. The first write to the stack following a system reset (assuming the SP is left as is) is to location 08H, and the second is to location 09H.

CALL instructions push the PC on the stack prior to branching to the subroutine. By convention on the 8051, the PC high-byte is pushed first, and the PC low-byte is pushed second.

38. (a) C0H

The stack can grow as high as FFH, so the maximum size of the stack is 64 bytes.

(b) On the 8031, this instruction is most likely a programming error, because the stack cannot exist above address 7FH – the highest memory location accessible using indirect addressing. (Note: The stack is accessed using indirect addressing using the instructions CALL, RET, RETI, PUSH, and POP. The stack pointer (SP) is the register used to access the stack.)

39. The initial value of the stack pointer after a system reset is 07H, so the stack will begin at address 08H and move "up" in memory. The register banks occupy locations 00H through 1FH, with register bank 0 at 00H-07H, register bank 1 at 08H-0FH, etc. The stack, therefore, overlaps the space assigned for register banks 1, 2, and 3. If a program uses these register banks, then the stack pointer must be initialized to a new value at the beginning of the program.

Any value 1FH or greater will do, as long as sufficient space is dedicated to the stack and as long as the stack does not exceed the highest indirectly accessible location (7FH on the 80x1, FFH on the 80x2). A minor, and unlikely, exception would be for a program that does not use the stack. In this case the stack pointer need not be initialized.

40. Power down mode can only be exited by a system reset; whereas, idle mode can be exited by system reset or any enabled interrupt.

41.

```
MOV     A,PCON      ;read PCON into A
ORL     A,#02H      ;set Power Down bit
MOV     PCON,A      ;write PCON with PD = 1
                    ;Power Down mode entered
```

The power control register is not bit-addressable, so setting bit 1 — the PD bit (see Table 2-4) — must use a byte transfer operation, as shown above.

If the previous content of PCON is of no concern, then this operation can be performed in a single instruction:

```
        MOV     PCON,#02H
```

42. See Figure 1.

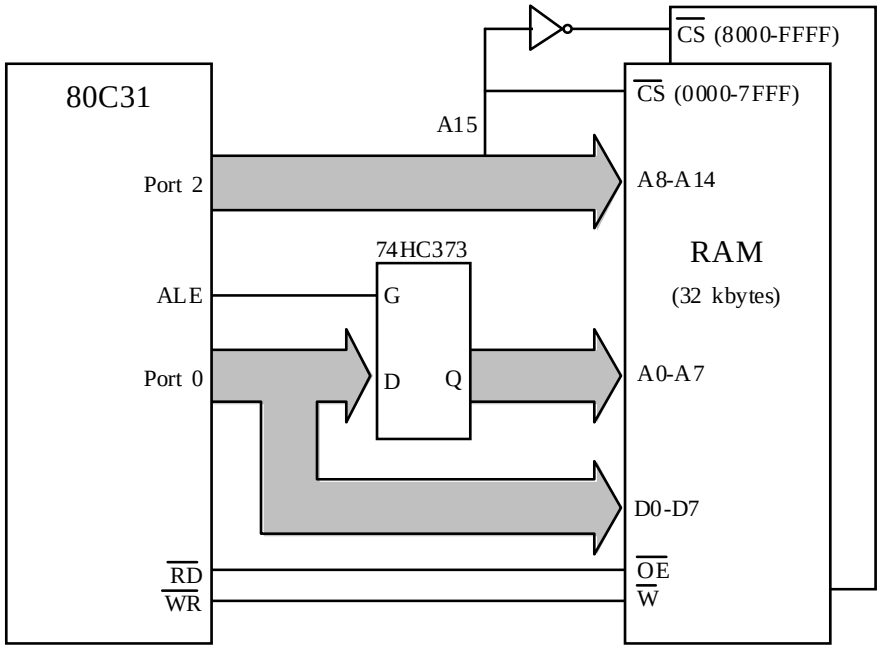


Figure 1. Interfacing RAMs to an 80C31

The interface uses A15 = 0 to select one RAM for addresses 0000H-7FFFH and A15 = 1 to select the other for addresses 8000H-FFFFH. The RAMs are selected only for accesses to external data memory since $\overline{\texttt{OE}}$ and $\overline{\texttt{W}}$ connect to the 8051's $\overline{\texttt{RD}}$ and $\overline{\texttt{WR}}$ lines respectively. Recall that (external) code memory is selected via $\overline{\texttt{PSEN}}$.

43. A = 00H
B = 00H
Internal RAM location 30H = 33H
SP = 07H

44. The phrase “I/O expansion” refers to increasing the number of input/output lines of a microprocessor or a microcontroller.

45. Microcontrollers that use memory-mapped I/O would treat I/O devices like memory locations, and so all instructions that access memory would apply to these I/O devices as well. In contrast, some microcontrollers connect to I/O devices through I/O ports meant specially for them. In this case, special-purpose I/O instructions are required to access these I/O devices.

46. The External Access ($\overline{EA}$) signal on pin 31 should be tied low to signify that the 8051 executes programs from external ROM. Meanwhile, the Program Store $\overline{Enable}$ (PSEN) pulses low during the fetch stage of an instruction. The program counter (PC) register contains the address of the next instruction to be executed, and upon reset, has the value of 00H which identifies the location of the first instruction to be fetched.

47. Even though the 8051 has 256 bytes of internal RAM, only the lower half is available for temporary storage of general data, whereas the upper half is reserved for special function registers. That's why we consider the effective size of internal data memory to be 128 bytes.

48. The stack is a sequence of locations in internal data memory that are used to temporarily store values in a last-in-first-out (LIFO) fashion. Meanwhile, the stack pointer is a register that contains the current location of the top of the stack.

Chapter 3 - Instruction Set Summary

1. (a) A3H

   (b) 74H, FEH

   (c) F0H

   (d) B4H, 0DH, 00H

   (e) C0H, E0H

   (f) D2H, A2H

2. (a) 85H, 84H, 83H

   (b) 30H, E0H, FCH

   (c) D0H, 83H

   (d) 74H, 3DH

   (e) 64H, 53H

   (f) C3H

3. (a)
```
MOV     R6,#2
```
   (b)
```
CLR     P1.7
```
   (c)
```
RRC     A
```
   (d)
```
MOV     @R0,A
```
   (e)
```
RET
```

(f)

```
MOV     DPTR,#$8030
```

4. (a) MOV A,R7

   (b) CALL 8050H

   (c) MOV TH1,A

   (d) INC A

   (e) MOVC A,@A+PC

   (f) MOV TL0,#-25

5.

```
MOV     direct,#data
MOV     dir,direct
CJNE    A,direct,relative
DJNZ    direct,relative
```

6.

```
AJMP    destination
ADD     A,#data
ADD     A,direct
```

7.

```
MOV     R0,#50H
MOV     A,@R0
```

R0 and R1 are the only registers that can function as pointers using indirect addressing. The first instruction initializes the pointer. The second instruction copies the contents of the memory location at by R0 to the Accumulator.

8. (i) MOV 3FH,A

   (ii) MOV R0,#3FH
   MOV @R0,A

9. A5H

The answer to this question is easily found by referring to the opcode map in Appendix B.

A vacant opcode allows for future expansion of the 8051 instruction set. Instructions could be added (by Intel) using two-byte opcodes beginning with A5H. So far, this has not occurred with any of the enhanced variations of the 8051.

Expanding an instruction set using unassigned opcodes has been done. Motorola's 6800 microprocessor had several unused opcodes. The subsequent 6809 microprocessor added numerous instructions to the basic 6800 repertoire. Many of the new instructions were encoded in two-byte opcodes beginning with either 10H or 11H, which were unassigned on the 6800.

10. 127

11. (a) 75H

    (b) 3 bytes

    (c) 1st byte: opcode
    2nd byte: direct address
    3rd byte: immediate data

    (d) 2 cycles

    (e) 1.5 μs

    The instruction in this problem is of the general form "MOV direct,#data". Appendix C contains the instruction definitions for all 8051 instructions, arranged alphabetically.

    The 8051 cycle frequency is $1/12^{th}$ the crystal frequency, which, for this problem, is 16 / 12 = 1.33 MHz. The cycle period is the reciprocal of the cycle frequency, or 1 / 1.33 = 0.75 μs. Since the instruction in this problem is a two-cycle instruction, it takes 2×0.75 = 1.5 μs to execute.

12. (a) B4H

    (b) 3

    (c) The first byte is the opcode. The second byte is the immediate data. The third byte is the relative offset for the jump (if it occurs).

    (d) 2

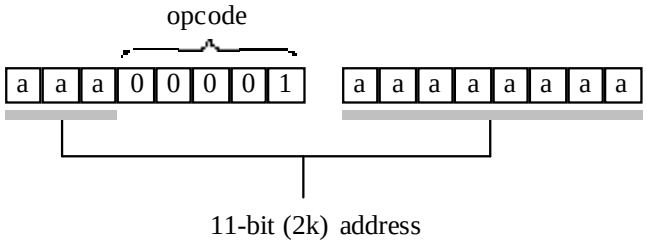
    (e) $2 \times 12/10 = 24/10 = 12/5 = 2.4$ microseconds

13. 1DH (0402H + 1DH = 041FH)

14. 8EH (A052H + FF8EH = 9FE0H)

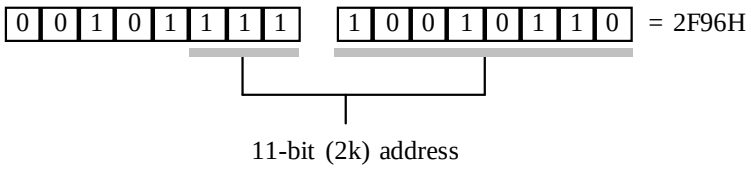
15. 1st byte: E1H
2nd byte: 96H

The AJMP instruction uses absolute addressing. The destination is specified as an 11-bit address only. The upper five bits of the address do not change. Since 25 = 32 and 64k / 32 = 2k, both the source and the destination of the jump must be within the same 2k page of code memory. The AJMP instruction is encoded with a 5-bit opcode and the 11-bit absolute (2k) address, as shown below.

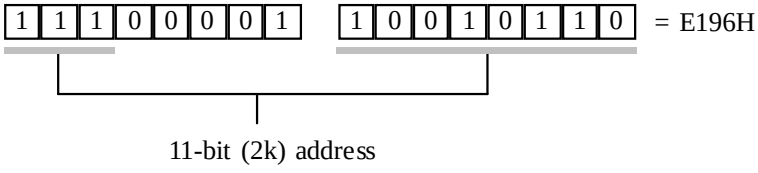


The destination address for this problem is 2F96H. The least-significant 11 bits are inserted into AJMP, as shown below.

Destination address:



AJMP encoding:



16. D1H, ABH

17.

```
          CJNE    A,#0DH,SKIP ;carriage return?
          LJMP    EXIT              ; yes: exit
SKIP:     ...                       ; no:  continue
```

Since the 8051 does not have a "compare and jump if equal" instruction, the arrangement shown above is needed. Of course, a "CJE" mnemonic could be defined as a macro:

```
%*DEFINE(CJE(VALUE, DESTINATION)) LOCAL SKIP
        (CJNE   A,#%VALUE,%SKIP
        LJMP    %DESTINATION
%SKIP:  )
```

The macro would be used as a solution for this problem as follows:

```
%CJE(0DH,EXIT)
```

18.

```
          CJNE    A,#'Q',SKIP
          JMP     EXIT
SKIP:     CJNE    A,#'q',SKIP2
          JMP     EXIT
SKIP2:    (continue)
```

19. 80H, ACH

Computing the relative offset for SJMP and the conditional jump instructions is tricky for students, at least initially. In fact, the calculation is simple if two points are considered: (1) the "from" address is the address following the jump instruction, and (2) the relative address is calculated using 16-bit signed notation and then is truncated to 8 bits.

The following equation is used:

```
"FROM" ADDRESS + RELATIVE OFFSET = DESTINATION ADDRESS
```

The "from" address in the problem is 0102H — the address following the SJMP instruction. The destination address is given as 00AEH. Thus

```
0102H + RELATIVE OFFSET =  00AEH
RELATIVE OFFSET = 00AEH - 0102H
                = 00AEH + (twos complement of 0102H)
                = 00AEH + FEFDH
                = FFABH
```

Only the lower byte (or ABH) is used.

Note that when the offset is computed as above in 16-bit notation, the upper 9 bits must be the same, otherwise the destination address is "out of range". Keeping the upper 9 bits

the same ensures that the destination address is no farther than -128 locations before or +127 locations after the "from" address. The legal range for the 16-bit offsets before truncating is shown below.

Hex	Decimal
FFF80H	-128
...	...
FFFFH	-1
0000H	0
0001H	1
...	...
007FH	+127

20. BFH, 5AH, BCH

21. Sets the carry flag

    SETB C is a better way to do this (because it is a one- byte instruction).

22. INC A is a one-byte instruction; whereas, INC ACC is a two-byte instruction of the general form "INC direct".

    Both instructions take the same length of time to execute (1 cycle), so the savings is purely in terms of memory.

    The trick in answering this question is in recognizing that "ACC" is a "pre-defined" assembler symbol, whereas "A" is a special assembler symbol. Pre-defined assembler symbols (e.g., ACC, TMOD, SBUF, DPH, RS0, RS1) have corresponding bit or byte addresses which are substituted as a direct address in the instruction. Special assembler symbols (e.g., A, C, DPTR, PC), although appearing in the operand field of an instruction, do not convert to addresses. The register or bit affected is identified implicitly in the opcode of the instruction.

23.

```
CLR     P1.7
NOP                 ;1 µs
NOP                 ;1 µs
NOP                 ;1 µs
NOP                 ;1 µs
SETB    P1.7        ;1 µs (total 5 µs)
```

The port bit is cleared at the end of the CLR P1.7 instruction, so timing of the pulse starts at the first instruction (NOP) following.

24. 02H (1st byte, opcode)
A0H (2nd byte, high-order byte of destination address)
F6H (3rd byte, low-order byte of destination address)

25. 6FH

26. 0A5H

27. 40H

28. 0A1H

29.

```
LOOP:     CPL    P1.7        ;1 cycle
          NOP                ;1 cycle
          NOP                ;1 cycle
          NOP                ;1 cycle
          SJMP   LOOP        ;2 cycles
```

A square wave with f = 83.3 kHz requires a period of 1 / 83.3 = 12 μs, with high-time = low-time = 6 μs. Three NOP instructions are needed to fine tune the period as required.

30.

```
LOOP:     CLR    P1.7        ;high (1 us)
          NOP                ;low  (1 us) begin pulse
          NOP                ;low  (1 us)
          NOP                ;low  (1 us)
          SETB   P1.7        ;low  (1 us) end pulse
          MOV    R7,#96      ;high (1 us)
WAIT:     DJNZ   R7,WAIT     ;high (96 x 2 = 192 us)
          SJMP   LOOP        ;high (2 us)
```

Generating a 4 μs pulse every 200 μs requires 196 μs high-time between pulses. The solution above uses a software loop to achieve the timing delay. The loop is a single 2 μs instruction (DJNZ) which branches to itself. The loop is only 192 μs because an additional 4 μs are consumed in other instructions.

The 4 μs active-low pulse begins at the end of the CLR P1.7 instruction and finishes at the end of the SETB P1.7 instruction. Three NOP instructions are needed to stretch the low-time to 4 μs.

Note that the delay count has been specified in decimal notation (96_{10}), rather that in hexadecimal. Since the problem is "thought-out" in terms of the decimal count required, this is the preferred notation. "Let the assembler do the work" aptly portrays the rationale for using decimal notation.

Laboratory Project Suggestions:

(a) Using P1.7, write a program that generates a 10 μs active-high pulse every 100 μs.

(b) Using P1.5, write a program that generates a 100 μs active-low pulse every millisecond.

31. Programs:

(a)

```
LOOP:     MOV    C,P1.4      ;n1 = 1 cycle
          ORL    C,P1.5      ;n2 = 2 cycles
          ORL    C,P1.6      ;n3 = 2 cycles
          CPL    C           ;n4 = 1 cycle
          MOV    P1.7,C      ;n5 = 2 cycles
          SJMP   LOOP        ;n6 = 2 cycles
```

(b)

```
LOOP:     CLR    P2.7        ;n1 = 1 cycle
START:    MOV    A,P1        ;n2 = 1 cycle
          CPL    A           ;n3 = 1 cycle
          JZ     LOOP        ;n4 = 2 cycles
          SETB   P2.7        ;n5 = 1 cycle
          SJMP   START       ;n6 = 3 cycles
```

(c)

```
LOOP:     MOV    C,P1.5      ;n1 = 1 cycle
          ANL    C,/P1.4     ;n2 = 2 cycles
          CPL    C           ;n3 = 1 cycle
          ORL    C,P1.6      ;n4 = 2 cycles
          MOV    P1.7,C      ;n5 = 2 cycles
          SJMP   LOOP        ;n6 = 2 cycles
```

Propagation delays:

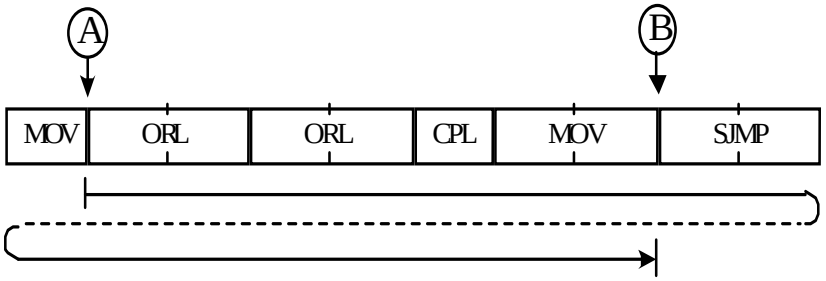
(a) 17 μs

The worse case delay occurs if a transition on P1.4 is just missed by the first instruction. The delay is computed from the second instruction up to the end of the second occurrence of MOV P1.7,C. In order of instruction execution, the number of cycles, n, is computed as follows:

$$n = n_2 + n_3 + n_4 + n_5 + n_6 + n_1 + n_2 + n_3 + n_4 + n_5$$
$$= 2 + 2 + 1 + 2 + 2 + 1 + 2 + 2 + 1 + 2$$
$$= 17$$

With a 12 MHz clock, 17 cycles translates into 17 μs.

The following figure helps visualize the timing. The solid line illustrates the instruction sequence for the worst-case scenario, as just described.



(b) 11 μs

The worst-case delay is a bit tricky for this problem. The path through the program is different depending on whether the output (P2.7) is high or low. The two possibilities are

(i) the output is low because all inputs are high, or

(ii) the output is high because at least one input is low.

For (i), the output is low and the instruction sequence is

n_1, n_2, n_3, n_4, n_1, n_2, n_3, n_4, n_1, etc.

For (ii), the output is high and the instruction sequence is

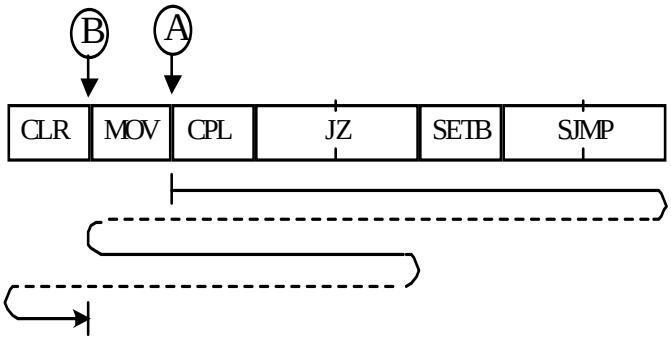
n_2, n_3, n_4, n_5, n_6, n_2, n_3, n_4, n_5, n_6, n_1, etc.

The inputs at port 1 are sensed at the second instruction (n_2) and, if a change at the output is necessitated, it is determined in the fourth instruction (n_4). The output is changed either at the first instruction (n_1) or at the fifth instruction (n_5). To determine the worst-case propagation delay, both scenarios must be considered. In either case, the approach is the same: Begin by assuming an input transition is just missed at n_2, then follow through a complete sequence of instructions until the next n_2, then switch to the other sequence at n_4 until either n_1 (output low) or n_5 (output high) is reached.

For the output switching high-to-low, the worst-case number of cycles, n, is tabulated as follows:

$$n = n_3 + n_4 + n_5 + n_6 + n_2 + n_3 + n_4 + n_1$$
$$= 1 + 2 + 1 + 2 + 1 + 1 + 2 + 1$$
$$= 11$$

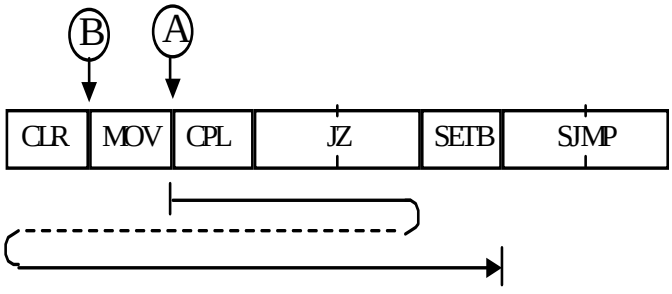
This is illustrated below. Remember, point "A" in the figure is where the timing begins (having just missed an input transition), and point "B" is where the correct output state finally appears.



For the output switching low-to-high, the worst-case number of cycles, *n*, is tabulated as follows:

$$n = n_3 + n_4 + n_1 + n_2 + n_3 + n_4 + n_5$$
$$= 1 + 2 + 1 + 1 + 1 + 2 + 1$$
$$= 9$$

This is illustrated below.



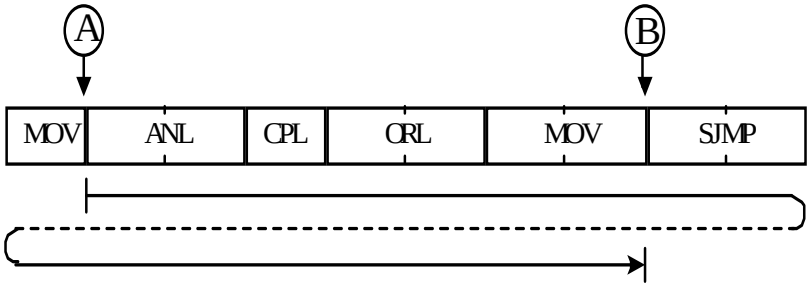
So, the worst case is 12 CPU cycles. At 12 MHz, 12 cycles translates into a worst-case propagation delay of 12 μs.

(c) 17 μs

The worst-case delay will occur if an input transition in P1.5 is just missed in the first instruction, n_1. Timing begins from n_2 and continues through the loop until the next execution of n_5 when the correct state appears on output pin P1.7. The total number of CPU cycles, n, is computed as follows:

$$\begin{aligned} n &= n_2 + n_3 + n_4 + n_5 + n_6 + n_1 + n_2 + n_3 + n_4 + n_5 \\ &= 2 + 1 + 2 + 2 + 2 + 1 + 2 + 1 + 2 + 2 \\ &= 17 \end{aligned}$$

With a 12 Mhz clock, the worst-case propagation delay is 17 μs. The sequence of instructions is illustrated below.



32. (a)

```
LOOP:     MOV    C,P1.0      ;1 cycle
          JNB    P1.1,SKIP   ;2 cycles
          CPL    C           ;1 cycle
SKIP:     ORL    C,P1.2      ;1 cycle
          MOV    P1.3,C      ;1 cycles
          SJMP   LOOP        ;2 cycles
```

The worst-case propagation delay is 13 μs, as illustrated in Figure 2. This will occur if the input signal on P1.0 changes just after the first instruction ("A", below). The change is not sensed until the next pass through the loop. The output signal on P1.3 is correctly updated on the fifth instruction ("B", below). All this takes 13 CPU cycles, or 13 microseconds if a 12 MHz clock is used. The solid line indicates the sequence of instruction execution for the worst-case propagation delay.

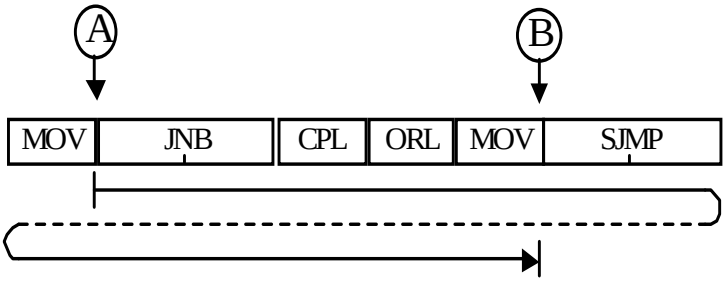


Figure 2. Worst-case timing example

(b)

```
LOOP:     MOV     C,P1.7      ;1 cycle
          ORL     C,P1.6      ;1 cycle
          CPL     C           ;1 cycle
          ANL     C,/P1.5     ;1 cycle
          CPL     C           ;1 cycle
          MOV     P1.4,C      ;1 cycle
          SJMP    LOOP        ;2 cycles
```

The worst-case propagation delay is 13 microseconds (see part a).

(c)

```
LOOP:     MOV     A,P1        ;1 cycle
          ANL     A,#0F0H     ;1 cycle
          JZ      SKIP        ;2 cycles
          CLR     P1.0        ;1 cycle
          SJMP    LOOP        ;2 cycles
SKIP:     SETB    P1.0        ;1 cycle
          SJMP    LOOP        ;2 cycles
```

The worst-case propagation delay is 11 microseconds (see part a). The delay is the same whether or not the jump is taken at the JZ instruction.

33. 79H

34. 0D2H (1st byte, opcode)
    0A6H (2nd byte, bit 6 of Port 2)

35.

```
MOV       C,F0
MOV       P1.5,C
```

36. (i) if the jump is forward, or
    (ii) if the jump destination is not within the same 2k page as the address following the jump opcode

37. PC = 789AH

The PC is saved on the stack low-byte first, high-byte second. When the PC is restored by a RET instruction, the first byte read from the stack is placed in the PC high-byte; the second byte read is placed in the PC low- byte. Since the SP contained 0BH before the RET instruction, the first byte read is 78H (from location 0BH) and the second byte read is 9AH (from location 0AH). Remember that the 8051 stack is "pre-increment" for store or push operations, and "post-decrement" for retrieve or pop operations.

38. (a) The subroutine clears internal memory locations 20H to 7FH.

(b) Number of cycles in each instruction (in order of appearance) is 1, 1, 1, 2, and 2.

(c) Number of bytes in each instruction (in order of appearance) is 2, 2, 1, 3, and 1.

(d)

```
0000  7820    SUB:   MOV   R0,#20H
0002  7600    LOOP:  MOV   @R0,#0
0004  04             INC   R0
0005  B880FA        CJNE  R0,#80H,LOOP
0008  22             RET
```

(e) $t = 1\ \mu s \times [1 + 96(1 + 1 + 2) + 2] = 387\ \mu s$

(Note: The loop executes 80H - 20H = 60H = 6 × 16 = 96 times.)

39.

```
LOOP:     MOV     A,P3         ;read port bits into A
          ANL     A,#0FH       ;clear upper bits (if set)
          ACALL   CONVRT       ;convert to segment data
          MOV     P1,A         ;send code to LEDs
          SJMP    LOOP         ;repeat
CONVRT:   INC     A
          MOVC    A,@A+PC
          RET
          DB      00111111B    ;0 (1 = segment ON)
          DB      00000110B    ;1
          DB      01011011B    ;2
          DB      01011011B    ;3
          DB      01100110B    ;4
          DB      01101101B    ;5
          DB      01111101B    ;6
          DB      00000111B    ;7
          DB      01111111B    ;8
          DB      01001111B    ;9
          DB      01110111B    ;A
          DB      01111100B    ;B
          DB      00111001B    ;C
          DB      01011110B    ;D
          DB      01111001B    ;E
          DB      01110001B    ;F
```

40. (a) MOV
    (b) MOVC
    (c) MOVX
    (d) MOVC

41. A lookup table is a table typically stored in ROM whose entries can be obtained by way of an index or offset that is added to the base of the table.

42. The MOV A, @R0 instruction uses indirect addressing mode. The source data is located in a memory location whose address is stored in the register R0. The MOV A, R0 uses register addressing mode. The contents of R0 are the actual data for use by the instruction.

43. The first half of the program writes the value 55H to memory locations from 10H up to 1FH. The second half of the program writes the value AAH to memory locations from 20H down to 60H. This means that it first writes to 20H, then to 1FH, and after that to 1EH, and so on until 00H. Decrementing further to obtain FFH, it proceeds to write AAH to each memory location and so forth until 60H is reached.

44. Move the value FFH to memory locations from 7FH down to 21H. Move the value 00H to memory locations from 30H up to 5EH

45. (a) Only memory location 30H is changed since the instruction complements the bit address 7FH which is the MSB of memory location 30H. The other memory locations are unchanged.

(b) Only memory location 30H is changed since the instruction clears the bit address 78H which is the LSB of memory location 30H. The other memory locations are unchanged.

(c) Only memory location 7FH is changed since the value in 78H is moved into 7FH.

46.
```
        MOV R0, #30H            ;point to first number
        MOV A, #0               ;initialize sum to zero
LOOP:   ADD A, @R0              ;add current number into sum
        INC R0                  ;point to next number
        CJNE R0, #70H, LOOP     ;all numbers added?
        MOV @R0, A              ;yes: move result to 70H
```

47. The following program calculates N! = N × (N – 1) × (N – 2) × … × 2 × 1.

```
        MOV B, 55H        ;B = N
        MOV A, B          ;A = N
        DEC B             ;B = N – 1
NEXT:   MUL AB            ;product of N and all numbers
        DJNZ B, NEXT      ; smaller than it, down to 1
        MOV 77H, A        ;store factorial result
```

48.
```
        ORG 0
LOOK:   MOV A, #0AH
        ADD A, #01H
        MOVC A, @A+PC
        SJMP OVER
        DB 0
        DB 6
        DB 10
        DB 12
        DB 14
        DB 16
        DB 17
        DB 18
        DB 19
        DB 20
        DB 21
        DB 22
        DB 22
        DB 23
```

```
            DB 24
OVER:       SJMP $
            END
```

49.

```
            LO EQU 0300H
            HI EQU 0400H
            ORG 0
LOOK:       PUSH ACC          ;assume value already in A,
                              ; now back it up
            MOV DPTR, #LO     ;point to low-byte table
            MOVC A, @A+DPTR   ;get low byte of result
            MOV R0, A         ;store low byte in R0
            POP ACC           ;restore A
            MOV DPTR, #HI     ;point to high-byte table
            MOVC A, @A+DPTR   ;get high byte of result
            MOV R1, A         ;store high byte in R0
            SJMP $            ;do nothing

            ORG LO            ;low-byte table for e^x
            DB 1              ;low byte of e^0
            DB 3              ;low byte of e^1
            DB 7              ;low byte of e^2
            DB 20             ;low byte of e^3
            DB 55             ;low byte of e^4
            DB 148            ;low byte of e^5
            DB 147            ;low byte of e^6
            DB 73             ;low byte of e^7
            ;the remaining low byte results are stored here ...

            ORG HI            ;high-byte table for e^x
            DB 0              ;high byte of e^0
            DB 0              ;high byte of e^1
            DB 0              ;high byte of e^2
            DB 0              ;high byte of e^3
            DB 0              ;high byte of e^4
            DB 0              ;high byte of e^5
            DB 1              ;high byte of e^6
            DB 4              ;high byte of e^7
            ;the remaining high byte results are stored here ...
            END
```

50.

```
CLR A
```

```
      MOV R3, #0
      LOOP: DEC R1            ;first time adding
      ADD A, R0         ;add R0 with previous sum
      JNC SKIP          ;sum is less than 255, go to SKIP
      CLR C             ;if addition caused a carry (sum > 255)
      INC R3            ;carry it over to the high byte, R3
      SKIP: CJNE R1, #0, LOOP ;have we added R0 enough times?
      MOV R2, A         ;low byte of sum move into R2
      END
```

51. Subroutines are groups of instructions that are written to perform a specific function. We normally use subroutines especially for those groups of instructions that are often used, so instead of having to repeatedly write them, we only need to call the same subroutine every time it is needed. This reduces the size of the assembly language program.

52. (a)

```
POW:  MOV R0, A   ;backup the base (A) in R0
      MOV R1, B   ;backup the exponent (B) in R1
      DEC R1      ;adjust the exponent for correct calculation
      MOV B, #0   ;initially, the value is 8 bits
NEXT: PUSH B      ;backup the high byte result
      MOV B, R0   ;set up low-byte to be multiplied A
      MUL AB      ;multiply low-byte with A
      MOV R7,A    ;backup the low byte result
      MOV R6,B    ;backup the next higher byte
      POP B       ;set up high-byte to be multiplied with A
      MOV A, R0
      MUL AB      ;multiply high-byte with A
      ADD A, R6   ;current low byte result add to previous
                  ; higher byte result
      MOV B, A    ;final high byte result put in B
      MOV A, R7   ;final low byte result put in A
      DJNZ R1, NEXT     ;decrement the exponent and repeat if it ; is
                        still nonzero
      RET
      END
```

(b)

```
CALCULATE:  MOV A, #2   ;get ready to calculate second term
            MOV B, #3
            CALL POW    ;calculate 2^3
            MOV R0, A
            MOV R1, B
            MOV A, #3   ;get ready to calculate first term
            MOV B, #4
            CALL POW    ;calculate 3^4
            SUBB A, R0  ;subtract the low-byte terms
            PUSH A      ;backup low-byte result
            MOV A, B    ;prepare to subtract the high-byte terms
            SUBB A, R1  ;subtract the high-byte terms
```

```
                MOV B, A    ;high-byte should be in B
                POP A       ;low-byte should be in A
                RET
                END
```

53.
```
SUM:        ADD A, B    ;A = A + B
            RET

TOTAL:      MOV A, R1   ;R1 = A
            MOV B, R2   ;R2 = B
            CALL SUM    ;A = A + B = R1 + R2
            MOV B, R3   ;B = R3
            CALL SUM    ;A = A + B = (R1 + R2) + R3
            MOV R4, A   ;R4 = A = (R1 + R2) + R3
            RET
            END
```

54.
```
XCHH:       SWAP A
            PUSH A
            MOV A, B
            SWAP A
            MOV B, A
            POP A
            MOV R0, #30H
            MOV @R0, B
            XCHD A, @R0
            MOV B, @R0
            RET
            END
```

55.
```
XRB:        JC    CARRY       ;if C=1, go to CARRY
NOCARRY:    JB    P, SET      ;if P=1, since C=0, SET
                  SJMP  CLEAR ;else since P=0 & C=0, CLEAR
CARRY:      JNB   P, SET      ;if P=0, since C=1, SET
CLEAR:      CLR   C           ;else since P=1 & C=1, CLEAR
                  SJMP  STOP  ;return
SET:        SETB  C           ;set C
STOP:       RET               ;return
            END
```

56. (a)
```
GUESSME:    CLR C             ;clear the carry flag
            RRC A             ;rotate A right through carry
            DJNZ  R0, GUESSME ;decrement R0 and repeat if not 0
            END               ;assembler directive to denote
                              ; end of the program
```

(b)
The subroutine first clears C before a rotate right through carry is done on A. This means that all bits of the A are shifted to the right by one bit position, except for the LSB which

is shifted out while a value of 0 is shifted into the MSB from the carry. This is similar to a shift right operation where the LSB is lost and a zero is shifted into the MSB. This shifting continues as long as R0 is still not zero. Hence, the subroutine essentially shifts the accumulator right by R0 bit positions or can be denoted as SR A, R0.

57.
```
      MOV R7, #0  ;R7 to temporarily store the quotient
      MOV A, R0   ;copy dividend to A
LOOP: SUBB A, R1  ;subtract divisor from dividend
      JC OVER     ;negative value, over subtracted!
      INC R7      ;increment quotient counter
      JZ DONE     ;if no remainder, done
      SJMP LOOP   ;still need to subtract, so repeat
OVER: ADD A, R1   ;add back the over subtracted amount
DONE: MOV B, A    ;put remainder in B
      MOV A, R7   ;put quotient in A
      END
```

58. (a)
```
      MOV A, PSW        ;move C and P bits to A
      MOV C, ACC.7      ;convert bit C to byte in P1
      MOV P1.0, C       ; via C
                        ;bit P is already in LSB of A
      XRL A, P1         ;xor the bytes containing C and P
      MOV C, ACC.0      ;convert xor result to bit in C
      END
```

(b)
```
      MOV A, PSW        ;move bits C and P to A
                        ; so C in ACC.7, P in ACC.0
      JC NEXT           ;if C = 1, goto NEXT
      JNB ACC.0, CLRC   ;C = 0: if P = 0, goto CLRC
      SJMP SETC         ;C = 0: if P = 1, goto SETC
NEXT: JB ACC.0, CLRC    ;C = 1: if P = 1, goto CLRC
SETC: SETB ACC.7        ;XOR result is 1
      SJMP STOP
CLRC: CLR ACC.7         ;XOR result is 0
STOP: MOV C, ACC.7      ;put result in C
      END
```

59.
```
PUSH 30H    ;push all values onto stack
PUSH 31H
PUSH 32H
PUSH 33H
```

```
PUSH 34H
PUSH 35H
PUSH 36H
PUSH 37H
PUSH 38H
PUSH 39H
POP 30H     ;and pop back
POP 31H     ;order will automatically be reversed
POP 32H
POP 33H
POP 34H
POP 35H
POP 36H
POP 37H
POP 38H
POP 39H
END
```

60.
```
;assume 16-bit numbers are in 30H:31H and 40H:41H
; and sum is to be stored in 50H:51H
MOV A, 31H  ;move first low byte to A
ADD A, 41H  ;add to second low byte
MOV 51H, A  ;store low byte sum in 51H
MOV A, 30H  ;move first high byte to A
ADDC A, 40H ;add to second high byte, with carry
MOV 50H, A  ;store high byte sum in 50H
END
```

Chapter 4 - Timer Operation

1.

```
LOOP:   CPL     P1.5    ;1 cycle
        NOP             ;1 cycle
        NOP             ;1 cycle
        SJMP    LOOP    ;2 cycles (total = 5 cycles)
```

For a frequency of 100 kHz, a period of 1 / 100 = 0.01 ms = 10 μs is needed. A square wave of this frequency requires a high-time of 5 μs and a low-time of 5 μs.

2. Enable Timer 1

3. Timer 1: pulse width measurement of signal on -INT1
Timer 0: event counting on T0

4. 333 kHz, 25%

5.

```
HUND    EQU     100             ;100 ¥ 10000 = 1 second
COUNT   EQU     -10000
        MOV     TMOD,#01H       ;Timer 0, Mode 1
START:  JNB     P1.6,$          ;wait for P1.6 = 1
        JB      P1.6,$          ;wait for P1.6 = 0
        CLR     F0              ;clear user flag in PSW
INIT:   CLR     TR0             ;stop timer (if running)
        MOV     R7,#HUND        ;prepare R7 and timer for
INIT2:  MOV     TH0,#HIGH COUNT ; 1 second delay
        MOV     TL0,#LOW COUNT
        CLR     P1.7            ;sound buzzer and
        SETB    TR0             ; begin timeout
CHECK:  JB      TF0,DONE        ;check for timeout
        JNB     P1.6,SKIP       ;when P1.6 = 1,
        SETB    F0              ; set user flag
        SJMP    CHECK
SKIP:                           ;if P1.6 = 0 & F0 = 1,
        JBC     F0,INIT         ; re-init timeout,
        SJMP    CHECK           ; otherwise, check TF0
DONE:   CLR     TF0
        CLR     TR0
        DJNZ    R7,INIT2
        SETB    P1.7            ;turn off buzzer
        SJMP    START
```

Restarting the timeout if a 1-to-0 transition occurs during the 1 second buzz requires that the timer delay routine test the TF0 flag and the P1.6 input. If P1.6 undergoes a new 1-to-0 transition, the timeout should be restarted; otherwise, turn off the buzzer when the timeout expires (TF0 = 1). To do all this, the delay subroutine is merged with the main instruction loop.

The user flag (F0) in the PSW is introduced in the solution to get around a tricky problem. Since a 1-to-0 transition initiates the timeout, it cannot be restarted simply by checking for a zero on P1.6. The timeout should only be restarted if a zero is detected after P1.6 has returned high. This can be seen in the timing diagram in the text. The solution is to set the F0 flag in the PSW when P1.6 returns high. The necessary condition for restart, therefore, is a zero level on P1.6 and F0 = 1.

6.

```
        MOV     TMOD,#02H       ;8-bit auto-reload mode
        MOV     TH0,#-42        ;reload count
        SETB    TR0             ;start timer
LOOP:   JNB     TF0,LOOP        ;wait for overflow
        CLR     TF0             ;clear overflow flag
        CPL     P1.2            ;complement port bit
        SJMP    LOOP            ;repeat
```

The required period for f = 12 kHz is 1 / 12 = 0.0833 ms = 83.3 μs. The closest square wave will have a period of 84 μs, for high- and low-times of 42 μs each. Since 42 < 256, 8-bit auto-reload mode can be used.

7.

```
COUNT   EQU     -10000          ;count 10,000 "events"
LIGHT   BIT     P1.7
        MOV     TMOD,#05H       ;16-bit "counter" mode
        MOV     TH0,#HIGH COUNT ;initialize count to
        MOV     TL0,#LOW COUNT  ; -10,000
        CLR     LIGHT           ;ensure light is off
        SETB    TR0             ;start timer/counter
        JNB     TF0,$           ;wait for 10,000th count
        SETB    LIGHT           ;turn light on
        SJMP    $               ;done!
```

The solution to this problem is surprisingly easy. Bit 2 in TMOD is the C/-T bit for Timer 0. Setting it connects external pin 14 as the clocking source for Timer 0. This pin is P3.4, which is called T0 when used in this manner. One-to-zero transitions on this pin increment the timer/counter. By initializing the count to -10,000, an overflow occurs on the 10,000th 1-to-0 transition ("event") on T0. When this occurs, the light is turned on.

8.

```
COUNT   EQU     -1136           ;reload value
        MOV     TMOD,#01H       ;16-bit timer mode
LOOP:   MOV     TH0,#HIGH COUNT ;(2 cycles overhead)
        MOV     TL0,#LOW COUNT  ;(2)
        SETB    TR0             ;(1) start timer
        JNB     TF0,$           ;(2) wait for overflow
        CPL     P1.1            ;(1) toggle output
        CLR     TR0             ;(1) stop timer
        CLR     TF0             ;(1) clear overflow flag
        SJMP    LOOP            ;(2) repeat
                                ;total = 12 cycles overhd
```

f(Actual) = 436 Hz

f(Crystal) = 11.9616 MHz

Inexpensive hand-held oscillators generating a 440 Hz tuning standard are a valuable tool-of-the-trade for today's musician. The required period is 1 / 440 × 0.0022727273 seconds = 2,272.7273 μs. If this is rounded to 2,272 μs, the resultant frequency is 440.14 Hz for an error of 0.032%, which is negligible. (This error is 1 / 200th of a musical semitone — a discrepancy which cannot be discerned by the human ear.)

Generating a square wave of this frequency requires high- and low-times of 1136 μs each. Since 1136 > 256, 16-bit timer mode is required.

Since auto-reload mode is not used, the execution time for the instructions in the loop represent overhead; that is, error. Twelve cycles are added onto the high- and low-time of the output waveform, for a resultant frequency of 1 / (2 × [1136 + 12]) = 436 Hz.

If the count is adjusted to -1136 + 12 = -1124, then the output frequency becomes 1 / (2 × [1124 + 12]) = 440.14 Hz, as calculated previously. All calculations thus far assume 12 MHz operation which corresponds to a cycle time of 1 μs. Reducing the crystal frequency by 0.032% to 11.9616 MHz will produce exactly 440 Hz (using the above program with a reload count of -1124).

9.

```
COUNT1  EQU     -600
COUNT2  EQU     -1400
ADJ1    EQU     10              ;adjustment for COUNT1
ADJ2    EQU     12              ;adjustment for COUNT2
        MOV     TMOD,#01H
LOOP:   SETB    P1.0            ;(1) begin high-time
        MOV     TH0,#HIGH (COUNT1 + ADJ1) ;(2)
        MOV     TL0,#LOW (COUNT1 + ADJ1)  ;(2)
        SETB    TR0             ;(1) start timer
        JNB     TF0,$           ;(2) wait for overflow
        CLR     TF0             ;(1)
        CLR     TR0             ;(1)
        CLR     P1.0            ;(1) begin low-time
        MOV     TH0,#HIGH (COUNT2 + ADJ2) ;(2)
        MOV     TL0,#LOW (COUNT2 + ADJ2)  ;(2)
        SETB    TR0             ;(1)
        JNB     TF0,$           ;(2)
        CLR     TF0             ;(1)
        CLR     TR0             ;(1)
        SJMP    LOOP            ;(2)
```

The required period of the waveform is 1 / 0.5 = 2 ms = 2,000 μs. The required high-time is 0.3 × 2000 = 600 μs. The required low-time is 1400 μs. Since 600 > 256, 16- bit timer mode must be used.

Adjustments are included in the solution above to accommodate the overhead of initializing the timers. An extra 2 cycles is needed for the low-time adjustment because of the SJMP instruction.

10.

```
COUNT   EQU     -60
HOUR    EQU     50H
MINUTE  EQU     51H
SECOND  EQU     52H
        MOV     50H,#0          ;clear time (midnight)
        MOV     51H,#0
        MOV     52H,#0
        MOV     T2CON,#02H      ;16-bit counter mode
        MOV     RCAP2H,#HIGH COUNT
        MOV     RCAP2L,#LOW COUNT
        SETB    TR2
LOOP:   JNB     TF2,$           ;wait for overflow
        CLR     TF2             ;clear overflow flag
;
        INC     SECOND          ;increment seconds
        MOV     A,SECOND        ;seconds > 59?
        CJNE    A,#60,LOOP      ; no: continue
        MOV     SECOND,#0       ;yes: reset and ...
;
        INC     MINUTE          ;increment minutes
        MOV     A,MINUTE        ;minutes > 59?
        CJNE    A,#60,LOOP      ; no: continue
        MOV     MINUTE,#0       ;yes: reset and ...
;
        INC     HOUR            ;increment hours
        MOV     A,HOUR          ;hours > 11?
        CJNE    A,#12,LOOP      ; no: continue
        MOV     HOUR,#0         ;yes: reset and
        SJMP    LOOP            ;continue
```

Since Timer 2 receives 60 clock pulses per second, a reload value of -60 is needed. An overflow occurs every second.

An alternative and preferred method for defining the internal RAM location for HOUR, MINUTE, and, SECOND is to define storage locations in the internal data segment, as shown below.

```
        DSEG    AT 50H
HOUR:   DS      1
MINUTE: DS      1
SECOND: DS      1
```

The optional "AT 50H" sets the absolute addresses to conform to the requirement of the problem. In most situations the actual address is of no concern. If "AT 50H" is left out, the assembler will used the next available addresses in the internal data segment. For small, self-contained programs such as that shown above, assignments begin at address 08H, just above the default locations for register bank 0.

For larger applications using several modules and/or relocatable segments, HOUR, MINUTE, and SECOND can be defined in a relocatable segment as follows:

```
ONCHIP   SEGMENT DATA
         RSEG    ONCHIP
HOUR:    DS      1
MINUTE:  DS      1
SECOND:  DS      1
```

Laboratory Project Suggestions:

For the following lab projects, a few simple changes may be necessary. First, Timer 0 can be used if the target system has an 8051/8031 IC instead of an 8052. Second, a function generator or 555 timer circuit can be used to provide a 60 Hz time-base if a power-line transformer is unavailable.

(a) Using standard subroutine(s), modify the program above to report the time on the terminal's CRT display each second. (Hint: Begin the revision by establishing a BCD format for the time.)

(b) Develop a simple user interface that allows setting the time from the keyboard. Merge the user interface with the project above as follows. After setting the time, begin outputting the time once per second. If any key is struck on the keyboard suspend output and return to the user interface to set a new time.

(c) Attach a debounced switch to an available input on Port 1 and report the time on the output display each time the switch is toggled.

Other possibilities: (a) use interrupts, (b) set an alarm time and sound a tone when the alarm goes off, (c) improve the interface to use direct cursor addressing (i.e., update the time directly at the correct x-y screen coordinates).

11. (a)

1kHz means the period is 1000ms. For the rectangular pulse, the high time should be 300ms and the low time 700ms. Both intervals cannot be timed by a timer in mode 2 hence mode 1 has to be used. Meanwhile, for the square wave, the high time and low time should both be 500ms. Mode 1 should also be used.

High/low time Calculation = 2 marks
Square wave generation = 2 marks
Rectangular wave generation = 3 marks

Simultaneous operation = 1 marks

```
        ONE   EQU –300
        ZERO  EQU –700
        HALF  EQU –500

        MOV TMOD, #00010001B    ;both timers to be used in mode 1
        MOV TH1, #HIGH ONE      ;timer 1 for the rectangular wave
        MOV TL1, #LOW ONE       ; and first we generate the high
        MOV TH0, #HIGH HALF     ;timer 0 used for the square wave
        MOV TL0, #LOW HALF
        SETB P1.0               ;first we generate the high time
        SETB TR1                ;start Timer 1
        SETB TR0                ;start Timer 0
LOOP:   JB TF1, RECT            ;if Timer 1 overflows, go to RECT
        JNB TF0, LOOP           ;if Timer 0 overflows, go below
SQU:    CLR TR0                 ;timer 0 overflows. Stop Timer 0
        CLR TF0                 ;clear overflow flag
        MOV TH0, #HIGH HALF     ;reload the high and low times
        MOV TL0, #LOW HALF
        CPL P2.0                ;this is obvious, right?
        SETB TR0                ;start Timer 0 again
        SJMP LOOP               ;go back to waiting for overflow

RECT:   CLR TR1                 ;timer 1 overflows. Stop Timer 1
        CLR TF1                 ;clear overflow flag
        JNB P1.0, SETIT         ;if previous low time, go to SETIT
        CLR P1.0                ;previously was high time. Clear it
        MOV TH1, #HIGH ZERO     ;reload for low time interval
        MOV TL1, #LOW ZERO
        SJMP CONT               ;skip over the high time processing
SETIT:  SETB P1.0               ;previously was low time. Set it
        MOV TH1, #HIGH ONE      ;reload for high time interval
        MOV TL1, #LOW ONE
CONT:   SETB TR1                ;restart Timer 1
        SJMP LOOP               ;go back to waiting for overflow
        END
```

12. f = 2.5kHz, hence one period is T = 1/f = 0.0004s = 400μs.
To generate a waveform with a 20% duty cycle, this means that the waveform would only be

HIGH for 20% of a period, or 20% × 400μs = 80μs

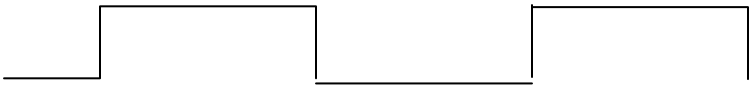
And LOW for 80% of a period, or 80% × 400μs = 320μs

We will use Timer 1, and set it to be in mode 1 (16-bit) as an interval timer.

```
      MOV TMOD, #00010000B    ;initialise Timer 1's mode
LOOP: MOV TL1, #-80           ;timer to start counting –80 counts
      MOV TH1, #00            ; before overflow
HGH:  SETB P1.0               ;set P1.0
      SETB TR1                ;start the timer
      JNB TF1, $              ;wait till it overflows
      CLR TR1                 ;stop the timer
      CLR TF1                 ;clear overflow flag for next usage
LOW:  MOV TL1, #LOW(-320)     ;timer to start counting –320
      MOV TH1, #HIGH(-320)    ; counts before overflow
      CLR P1.0                ;clear P1.0
      SETB TR1                ;start the timer
      JNB TF1, $              ;wait till it overflows
      CLR TR1                 ;stop the timer
      CLR TF1                 ;clear the overflow flag
      SJMP LOOP               ;loop back and repeat
      END
```

13.
```
LOOP: CPL P1.0
      SJMP LOOP
      END
```

Assume P1.0 is initially LOW. Then the waveform is:



CPL SJMP CPL SJMP CPL SJMP CPL

Using an external crystal frequency of 16MHz, then this causes the internal 8051 clock frequency of 1/12 × 16MHz = 1.3333MHz. Hence, one machine cycle would be 1/1.3333MHz = 0.75μs. The waveform generated by the program above takes 6 machine cycles for each period, hence the period, T = 6 × 0.75μs = 4.5μs. This gives a frequency, f = 1/T = 0.2222MHz = 222.22kHz.

As can be seen from the waveform, the durations for both the HIGH and LOW times are the same, hence it has a duty cycle of 50%.

14. 1 hour = 60 minutes = 3600 seconds and 3600 × 1000000 microseconds. In this case we would prefer to use mode 1 for a 16-bit interval timer, and then loop it as many times as needed to generate the duration of 1 hour.

15. f = 21kHz, hence one period is T = 1/f ≈ 48μs.
To generate a waveform with a 10% duty cycle, this means that the waveform would only be

HIGH for 10% of a period, or 10% × 48μs ≈ 5μs
And LOW for 90% of a period, or 90% × 48μs = 43μs

Since the counts are less than 256, we will use a timer set to mode 2 (8-bit auto reload) as an interval timer.

16. The highest that an 8051 timer could count to is $2^{16} - 1 = 65535$ since its count is only a 16-bit value. When used as an event counter, this limits the number of events that the timer could count to just 65535. Counting further causes the count to recycle to zero.

17. A 16-bit timer is one that has a count of 16 bits in size, so the maximum count that it could go to is $2^{16} - 1 = 65535$ before it recycles to zero.

18. (a) f = 3kHz, hence one period is T = 1/f ≈ 0.000333s = 333μs.
To generate a waveform with a 30% duty cycle, this means that the waveform would only be

HIGH for 30% of a period, or 30% × 333μs ≈ 100μs
And LOW for 70% of a period, or 70% × 333μs ≈ 233μs

We will use Timer 1, and set it to be in mode 2 (8-bit) as an interval timer.

```
      MOV TMOD, #00100000B    ;initialise Timer 1's mode
LOOP: MOV TH1, #-100          ;timer to start counting
                              ; –100 counts before overflow
      SETB TR1                ;start the timer
HGH:  SETB P2.0               ;set P2.0
      JNB TF1, $              ;wait till it overflows
      CLR TF1                 ;clear overflow flag for next usage
LOW:  MOV TH1, # -233         ;timer to start counting
                              ; –233 counts before overflow
      CLR P2.0                ;clear P2.0
      JNB TF1, $              ;wait till it overflows
      CLR TF1                 ;clear the overflow flag
      SJMP LOOP               ;loop back and repeat
      END
```

(b) Yes. The time that it is high should be 30% × 333.3333μs = 100μs which does not cause much round-off error. However, the low time should be 70% × 333.3333μs = 233.3333μs but it was rounded off to the nearest integer which is 233. Therefore, the percentage round-off error is (233.333 – 233)/233.333 × 100% = 0.1428%.

Chapter 5 - Serial Port Operation

1.

```
OUTSTR: CLR     A               ;clear A (no offset)
        MOVC    A,@A+DPTR       ;get a character
        JZ      EXIT            ;if null, finished
        ACALL   OUTCHR          ;otherwise, send it
        INC     DPTR            ;increment pointer
        SJMP    OUTSTR          ;repeat
EXIT:   RET
```

This subroutine appears exactly in this form in the listing for MON51 (see Appendix G). The instruction "CLR A" is needed because the following MOVC instruction uses the Accumulator as an offset. No offset is used, so the Accumulator must be explicitly cleared.

2. INLINE: MOV R0,#50H ;initialize R0 as pointer

```
MORE:   ACALL   INCHAR          ;get a character
        MOV     @R0,A           ;put it in the buffer
        INC     R0              ;increment pointer
        CJNE    A,#0DH,MORE     ;carriage return?
                                ; no: get another char.
        MOV     @R0,#0          ;yes: add null byte and
        RET                     ; return to main program
```

A more sophisticated version of INLINE appears in the listing for MON51 (see Appendix G). The solution above performs only the essential task of getting characters from the keyboard and placing them in a buffer. In practice, it is also necessary to check for back-space (or delete) characters to allow for command-line editing. Buffer overflow and underflow conditions should also be guarded against.

3.

```
START:  MOV     A,#'a'
AGAIN:  ACALL   OUTCHR
        INC     A
        CJNE    A,#'z'+1,AGAIN
        SJMP    START
```

4.

```
START:  MOV     A,#20H          ;initialize A
AGAIN:  ACALL   OUTCHR          ;send character
        INC     A               ;next code > 7EH?
        CJNE    A,#7FH,AGAIN    ; no: sent it
        SJMP    START           ;yes: reset to 20H
```

5.

```
XOFF    EQU     13H             ;control-S
XON     EQU     11H             ;control-Q
START:  MOV     A,#20H
AGAIN:  ACALL   OUTCHR
        JNB     RI,SKIP         ;if no character waiting,
                                ; carry on as before
        PUSH    ACC             ;if character waiting,
        ACALL   INCHAR          ;save A and get character
        CJNE    A,#XOFF,SKIP2   ;if not XOFF, ignore
CHECK:  ACALL   INCHAR          ;if XOFF, wait for XON
        CJNE    A,#XON,CHECK    ;check until XON received
SKIP2:  POP     ACC             ;restore A and carry on
SKIP:   INC     A
        CJNE    A,#7FH,AGAIN
        SJMP    START
```

6.

```
START:  ACALL   INCHAR
        CJNE    A,#'a',$+3
        JC      SKIP
        CJNE    A,#'z'+1,$+3
        JNC     SKIP
        CLR     ACC.5
SKIP:   ACALL   OUTCHR
        SJMP    START
```

The problem is to identify whether or not the character read is within the range a-z inclusive. The two CJNE instructions are used strictly to set or clear the carry flag based on the compare; so in both cases the jump destination is "$+3"; that is, the next instruction.

The first CJNE instruction tests the character against 'a'. The carry is set for values less than 'a', in which case the character is echoed as is. The following instruction (JC) jumps directly to SKIP and the character is loaded in SBUF for transmission. The second CJNE is reached only if the character code is greater-than-or- equal-to 'a'. The test is against 'z'+1 and sets the carry flag if the character is 'z' or lower. JNC SKIP is the appropriate condition for echoing the character as is, since the code is numerically higher than 'z'.

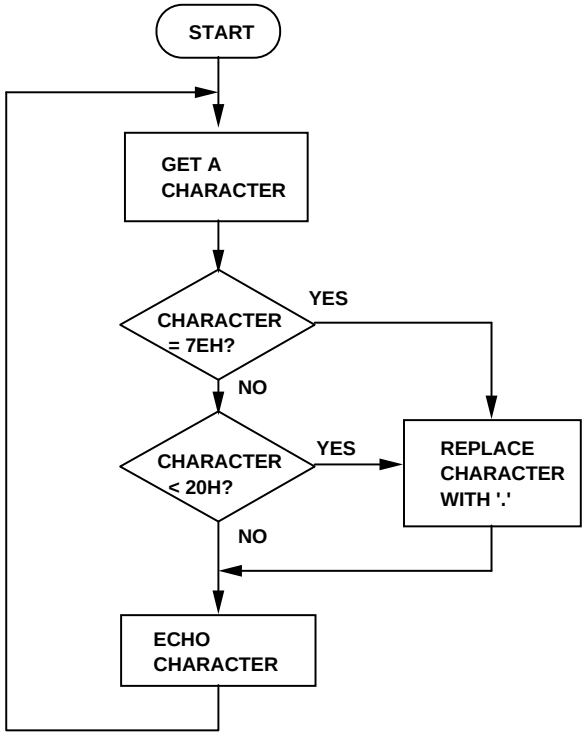
Finally, the CLR instruction is reached only if the character is lowercase alpha (a-z). Clearing bit 5 converts the character to uppercase before it is echoed. As an example, compare the lowercase and uppercase ASCII codes for 'a':

```
01100001B = 61H = 'a' (bit 5 set)
01000001B = 41H = 'A' (bit 5 clear)
```

7.

```
START:  ACALL   INCHAR          ;get a character
        CJNE    A,#7EH,SKIP     ;char = 7EH?
        SJMP    CHANGE          ;yes: change to '.'
SKIP:   CJNE    A,#20H,$+3      ; no: char < 20H?
        JNC     ECHO            ;      no: echo as is
CHANGE: MOV     A,#'.'          ;     yes: change to '.'
ECHO:   ACALL   OUTCHR          ;echo character
        SJMP    START           ;repeat
```

Flowchart:



8.

```
COUNT   EQU     10
ESC     EQU     1BH
CR      EQU     0DH
LF      EQU     0AH
START:  MOV     DPTR,#CLR       ;point to clear screen
        ACALL   OUTSTR          ;send codes to VDT
        MOV     R7,#COUNT       ;initialize counter
LOOP:   MOV     DPTR,#MESSGE    ;point to message
        ACALL   OUTSTR          ;send message to VDT
        DJNZ    R7,LOOP         ;repeat until count = 0
        SJMP    $               ;done
CLR:    DB      ESC,'[2J',0     ;codes for clear screen
MESSGE: DB      'Scott MacKenzie',CR,LF,0 ;message codes
```

9.

```
          MOV     SCON,#0          ;mode 0, shift register
LOOP:     MOV     SBUF,20H
          ACALL   DELAY
          SJMP    LOOP
DELAY:    MOV     R7,#3
DELAY3:   MOV     DPTR,#-50000
DELAY2:   INC     DPTR             ;2 cycles
          MOV     A,DPH            ;1
          ORL     A,DPL            ;1
          JNZ     DELAY2           ;2 (6 ¥ 55,555 = 333,333)
          DJNZ    R7,DELAY3
          RET
```

10. The 8051 supports full-duplex serial communication because it allows simultaneous transmission and reception. The serial port can be thought of to consist of the serial port buffer, SBUF and a parallel-to-serial shift register. The SBUF is actually two physically different registers, one for transmission and the other for reception. Even though they are called by the same name, writing to SBUF during transmission uses the first register while reading from SBUF during reception uses the second register, so this avoids any data clashes, hence allowing full-duplex serial communication.

11. We set the serial port to operate in mode 0, making it act as a parallel-to-serial shift register. We connect the TXD line to the CLK input of an external serial-to-parallel shift register, and the RXD line to the DATA input of that external shift register. This enables us to have 8 extra I/O ports at the expense of using the serial port. The disadvantage is that it takes longer since there is parallel-to-serial and then serial-to-parallel conversion, and also since transmission of data is done serially through the serial port.

12.

```
OUTCHR9:    PUSH  SCON        ;backup SCON
            MOV   TB8, B.0    ;move 9th bit to TB8
            JNB   TI, $       ;wait for end of previous Tx
            CLR   TI          ;clear TI flag
            MOV   SBUF, A     ;start transmission
            POP   SCON        ;restore SCON values
            RET               ;return
            END
```

13.

```
            ORG 8100H
INCHAR8:    JNB RI, $         ;wait for character
            CLR RI            ;clear flag
            MOV A, SBUF       ;read char into A
            MOV C, P          ;for odd parity in A,
            CPL C             ; P should be complemented
            JC CHECK          ;If C = 1, check RB8
            JB RB8, ERROR     ;C = 0, RB8 = 1, parity error!
            SJMP DONE         ;C = 0, RB8 = 0, no parity error
CHECK:      CLR C             ;C = 1 previously, and is not a
```

```
                                ; parity, then error message
            JB RB8, DONE        ;C = 1, RB8 = 1, so no parity error
ERROR:      SETB C              ;C = 1, RB8 = 0, parity error!
DONE:       RET
            END
```

14.
```
OUTCHR8:    MOV C, P            ;put parity bit in C flag
            CPL C               ;change to odd parity
            MOV   TB8, C        ;store parity bit in TB8 as 9th bit
            JNB   TI, $         ;wait for end of previous Tx
            CLR   TI            ;clear TI flag
            MOV   SBUF, A       ;start transmission
            RET                 ;return
            END
```

15. (a)

SCON:	1	1	0	1	0	0	1	0
TMOD:	0	0	1	0	0	0	0	0
TCON:	0	1	0	0	0	0	0	0
TH1:	1	1	1	1	1	1	0	1

```
            ORG 8100H
INIT: MOV SCON, #D2H
      MOV TMOD, #20H
      MOV TH1, # – 3
      SETB TR1
      END
```

BR $= T_{1(ovf)} \div 32$

$T_{1(ovf)} = BR \times 32$

$= 9600 \times 32$

$= 307.2\text{kHz}$

Timer 1 is clocked at a rate of 1MHz = 1000kHz, meaning an overflow is required every 1000/307.2 ≈ 3 clocks (round to 3). This means that we need to initialize Timer 1 to start counting 3 counts prior to overflow, which is FDH or – 3.

(b) No. The actual BR = $BR_{actual} = T_{1(ovf)actual} \div 32 = 333.33\text{kHz} \div 32 = 10416.67$ baud.

$$\%\ \text{error} = \frac{BR_{actual} - BR_{desired} \times 100\%}{BR_{desired}}$$

$$= \frac{10416.67 - 9600 \times 100\%}{9600} = 8.5\%$$

Chapter 6 - Interrupts

1.

```
        ORG     0
        LJMP    MAIN
        ORG     000BH
T0ISR:  CPL     P1.0
        JNB     RI,EXIT         ;keyboard input?
        CLR     EA              ;yes: shut off interrupts
EXIT:   RETI                    ;     before returning
        ORG     0030H
MAIN:   MOV     SMOD,#52H       ;init serial port as
        MOV     TH1,#0F3H       ; 2400 baud UART
        MOV     TH0,#-50
        MOV     TMOD,#22H
        SETB    TR1             ;start baud rate clock
        SETB    TR0             ;start interrupt timer
        MOV     IE,#82H
        SJMP    $
```

Two changes have been introduced in the example in the text. First, since the program is assumed "stand alone", instructions to initialize the serial port have been added. The same initialize sequence is used as in MON51 (see Appendix G in the text).

Second, the timer interrupt service routine now checks the serial port status as well as complementing P1.0. If RI = 0, no character has been entered on the keyboard and the ISR returns to the main program, as before. If a character has been entered on the keyboard, RI will be set. The Enable All (EA) bit in the Interrupt Enable register is cleared before returning to the main program. This will prevent further interrupts from being processed (even though Timer 0 continues to run).

2.

```
COUNT   SET     -500            ;500 ms high, 500 ms low
ADJUST  EQU     16              ;adjustment for overhead
COUNT   SET     COUNT + ADJUST  ; cycles, shown as (x)
        ORG     0               ;stand alone operation
        LJMP    MAIN            ;begin at MAIN
        ORG     000BH           ;Timer 0 interrupt entry
        LJMP    T0ISR           ;(2) execute ISR
        ORG     0030H           ;MAIN entry point
MAIN:   MOV     TMOD,#01H       ;Timer 0, 16-bit mode
        SETB    TF0             ;force first interrupt
        MOV     IE,#82H         ;enable Timer 0 interrpts
        SJMP    $               ;(2+2) now do nothing
T0ISR:  CPL     P1.7            ;(1) create square wave
        CLR     TR0             ;(1) stop timer
        MOV     TH0,#HIGH COUNT ;(2) initialize count
        MOV     TL0,#LOW COUNT  ;(2)
        SETB    TR0             ;(1) start timer
        RETI                    ;(2) return to main
```

Since a 1 kHz square wave has high- and low-times of 500 μs each, 16-bit timer mode is required. The error introduced in re-initializing the timer after each interrupt is shown (in machine cycles) in parentheses following the relevant instructions. The SJMP overhead is indicated as "2+2" to allow for the execution time of the instruction and the implicit CALL instruction that executes as part of interrupt acceptance.

Sixteen cycles are added to COUNT as an adjustment. The SET directive sets COUNT to -500 initially, and then to COUNT + ADJUST. Recall that SET is the same as EQU except that values can be re-assigned using a subsequent SET.

3.

```
HCOUNT  EQU     -43     ;high-time (ms) unadjusted
LCOUNT  EQU     -100    ;low-time (ms) unadjusted
        ORG     0
        LJMP    MAIN
        ORG     000BH
        LJMP    T0ISR
        ORG     0030H
MAIN:   MOV     TMOD,#01H       ;16-bit timer
        SETB    TF0             ;force first interrupt
        MOV     IE,#82H         ;enable Timer 0 interrupt
        SJMP    $               ;nothing left to do
T0ISR:  CLR     TR0             ;stop timer
        JBC     P1.6,SKIP       ;nice use of "JBC" here!
        SETB    P1.6            ;begin high-time
        MOV     TH0,#HIGH HCOUNT
        MOV     TL0,#LOW HCOUNT
        SJMP    EXIT
SKIP:   MOV     TH0,#HIGH LCOUNT ;begin low-time
        MOV     TL0,#LOW LCOUNT
EXIT:   SETB    TR0             ;turn timer on and
        RETI                    ; return to SJMP
```

A 7 kHz waveform has a period of approximately 143 μs. For a 30% duty cycle, the required high- and low-times are approximately 43 μs and 100 μs respectively.

4.

```
BAUD    EQU     -26             ;count for 1200 baud
COUNT   EQU     -50             ;count for 10 kHz
        ORG     0
        LJMP    MAIN
        ORG     0BH             ;Timer 0 entry point
        CPL     P1.0            ;do it here!
        RETI                    ;done
        ORG     23H             ;Serial Port entry point
        LJMP    SPISR
        ORG     30H
MAIN:   MOV     TMOD,#22H       ;Mode 2 (both timers)
        MOV     TH1,#BAUD       ;Timer 1 for baud rate
        MOV     TH0,#COUNT      ;Timer 0 for 10 kHz
        SETB    TR1             ;start baud rate clock
        SETB    TR0             ;start waveform clock
        MOV     SCON,#42H       ;Mode 1 (Serial Port)
        MOV     A,#20H          ;ASCII "space" char.
        MOV     IE,#92H         ;enable Serial Port &
                                ; Timer 0 interrupts
        SJMP    $               ;that's it!
SPISR:  CJNE    A,#7FH,SKIP     ;code past DEL char.?
        MOV     A,#20H          ;yes: reset to SPACE
SKIP:   MOV     SBUF,A          ;send code to VDT
        INC     A               ;inc A to next code
        CLR     TI              ;clear tx interrupt flag
        RETI                    ;return to SJMP
```

This problem merges the programs from two examples in the text. The instructions which initialize TMOD and IE now set the bits appropriate for each example. Otherwise, the program above is identical to those in the text.

5.

```
COUNT   EQU     -50000  ;50,000 ms
COUNT2  EQU     20      ;  ¥ 20 = 1 second
SPACE   EQU     20H     ;ASCII codes
DEL     EQU     7EH
        ORG     0
        LJMP    MAIN
        ORG     000BH
        LJMP    T0ISR
        ORG     0030H
MAIN:   MOV     TMOD,#21H       ;Timer 1 =  8-bit mode
                                ;Timer 0 = 16-bit mode
        MOV     TH1,#-26        ;1200 baud reload count
        SETB    TR1             ;start baud rate clock
        MOV     SCON,#42H       ;initialize serial port
        MOV     A,#SPACE        ;put SPACE code in A
        MOV     R7,#COUNT2      ;use R7 as counter
        MOV     IE,#82H         ;enable Timer 0 interrpts
        SJMP    $               ;That's all folks!
T0ISR:  CLR     TR0             ;stop timer
        MOV     TH0,#HIGH COUNT ;(re)initialize count
        MOV     TL0,#LOW COUNT
        DJNZ    R7,EXIT         ;every 20 interrupts ...
        MOV     SBUF,A          ; send ASCII code &
        INC     A               ; increment to next code
        CJNE    A,#DEL + 1,EXIT ; if DEL just sent,
        MOV     A,#SPACE        ;  reset code to SPACE
EXIT:   SETB    TR0             ;turn time on &
        RETI                    ; return to MAIN
```

The solution above assumes transmission of the previous character has completed and that the serial port is ready for the next character. The RI bit, therefore, is not checked, nor is it cleared as is normally the case. This should not pose a problem; however, a reasonable alternative is to check RI and include a conditional jump instruction to an error routine in the event it is not set one second after the previous character was loaded for transmission.

6. Actually, since Example 6–5 is an interrupt-enabled program, so whenever a high-to-low transition will trigger the external 0 (EXT 0) interrupt, which resets the repetition counter, R7 to 20, hence restarting the timing loop to sound for another second. Therefore, there is no need to modify Example 6.5. Note that this is different from problem 5 of Chapter 4 which required to rewrite the solution to Example 4–7 to include a “restart” mode since it did not make use of interrupts and hence the program has to wait in a timing loop while the buzzer is sounding.

7. By default, when two interrupts occur simultaneously, they will be serviced based on polling, meaning INT0 will be serviced first since it will be polled before the Timer 0 interrupt.

8. Initialize the IP = xxx1x0xx, where x denotes “don’t care” values. By setting the interrupt priority bit of the serial port interrupt to 1, and the interrupt priority bit of INT1

to 0, the serial port interrupt would have a higher priority and hence would be serviced first.

9. Notice that the IE register has been set such that only the two external interrupts and the two timer interrupts are enabled, hence a serial port interrupt will never happen. Also, the IP register states that INT1 and Timer 0 interrupts are at the same HIGH priority, so they will be serviced based on polling, meaning Timer 0 will be serviced first since it will be polled before INT1.

10. A small ISR is one where all its instructions can fit into the allocated 8 bytes between subsequent ISR vector addresses. A large ISR is one that does not, so a jump elsewhere is needed in order to ensure that the instructions do not overflow over to the next ISR.

11.
```
ORG   0000H
LJMP  MAIN
ORG   0013H
SETB  P1.0
MOV   R0, #250
DJNZ  R0, $
CLR   P1.0
RETI
ORG   0030H
SETB  TCON.2
MOV   IE, #100000100B
SJMP  $
END
```

Chapter 7 - Assembly Language Programming

1.

```
MOV     A,#11111111B
MOV     A,#00001001B
MOV     A,#00011010B
MOV     A,#01000001B
```

2.

```
MOV     A,#FFH
MOV     A,#FCH
MOV     A,#7AH
MOV     A,#1BH
MOV     A,#24H
MOV     A,#40H
```

3. The data constant F0H should be coded 0F0H, otherwise the assembler will attempt to interpret it as a symbol (because the first character is a letter).

4.

?byte.bit	"?" and "." are illegal characters
@GOOD_bye	symbols must begin with a letter
1ST_FLAG	symbols must begin with a letter
MY-PROGRAM	"-" is an illegal character

5.

```
MOV     DPTR,#0FFFFH
MOV     DPTR,#4100H
MOV     DPTR,#0FFFFH
MOV     DPTR,#0FFFEH
```

8.

CODE	code segment
XDATA	external data segment
DATA	internal data segment accessible by indirect addressing
IDATA	entire internal data segment (accessible by direct and indirect addressing)
BIT	bit segment (overlapping byte locations 20H-2FH)

9.

```
OFFCHIP  SEGMENT XDATA
         RSEG   OFFCHIP
XBUFFER: DS     100
```

10.

```
        BSEG    AT 08H
FLAG1   DBIT    1
FLAG2   DBIT    1
FLAG3   DBIT    1
FLAG4   DBIT    1
FLAG5   DBIT    1
```

11. Equating symbols to values using the EQU directive improves the readability of programs and makes them easier to change.

    Programs become more readable because the body of the program contains the symbol rather than the value of the symbol. Thus we might find "COUNT" rather than "75". The former immediately gives us a clue as to the purpose of the instruction.

    Programs which make generous use of equated symbols are easier to change because the symbol, although used many times, is equated only once. If a change is necessary only the line containing the EQU directive needs changing, rather than every line where the symbol is used. When the program is re-assembled the new value is automatically substituted for each occurrence of the symbol.

12. DB (define byte) and DW (define word) differ only in that the each data constant is defined as an 8-bit result for DB; whereas, it is defined as a 16-bit result for DW.

13.

```
MEMORY  DATA
------------
000F    00
0010    F0
0011    FF
0012    00
0013    30
```

    The expression in the first DW statement is converted to 00F0. The expression in the DB statement is FFFF in hexadecimal; however, only the lower 8 bits are used (since the directive is define byte). The last DW directive is converted to 0030, which is the ASCII code for "0" (expressed as a 16-bit value).

14. CSEG

15. $INCLUDE(ASCII)

    Note that the definitions in the file ASCII will appear in the listing of the assembled file. This can be prevented as follows:

```
$NOLIST
$INCLUDE(ASCII)
$LIST
```

16. Put $EJECT on the line preceding each subroutine.

17.

```
%*DEFINE(FILL(VALUE, XADD, COUNT)) LOCAL LOOP SKIP EXIT
        (MOV    DPTR,#%XADD
%LOOP:  MOV     A,DPH
        CJNE    A,#HIGH(%XADD + %COUNT),%SKIP
        MOV     A,DPL
        CJNE    A,#LOW(%XADD + %COUNT),%SKIP
        SJMP    %EXIT
%SKIP:  MOV     A,#%VALUE
        MOVX    @DPTR,A
        INC     DPTR
        SJMP    %LOOP
%EXIT:  )
```

This macro will work for any count, including zero. The terminating condition is tested before VALUE is written to external memory; so if COUNT = 0 no writing takes place.

18.

```
%*DEFINE(JGE(VALUE, LABEL))
        (CJNE   A,#%VALUE,$+3
        JNC     %LABEL
        )

%*DEFINE(JLT(VALUE, LABEL))
        (CJNE   A,#%VALUE,$+3
        JC      %LABEL
        )

%*DEFINE(JLE(VALUE, LABEL))
        (CJNE   A,#%VALUE+1,$+3
        JC      %LABEL
        )

%*DEFINE(JOR(LOWER, UPPER, LABEL))
        (CJNE   A,#%LOWER,$+3
        JC      %LABEL
        CJNE    A,#%UPPER+1,$+3
        JNC     %LABEL
        )
```

These macros appear in the listing of the monitor program MON51 (see Appendix G in the text).

19.

```
%*DEFINE(CJNE_DPTR(VALUE, LABEL)) LOCAL EXIT
        (PUSH   ACC
        MOV     A,DPH
        CJNE    A,#HIGH %VALUE,%EXIT
        MOV     A,DPL
        CJNE    A,#LOW %VALUE,%EXIT
        POP     ACC
        LJMP    %LABEL
%EXIT:  POP     ACC
        )
```

Since "CJNE direct,#data,rel" does not exist (see Appendix A), the contents of DPH and DPL had to be moved into the Accumulator before testing against VALUE. PUSH and POP instructions are needed to ensure that the previous content of the Accumulator is not destroyed.

Chapter 8 - 8051 C Programming

1. The 8051 C language is a derivative of the conventional C language. It supports all features of conventional C, in addition to additional data types, memory types and memory models specially suited for the 8051. It also allows for the programmer to write special interrupt functions. Another difference between the two is that while source codes for the latter are compiled into executable form for use in computers, source codes for the latter are compiled into executable form for use on the 8051.

2.
```
unsigned char bdata temp _at_ 0x24;
sbit tempBit = temp ^ 0;
```

3. Used to refer to a certain bit position within a bit-addressable location.

4. We can declare sbit variables in 3 ways:

   - `sbit P = 0xD0;`
     Directly specifying the bit address that it should be referring to.

   - `sfr PSW = 0xD0;`
     `sbit P = PSW ^ 0;`
     First declaring a bit-addressable byte variable and then using that as the base address to refer to a certain bit within it

   - `sbit P = 0xD0 ^ 0;`
     Directly declaring it as a certain bit within a bit-addressable location

5. An SFR variable is an 8-bit variable used to refer to a special function register (SFR) in the 8051. An unsigned char variable is an 8-bit variable that may reside anywhere and may be used to store any data.

6. Memory types are used to specify where in memory a certain variable should reside. Memory models are used to specify the default memory type in which all variables in the program should reside.

7. A function is a group of C statements that perform a related task, and that appear frequently in the program. The function would be executed whenever it is called from the main program. An interrupt function is a special type of function that will only be called if an interrupt is triggered.

8. A pointer is a special type of variable that stores not a value but an address. It can then be used to indirectly point to the location specified by that address. We can thus say that a pointer is pointing to that location.
   Example:
```
int num = 0;
int * numPtr = &num;
```

9. We use the **large** memory model for the entire program so that all variables are by default residing in external ram. Meanwhile, for the certain variables that need to be in internal ram, we use the **data** memory type specifier when declaring these variables.

10. Because the 8051 automatically uses the ACC almost every machine cycle, so it is most probable that a C statement that you write would be compiled into assembly language instructions that automatically make use of and hence affect the contents of ACC.

11.
```
unsigned char xdata * idata DPTR;   /* DPTR in data, points to */
                                    /*    xdata                */
unsigned char bdata A;              /* represents ACC */
unsigned char code numbers[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
sbit inbit = P1^0;                  /* input bit */
int i;

main()
{
DPTR = 0x1234;    /* DPTR points to external location 1234H */
P1 = 0xFF;        /* let P1 be input */
if (inbit==1)     /* if input bit is 1, copy odd numbers */
      for (i = 0; i < 10; i=i+2)
            *DPTR++=numbers[i];
else              /* if input bit is 0, copy even numbers */
      for (i = 1; i < 10; i=i+2)
            *DPTR++=numbers[i];
}
```

12.
```
unsigned char data * sPtr = 0x30;   /* point to source */
unsigned char xdata * dPtr = 1234;  /* point to destination */
int i = 0;


main()
{
      while(1)                      /* repeat forever
            {
            dPtr[i] = sPtr[i];      /* from source to dest */
            i++;                    /* point to next location */
            }
      }
```

13. (a)
```
float code F[10] = {20.1, 54.6, 148.4, 403.4, 1096.6, 2981,
      8103.1,220265.5, 59874.1, 162754.8};
```

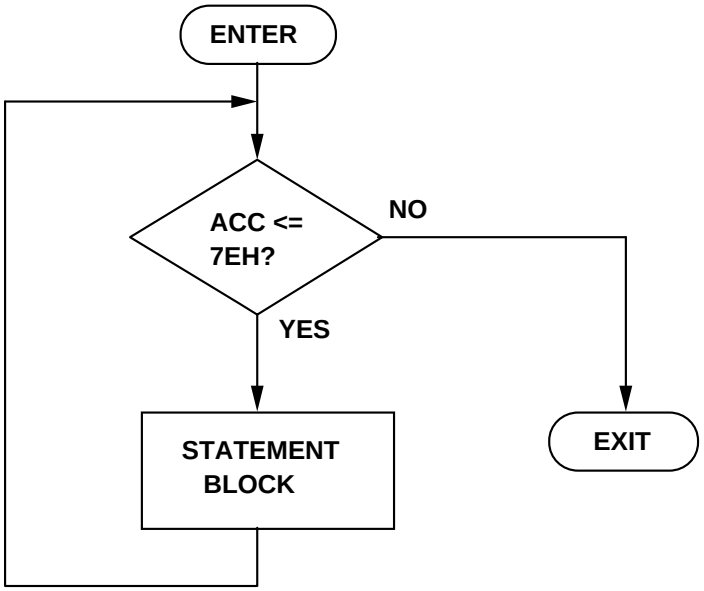
(b)
```
int code * idata FPtr = code;
printf("%f", FPtr[1]);
```

Chapter 9 - Program Structure and Design

1. Pseudo Code:

```
WHILE [ACC <= 7EH] DO BEGIN
        [statement]
        .
        .
        .
END
```

Flowchart:



8051 Code:

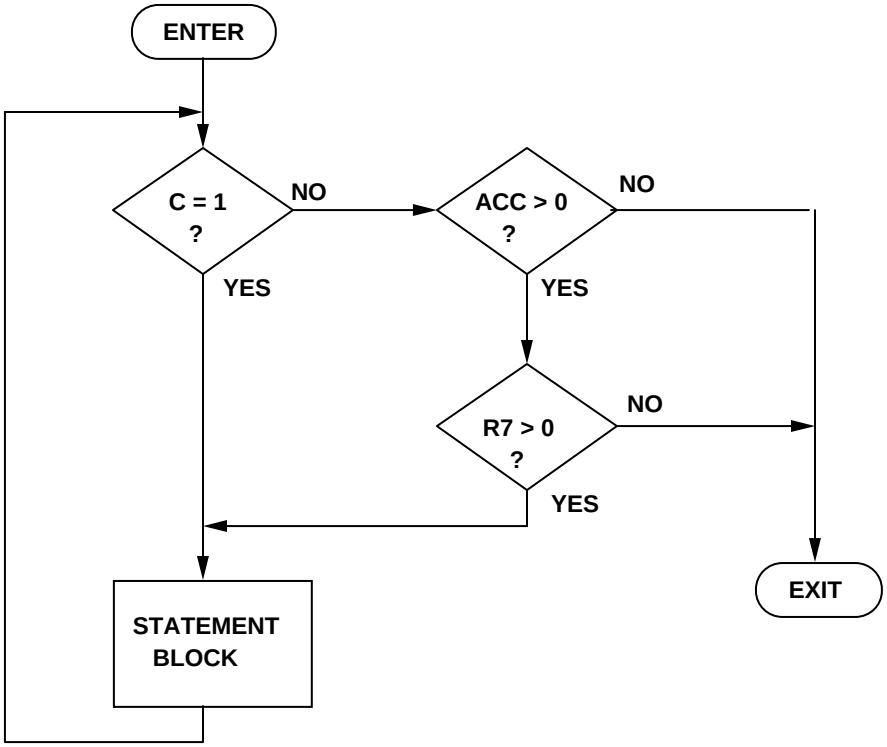
```
ENTER:  CJNE    A,#7EH+1,$+3
        JNC     EXIT
STMENT: (one or more statements)
        .
        .
        .
        JMP     ENTER
EXIT:   (continue)
```

2.

Pseudo Code:

```
WHILE [(ACC > 0 AND R7 > 0) OR C == 1] DO
        [statement]
```

Flowchart:



8051 Code:

```
ENTER:  JC      STMENT
        JZ      EXIT
        CJNE    R7,#0,STMENT
        SJMP    EXIT
STMENT: (one or more statements)
        .
        .
        .
        SJMP    ENTER
EXIT:   (continue)
```

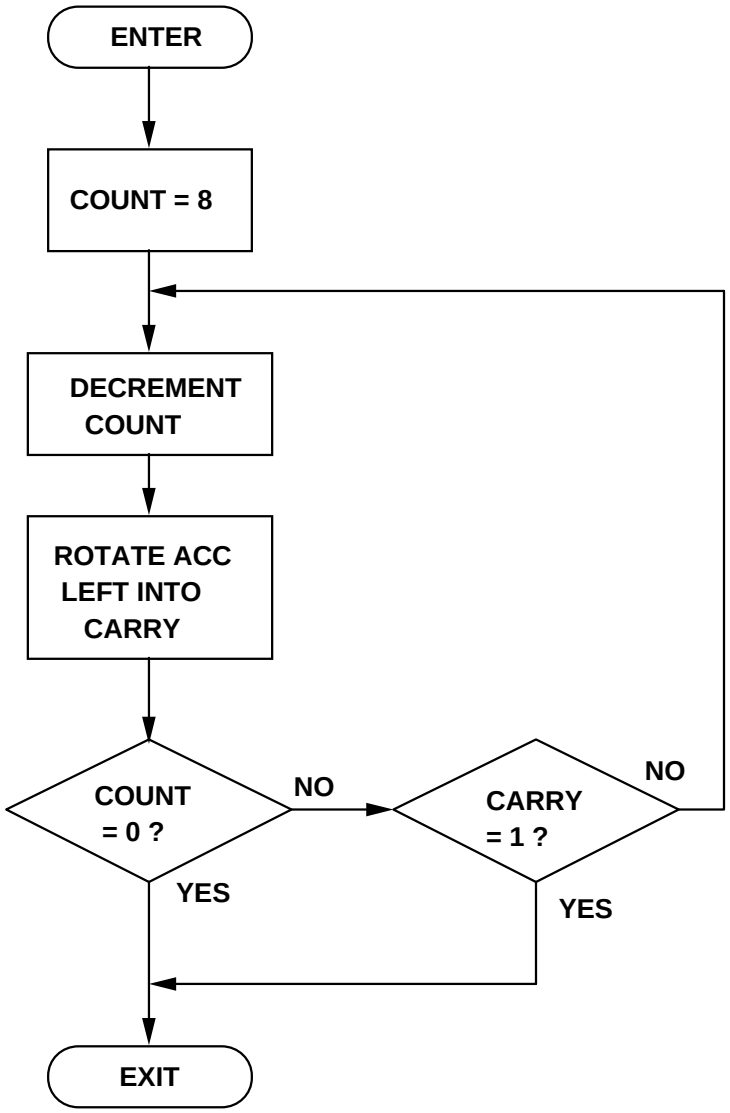
The solution above uses three consecutive conditional jump instructions. Getting the "jump" and "no jump" conditions right is tricky; however, solving the problem first in Pseudo Code and/or a flowchart is helpful. Have students focus on the conditions needed to enter the statement block vs. the conditions needed to exit the structure. The use of labels such an "STATEMENT" and "EXIT" helps (at least initially).

3.

Pseudo Code:

```
[COUNT = 8]
REPEAT
        [decrement COUNT]
        [rotate ACC left through carry]
UNTIL [carry == 1 OR COUNT == 0]
```

Flowchart:



8051 Code:

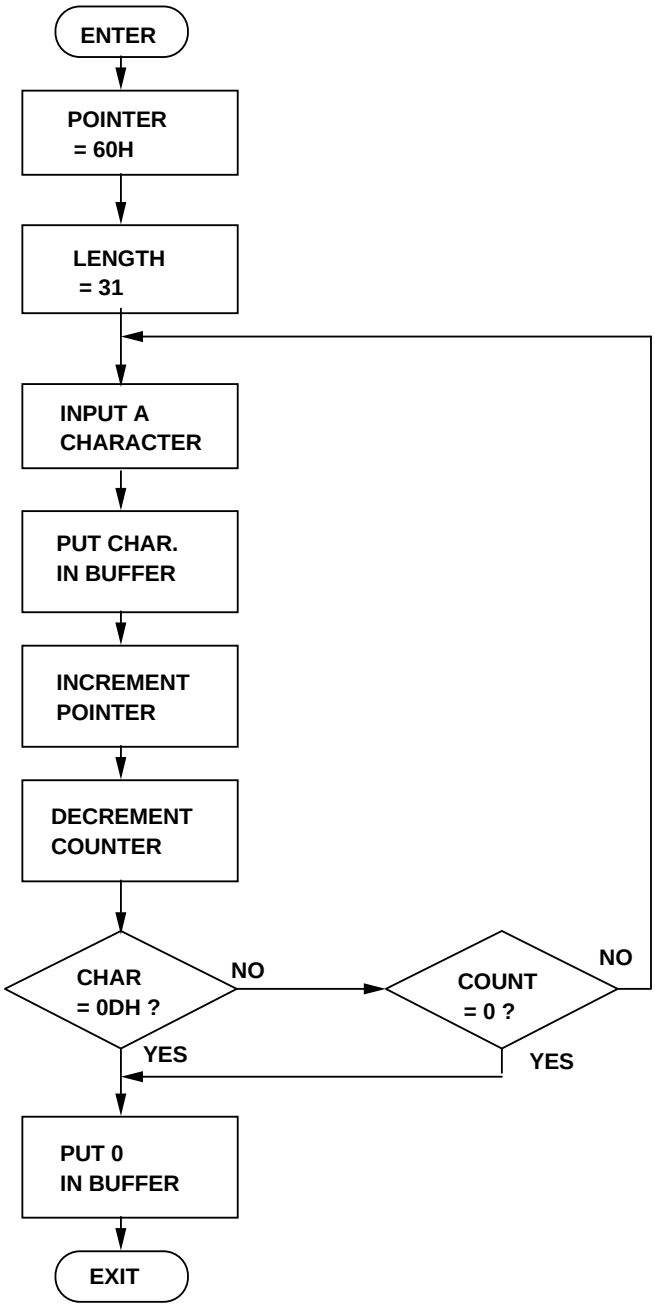
```
FIND:   MOV     R7,#8
STMENT: DEC     R7
        RRC     A
        JC      EXIT
        CJNE    R7,#0,STMENT
EXIT:   RET
```

4.

Pseudo Code:

```
[pointer = 60H]
[length = 31]
REPEAT
        [input character]
        [@pointer = character]
        [increment pointer]
        [decrement length]
UNTIL [character == 0DH OR length == 0]
[@pointer = 0]
```

Flowchart:


ENTER
POINTER = 60H
LENGTH = 31
INPUT A CHARACTER
PUT CHAR. IN BUFFER
INCREMENT POINTER
DECREMENT COUNTER
CHAR = 0DH ?
NO
COUNT = 0 ?
NO
YES
YES
PUT 0 IN BUFFER
EXIT

8051 Code:

(closely structured; 20 bytes)

```
INLINE: MOV     R0,#60H         ;2 bytes
        MOV     R7,#31          ;2
STMENT: ACALL   INCH            ;2
        MOV     @R0,A           ;1
        INC     R0              ;1
        DEC     R7              ;1
        CJNE    A,#0DH,SKIP     ;3
        SJMP    EXIT            ;2
SKIP:   CJNE    R7,#0,STMENT    ;3
EXIT:   MOV     @R0,#0          ;2
        RET                     ;1
```

(loosely structured; 18 bytes)

```
INLINE: MOV     R0,#60H         ;2 bytes
        MOV     R7,#31          ;2
STMENT: ACALL   INCH            ;2
        MOV     @R0,A           ;1
        INC     R0              ;1
        CJNE    A,#0DH,SKIP     ;3
EXIT:   MOV     @R0,#0          ;2
        RET                     ;1
SKIP:   DJNZ    R7,STMENT       ;2
        SJMP    EXIT            ;2
```

Chapter 10 - Tools and Techniques for Program Development

1. (a) 307 units

The production costs ($) using the mask-programmed 8051 for N devices is

```
3N + 10,000
```

The production costs using the 8751 EPROM for N devices is

```
30N
```

The break-even point for the two approaches is

```
3N + 10,000 = 30N
        27N = 10,000
          N = 307 units
```

For production runs above 307 units, the mask-programmed 8051 is more cost-effective than the 8751 EPROM.

(b) $71,000

The cost for 3,000 8751 EPROMs is

```
3,000 x 30 = $90,000
```

The cost for 3,000 mask-programmed 8051s is

```
3 x 3,000 + 10,000 = $19,000
```

The savings using 8051s instead of 8751s is

```
90,000 - 19,000 = $71,000
```

2. (a) 0800

(b) $10_{16} + 10_{16} + 10_{16} + 0A_{16} = 3A_{16} = 58_{10}$ bytes

(c) 083A

3. 7F (Add all bytes except checksum. Subtract from 0 to get checksum)

4.

```
        ORG     0
        MOV     R0,#20H
LOOP:   MOV     @R0,#55H
        INC     R0
        CJNE    R0,#80H,LOOP
```

The first two characters in the hex file ("09") indicate that the current record contains nine bytes. The next two characters ("01") indicate a data record. The next four characters ("0000") give the load address for the data bytes that follow. The data bytes begin with the next two characters ("78") which correspond to the opcode of the first instruction in the program.

The easiest way to proceed is by looking-up the opcodes in Appendix B in the text. The form of the instruction is given along with the number of bytes and the number of cycles. For the first instruction, the opcode is 78H. From Appendix B, the instruction is identified as the two-byte instruction

```
MOV     R0,#data
```

The next two characters in the hex file ("20"), therefore, represent the immediate data for this instruction. The complete instruction is

```
MOV     R0,#20H
```

The next two characters ("76") must be the opcode for the next instruction, which is also looked-up in Appendix B, and so on.

It is extremely important not to "get out of sync" when following the procedure above. If the number of bytes in an instruction is misinterpreted, not only will the conversion be wrong for the current instruction, but so too will conversions for subsequent instructions (since the wrong characters will be interpreted as opcodes).

Chapter 11 - Design and Interface Examples in Assembly

1. (sorry, no solution available at press time)

2. (sorry, no solution available at press time)

3. (a) 976 Hz

   Discussion: The loop takes 4 cycles, or 4 microseconds to execute. With STEP = 1, 256 increments are required for each overflow of ACC. The period is 4 x 256 = 1024 microseconds = 1.024 ms. The frequency is 1/1.024 = 0.976 kHz = 976 Hz.

   (b) 10

   (c) F = STEP / 1.024 kHz

4. 05F0H (see MON51.MAP)

5.

```
                        1     $DEBUG
                        2     $PAGEWIDTH(98)
                        3     $TITLE(*** 8155 RAM I/O TIMER - EXAMPLE 5 ***)
                        4     $NOPAGING
                        5     $NOSYMBOLS
                        6     ;*********************************************************************
                        7     ;                                                                    *
                        8     ;                    8155 RIOT INTERFACE TO SBC-51                   *
                        9     ;                                                                    *
                       10     ; Create a 1 kHz square wave on P1.7 using the 8155 timer and        *
                       11     ; External 0 interrupts.                                             *
                       12     ;                                                                    *
                       13     ; SBC-51 JUMPERS:                                                    *
                       14     ;    X3 not installed (MON51 executes)                               *
                       15     ;    X4 not installed (interrupts directed to jump table at 8003H)   *
                       16     ;    X7     installed (8155 TIMER OUT connects to 8031 -INT0)        *
                       17     ;*********************************************************************
                       18
                       19     ;*********************************************************************
                       20     ; SYMBOL DEFINITIONS                                                 *
                       21     ;*********************************************************************
  00BC                 22     MON51   CODE    00BCH           ;MON51 entry point (V12)
  0100                 23     X8155   XDATA   0100H           ;8155 address in external RAM
  0104                 24     TIMER   XDATA   X8155 + 4       ;timer registers (low byte first)
  00C0                 25     START   EQU     0C0H            ;8155 start timer command
  0002                 26     FREQ    EQU     2               ;in kHz (interrupts every 0.5 ms)
  01F4                 27     PERIOD  EQU     1000 / FREQ     ;in microseconds
  03E8                 28     COUNT   EQU     2 * PERIOD      ;8155 clocked by ALE every 0.5 us
  0040                 29     MODE    EQU     01000000B       ;timer mode bits (cont. square wave)
                       30
                       31     ;*********************************************************************
                       32     ; JUMP TABLE (ENTRY POINTS FROM MON51 FOR MAIN PROGRAM AND ISRs)     *
                       33     ;*********************************************************************
8000                   34             ORG     8000H
8000 028015            35             LJMP    MAIN            ; Main program entry
8003 02802D            36             LJMP    EX0ISR          ; External 0 interrupt
8006 028030            37             LJMP    T0ISR           ; Timer 0 interrupt
```

```
8009 028030          38             LJMP    EX1ISR          ; External 1 interrupt
800C 028030          39             LJMP    T1ISR           ; Timer 1 interrupt
800F 028030          40             LJMP    SPISR           ; Serial Port interrupt
8012 028030          41             LJMP    T2ISR           ; Timer 2 interrupt
                     42
                     43     ;*********************************************************************
                     44     ; MAIN PROGRAM BEGINS (INITIALIZE 8155 AND ENABLE INTERRUPTS)       *
                     45     ;*********************************************************************
8015 900104          46     MAIN:   MOV     DPTR,#TIMER     ;initialize 8155 timer low register
8018 74E8            47             MOV     A,#LOW(COUNT)
801A F0              48             MOVX    @DPTR,A
801B A3              49             INC     DPTR            ;initialize high register
801C 7443            50             MOV     A,#HIGH(COUNT) OR MODE
801E F0              51             MOVX    @DPTR,A
801F 900100          52             MOV     DPTR,#X8155     ;start timer
8022 74C0            53             MOV     A,#START
8024 F0              54             MOVX    @DPTR,A
8025 D2AF            55             SETB    EA              ;enable interrupts (in general)
8027 D2A8            56             SETB    EX0             ;enable External 0 interrupt
8029 D288            57             SETB    IT0             ;negative-edge triggered
802B 80FE            58             SJMP    $               ;DONE!
                     59
                     60     ;*********************************************************************
                     61     ; EXTERNAL 0 INTERRUPT SERVICE ROUTINE                              *
                     62     ;*********************************************************************
802D B297            63     EX0ISR: CPL    P1.7             ;complement port bit (every 0.5 ms)
802F 32              64             RETI                    ; and return
                     65
                     66     ;*********************************************************************
                     67     ; UNUSED INTERRUPTS (ERROR; RETURN TO MONITOR PROGRAM)              *
                     68     ;*********************************************************************
                     69     T0ISR:
                     70     EX1ISR:
                     71     T1ISR:
                     72     SPISR:
8030 0200BC          73     T2ISR:  LJMP    MON51
                     74             END
```

6. (a) Change line 16 to "COUNT EQU 6"
   (b) 25H to 2AH (see line 89)
   (c) 6 + (6 x 53) = 324 microseconds (see line 42)
   (d) (0.000324 / 1) x 100 = 0.0324%

7. `SETB STEP`

   Discussion: The instruction above attempts to use STEP as the address of a bit location (which it is not). This instruction will generate a "BIT SEGMENT ADDRESS EXPECTED" error message if STEP is defined using the DATA assembler directive. If STEP is define using EQU, no error message is generated.

8. The formula to calculate the key value, K from row, R and column, C is
   $K = 4*C + R$.

   The column information, C can be obtained from R6. A quick glance reveals that $C = 3 – R6$. Meanwhile, the row information, R can be obtained from P1. We check each of the upper 4 bits for a 0. When a 0 is detected, it reveals which row the key-press

is in. The subroutine is given below. We will use R0 to store the row information, R and R1 to store the column information, C.

```
HIT:  MOV A, #3
      CLR C
      SUBB A, R6        ;A = A – R6 = 3 – R6
      MOV R1, A         ;R1 = A = 3 – R6
      JNB P1.4, ROW0    ;if P1.4 is 0, it is row 0
      JNB P1.5, ROW1    ;if P1.5 is 0, it is row 1
      JNB P1.6, ROW2    ;if P1.6 is 0, it is row 2
      JNB P1.7, ROW7    ;if P1.7 is 0, it is row 3
      SJMP STOP         ;no row detected, end
ROW0: MOV R0, #0
      SJMP KEY          ;output the key
ROW1: MOV R0, #1
      SJMP KEY
ROW2: MOV R0, #2
      SJMP KEY
ROW3: MOV R0, #3
KEY:  MOV A, R1         ;A = R1 = C
      MOV B, #4
      MUL AB            ;A = A x 4
      ADD A, R0         ;A = A + R
STOP: RET               ;return from subroutine, the key value is in A
```

9.

```
1     $DEBUG
2     $NOSYMBOLS
3     $NOPAGING
4     ;FILE: LCD.SRC
5     ;***********************************************
6     ;                      PROBLEM 11-10           *
7     ;                                              *
8     ; This program continually displays on the      *
9     ; LCD the message "Welcome to the 8051          *
10    ; Microcontroller Experience. Hope you enjoy    *
11    ; your reading adventure."                      *
12    ;***********************************************
13
00B0  14            RS      EQU     0B0H
00B1  15            RW      EQU     0B1H
00B2  16            E       EQU     0B2H
0090  17            DBUS    EQU     P1
REG   18            COUNT   EQU     R1            ;LCD cursor pointer
000D  19            CR EQU 0DH                    ;ASCII carriage return code
      20
8000  21            ORG 8000H
8000 1115  22     MAIN:   ACALL INIT              ;initialize LCD
8002 7916  23             MOV COUNT, #16H        ;initialize LCD cursor pointer
8004 908054  24   STRT:   MOV DPTR, #MSG         ;initialize pointer to message
8007 93  25       NEXT:   MOVC A, @A+DPTR        ;get character
8008 1136  26             ACALL DISP             ;display on LCD
800A A3  27               INC DPTR               ;point to next character
800B D904  28             DJNZ COUNT, TEST       ;is it end of LCD line?
800D 7916  29             MOV COUNT, #16H        ;yes: reinitialize count
800F 1125  30             ACALL NEW              ; and refresh LCD
8011 70F4  31     TEST:   JNZ NEXT               ;if not last character, next
8013 80EF  32             SJMP STRT              ;last character, go back to start
      33
      34    ;***********************************************
      35    ; Initialize the LCD                          *
      36    ;***********************************************
      37
8015 7438  38     INIT:   MOV A, #38H            ;2 lines, 5 x 7 matrix
8017 113D  39             ACALL WAIT             ;wait for LCD to be free
8019 C2B0  40             CLR RS                 ;output a command
801B 114B  41             ACALL OUT              ;send it out
      42
```

```
801D 740E            43            MOV A, #0EH           ;LCD on, cursor on, blinking off
801F 113D            44            ACALL WAIT            ;wait for LCD to be free
8021 C2B0            45            CLR RS                ;output a command
8023 114B            46            ACALL OUT             ;send it out
                     47
8025 7401            48     NEW:   MOV A, #01H    ;clear LCD
8027 113D            49            ACALL WAIT            ;wait for LCD to be free
8029 C2B0            50            CLR RS                ;output a command
802B 114B            51            ACALL OUT             ;send it out
                     52
802D 7480            53            MOV A, #80H           ;cursor: line 1, position 1
802F 113D            54            ACALL WAIT            ;wait for LCD to be free
8031 C2B0            55            CLR RS                ;output a command
8033 114B            56            ACALL OUT             ;send it out
                     57
8035 22              58            RET
                     59
                     60     ;*********************************************
                     61     ; Display data on LCD                        *
                     62     ;*********************************************
                     63
8036 113D            64     DISP:  ACALL WAIT            ;wait for LCD to be free
8038 D2B0            65            SETB RS               ;output a data
803A 114B            66            ACALL OUT             ;send it out
803C 22              67            RET
                     68
                     69     ;*********************************************
                     70     ; Wait for LCD to be free                    *
                     71     ;*********************************************
803D C2B0            72     WAIT:  CLR RS                ;command
803F D2B1            73            SETB RW               ;read
8041 D297            74            SETB DBUS.7           ;DB7 = in
8043 D2B2            75            SETB E                ;1-to-0 tansition to
8045 C2B2            76            CLR E                 ; enable LCD
8047 2097F3          77            JB DBUS.7, WAIT
804A 22              78            RET
                     79
                     80     ;*********************************************
                     81     ; Output to LCD                              *
                     82     ;*********************************************
804B F590            83     OUT:   MOV DBUS, A
804D C2B1            84            CLR RW
```

```
804F D2B2             85            SETB E
8051 C2B2             86            CLR E
8053 22               87            RET
                      88
                      89     ;*********************************************
                      90     ; Message                                     *
                      91     ;*********************************************
8054 57656C63         92     MSG: DB 'Welcome to the 8051 Microcontroller Experience.',CR
8058 6F6D6520
805C 746F2074
8060 68652038
8064 30353120
8068 4D696372
806C 6F636F6E
8070 74726F6C
8074 6C657220
8078 45787065
807C 7269656E
8080 63652E
8083 0D
8084 486F7065         93            DB 'Hope you enjoy your reading adventure.',CR,0
8088 20796F75
808C 20656E6A
8090 6F792079
8094 6F757220
8098 72656164
809C 696E6720
80A0 61647665
80A4 6E747572
80A8 652E
80AA 0D
80AB 00
                      94            END
```

10. Mode 1 – Strobed I/O: This is a handshaking mode that uses ports A and B as I/O while port C is used for handshaking signals to the I/O devices connected to ports A and B.
Mode 2 – Strobed bi-directional I/O: Only port A can be in this mode. This is also a handshaking mode and is very much similar to mode 1. The only difference is that port A can be used as a bi-directional data bus.

Assuming that the 8255 has been connected as a memory-mapped I/O with the address 0103H referring to the control register, as in Section 11.8.2, then the instructions to set up the 8255 in mode 1 are:

```
MOV A, #101xx1xxB  ;Port A = mode 1, Port B = mode 1
MOV DPTR, #0103H   ;point to control register
MOVX @DPTR, A      ;send control word
```

Note that the x's above are "don't care"s and can be any bit value.

To set port A in mode 2, the following instructions are used:

```
MOV A, #110xxxxxB  ;Port A = mode 2
MOV DPTR, #0103H   ;point to control register
MOVX @DPTR, A      ;send control word
```

11. The serial interface transfers data bit by bit through a single wire, while the parallel interface transfers data simultaneously. In the case of the 8051, with data being processed 8 bits at a time, parallel interface means transferring all 8 bits of data at a time through 8 separate wires. Serial interfaces are therefore much slower than parallel interfaces, but are cost-effective especially when the distance between two communicating devices is quite far. The reason why serial interfaces are used with keyboards, mice and modems is because time is not as critical in the first two cases while distance is an important factor in the third case.

12. Pseudo Code:

```
WHILE [1] DO BEGIN
      IF [button_pressed == TRUE] DO BEGIN
            [traffic light = red]
            [pedestrian light = green]
            [wait 10 seconds]
            [traffic light = green]
            [pedestrian light = red]
      END
END
```

13. Types of stepper motors: Permanent magnet, variable reluctance, hybrid.
Applications: Computer peripherals (floppy disk drives, printers, tape readers), business machines (card readers, copy machines, automatic typewriters), machine tools (milling/grilling/grinding machines, laser cutting, sewing).

Chapter 12 - Design and Interface Examples in C

1.

```
#include <reg51.h>
#include <stdio.h>

sbit RS = P3^0;
sbit RW = P3^1;
sbit E = P3^2;
sbit busy = P1^7;
unsigned char bdata A;  /* represents ACC */
unsigned char * DPTR;
unsigned char count;
char * MSG = “Welcome to the 8051 Microcontroller Experience. Hope you
            enjoy your reading adventure.”;

void INIT(void);
void NEW(void);
void DISP(void);
void WAIT(void);
void OUT(void);

main( )
{
INIT();                        /* initialize LCD */
count = 16;                    /* initialize character count */
while(1)
      {
      DPTR = MSG;              /* point to message */
      do
            {
            A = *DPTR++;       /* get character */
            DISP();            /* display on LCD */
            if (count==0)
                  {
                  count = 16; /* if end of LCD line, reinitialize */
                  NEW();       /*  and refresh LCD */
                  }
            }
      while(A != ‘\0’);        /* repeat as long as not end of message */
      }
}

void INIT(void)
{
A = 0x38;          /* 2 lines, 5 × 7 matrix */
WAIT();            /* wait for LCD to be free */
RS = 0;            /* output a command */
OUT();             /* send it out */

A = 0x0E;          /* LCD on, cursor on */
WAIT();            /* wait for LCD to be free */
RS = 0;            /* output a command */
OUT();             /* send it out */
NEW();             /* refresh LCD display */
}
```

```
void NEW(void)
{
A = 0x01;          /* clear LCD */
WAIT();            /* wait for LCD to be free */
RS = 0;            /* output a command */
OUT();             /* send it out */

A = 0x80;          /* cursor: line 1, position 1 */
WAIT();            /* wait for LCD to be free */
RS = 0;            /* output a command */
OUT();             /* send it out */
}

void DISP(void)
{
WAIT();            /* wait for LCD to be free */
RS = 1;            /* output a data */
OUT();             /* send it out */
}

void WAIT(void)
{
do
      {
      RS = 0;      /* command */
      RW = 1;      /* read */
      busy = 1;    /* make busy bit = input */
      E = 1;       /* 1-to-0 transition to */
      E = 0;       /* enable LCD */
      }
while (busy);      /* if busy, wait */
}

void OUT(void)
{
P1 = A;            /* get ready output to LCD */
RW = 0;            /* write */
E = 1;             /* 1-to-0 transition to */
E = 0;             /* enable LCD */
```

```
}
```

2. To solve this problem, we would have to modify the `CW()` and `CCW()` functions of Example 12–14 so that they only step the motor according to the number of steps specified by an input variable `steps`. The modified functions are as below:

```
void CW(int steps)      /* function for clockwise rotation */
{
DPTR = SEQ;       /* point to start of table */
for (A = 0; A < steps; A++)
      {
      tempA = A;              /* backup index in A */
      A = DPTR[A];            /* get step pattern into 4 LSBs of A */
      A = A | (P1&0xF0);      /* retain 4 MSBs of P1 and put into A */
      P1 = A;                 /* send to stepper motor */
      delay();                /* wait for 1 sec */
      A = tempA;              /* restore index into A */
      }
}

void CCW(int steps)     /* function for counterclockwise rotation */
{
DPTR = SEQ;       /* point to start of table */
for (A = steps; A > 0; A--)
      {
      tempA = A;              /* backup index in A */
      A = DPTR[A];            /* get step pattern into 4 LSBs of A */
      A = A | (P1&0xF0);      /* retain 4 MSBs of P1 and put into A */
      P1 = A;                 /* send to stepper motor */
      delay();                /* wait for 1 sec */
      A = tempA;              /* restore index into A */
      }
}
```

Then, the `safe()` function is:

```
void safe(int * seq, int seqsize)
{
for (i = 0; i < seqsize; i=i+2)     /* repeat (seqsize÷2) times */
      {
      CW(seq[i]);                   /* alternately move clockwise  */
      CCW(seq[i+1]);                /*  and counterclockwise based */
```

```
      }                              /*  based on input sequence      */
}
```

3. First, we need to write a function to calculate the length of the message string `plain`.

```
int getlength(char * plain)
{
      int len = 0;
      while (plain[len] != 0)       /* as long as not end of string */
            len++;                  /*  increment character counter */
      return len;
}
```

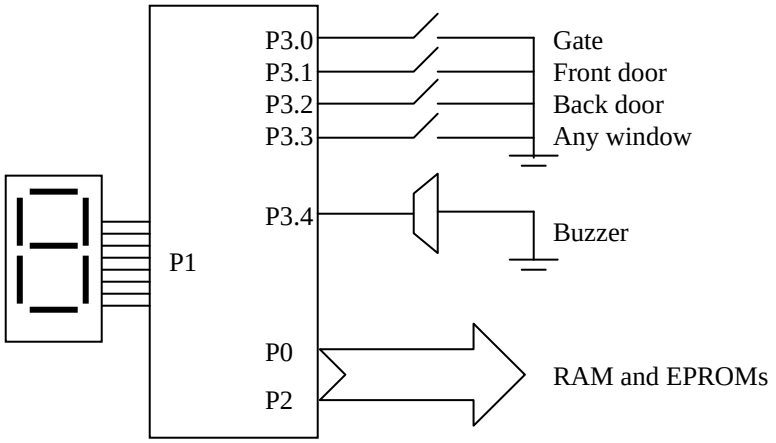
The encryption function is then:

```
void CaesarEncrypt(char * plain, char * cipher)
{
      for (i=0;i<getlength(plain);i++)
      {
            cipher[i]= plain[i]+3;
            if (cipher[i]>'z')      /* adjustment to rotate alphabets */
                  cipher[i]=cipher[i]-26;
      }
}
```

4.
```
void CaesarDecrypt(char * plain, char * cipher)
{
      for (i=0;i<getlength(cipher);i++)
      {
            plain[i]= cipher[i]-3;
            if (plain[i]<'a') /* adjustment to rotate alphabets */
                  plain[i]= plain[i]+26;
      }
}
```

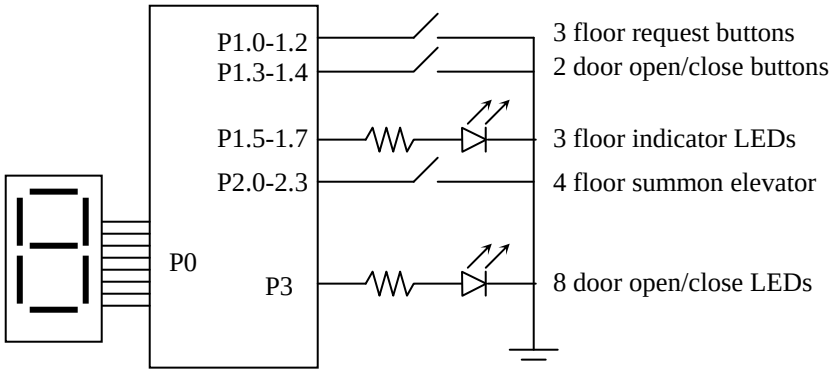
Chapter 13 - Example Student Projects

1.



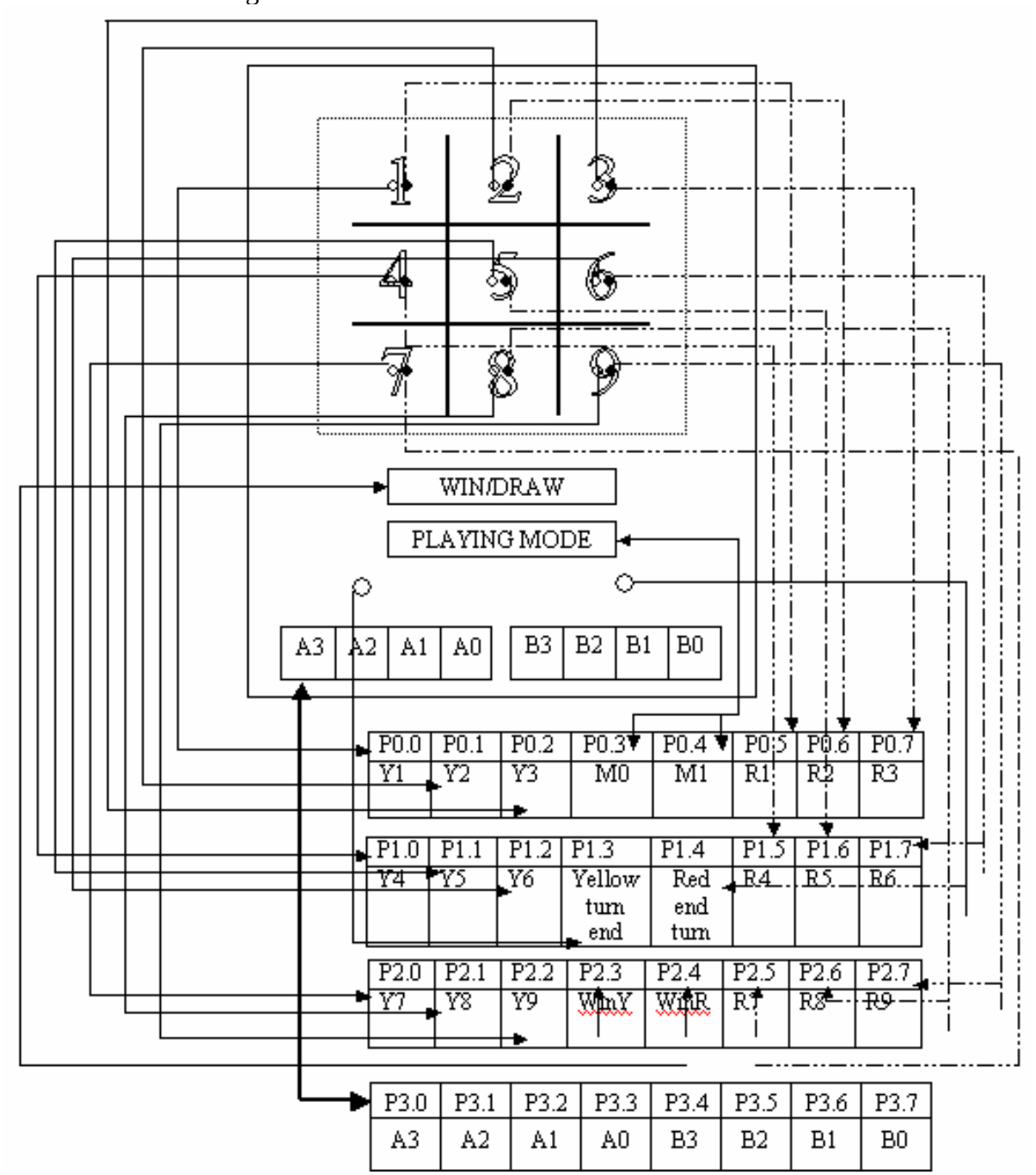
The program (either in assembly or C) is fairly straightforward and can be directly converted from the pseudo code given in Section 13.2.4.

2.



3. Most elevators service elevator requests that are going in the same direction, so an elevator would travel from the lowest to the highest floor, servicing all requests and then move in the opposite direction to continue servicing requests. This way, the elevator does not have to change direction frequently and is more economical.

4. Schematic Diagram:[1]



[1] The second author wishes to thank his students T.-B. Tay, E. Razali and J. T.-S. Teo for providing the schematic diagram and source code.

Software:

```
Start:  MOV P3, #00H
        MOV P2, #00H
        MOV P1, #00H
        MOV P0, #00H
        MOV R0, #00H
REDO:   JB P1.3, PCHK
        JB P1.4, P2P
        LJMP REDO
PCHK:   JNB P1.4, CATCH
        LJMP P2A
CATCH:  LJMP A2P
P2P:
YLW:    CLR P0.3
        CLR P0.4
        MOV P3, #00H
        JNB P0.3,  $
        MOV A, P3
        ANL A, #00001111B
        CJNE A, #00H, ONEHERE
        LJMP YLW
ONEHERE:CJNE A, #00000101B, TWOHERE
        LJMP YLW
TWOHERE:CJNE A, #00001101B, THRHERE
        LJMP YLW
THRHERE:JNB P3.0, THERE
        JNB P3.1, THERE
        LJMP YLW
THERE:  CALL CHECKYLW
        CALL YCHK
        INC R0
        CALL CHKDRAW
        LJMP RED
RED:    CLR P0.3
        CLR P0.4
        MOV P3, #00H
        JNB P0.4, $
        MOV A, P3
        ANL A, #11110000B
        CJNE A, #00H, ONEHER1
```

Reset all values in Port.

Let the user determine the playing mode, M1 & M0.

Player V.S. Player Mode

Yellow player takes his/her move first.
The process includes checking P0.3,
Yellow finishes his turn and the input, A3-A0, in port 3.
Afterwards check the status of the game, eg. Win or Draw.

Red player takes his/her move.
The process includes checking P0.4,
Red finishes his turn and the input, B3-B0, in port 3.
Afterwards check the status of the game, eg. Win or draw.

```
            LJMP RED
ONEHER1:CJNE A, #01010000B, TWOHER1
            LJMP RED
TWOHER1:CJNE A, #11010000B, THRHER1
            LJMP RED
THRHER1:JNB P3.4, THERE1
            JNB P3.5, THERE1
            LJMP RED
THERE1:  CALL CHECKRED
            CALL RCHK
            INC R0
            CALL CHKDRAW
            LJMP YLW
```

Player V.S. A.I. Mode

```
P2A:
PLAYER1:CLR P0.3
            CLR P0.4
            MOV P3, #00H
            JNB P0.3,  $
            MOV A, P3
            ANL A, #00001111B
            CJNE A, #00H, ONEHER2
            LJMP PLAYER1
ONEHER2:CJNE A, #00000101B, TWOHER2
            LJMP PLAYER1
TWOHER2:CJNE A, #00001101B, THRHER2
            LJMP PLAYER1
THRHER2:JNB P3.0, THERE2
            JNB P3.1, THERE2
            LJMP PLAYER1
THERE2:  CALL CHECKYLW
            CALL YCHK
            INC R0
            CALL CHKDRAW
            LJMP AI2
AI2:        MOV P3, #00H
            CALL AITHINK2
            JNB P0.4,  $
            CALL CHECKRED
            CALL RCHK
            INC R0
            CALL CHKDRAW
            LJMP PLAYER1
```

Yellow player takes his/her move first.
The process includes checking P0.3,
Red finishes his turn and the input, B3-B0, in port 3.
Afterwards check the status of the game, eg. Win or draw.

Continue from Player V.S. A.I., Yellow Player procedures.

These are the general steps taken by Red A.I.
The procedure is almost the same as Red player except that the A.I.'s thinking sub-routine will be made to correspond to player's input

```
A2P:
AI1:        MOV P3, #00H
            CALL AITHINK1
            CALL CHECKYLW
            CALL YCHK
            INC R0
            CALL CHKDRAW
            LJMP PLAYER2
PLAYER2:CLR P0.3
            CLR P0.4
            MOV P3, #00H
            JNB P0.4,  $
            MOV A, P3
            ANL A, #11110000B
            CJNE A, #00H, ONEHER3
            LJMP PLAYER2
ONEHER3:CJNE A, #01010000B, TWOHER3
            LJMP PLAYER2
TWOHER3:CJNE A, #11010000B, THRHER3
            LJMP PLAYER2
THRHER3:JNB P3.4, THERE3
            JNB P3.5, THERE3
            LJMP PLAYER2
THERE3:  CALL CHECKRED
            CALL RCHK
            INC R0
            CALL CHKDRAW
            LJMP AI1
CHECKRED:       CLR P0.4
                JNB P3.4, NEN1
                JNB P3.7, EIGHT1
                LJMP NINE1
NEN1:           JNB P3.5, OTT1
                JNB P3.6, FF1
                JNB P3.7, SIX1
                LJMP SEVEN1
OTT1:           JNB P3.6, ONE1
                JNB P3.7, TWO1
                LJMP THREE1
FF1:            JNB P3.7, FOUR1
                LJMP FIVE1
```

A.I. playing as Yellow goes first. The A.I. thinking sub-routine will a bit different from playing as Red.

Red player takes his/her move.
The process includes checking P0.4, Red finishes his turn and the input, B3-B0, in port 3.
Afterwards check the status of the game, eg. Win or draw.

This CheckRed sub-routine will determine the input from B3-B0 in Port 3 and light up the Red LED in the game board.
Indicate from upper 3 bits of P0, P1 and P2 according to the number in sequence.

```
ONE1:       JB P0.0, RETRED
            JB P0.5, RETRED
            SETB P0.5
            RET
TWO1:       JB P0.1, RETRED
            JB P0.6, RETRED
            SETB P0.6
            RET
THREE1:     JB P0.2, RETRED
            JB P0.7, RETRED
            SETB P0.7
            RET
FOUR1:      JB P1.0, RETRED
            JB P1.5, RETRED
            SETB P1.5
            RET
FIVE1:      JB P1.1, RETRED
            JB P1.6, RETRED
            SETB P1.6
            RET
SIX1:       JB P1.2, RETRED
            JB P1.7, RETRED
            SETB P1.7
            RET
SEVEN1:     JB P2.0, RETRED
            JB P2.5, RETRED
            SETB P2.5
            RET
EIGHT1:     JB P2.1, RETRED
            JB P2.6, RETRED
            SETB P2.6
            RET
NINE1:      JB P2.2, RETRED
            JB P2.7, RETRED
            SETB P2.7
            RET
RETRED: LJMP RED
CHECKYLW:   CLR P0.3
            JNB P3.0, NEN
            JNB P3.3, EIGHT
            LJMP NINE
NEN:        JNB P3.1, OTT
```

```
        JNB P3.2, FF
        JNB P3.3, SIX
        LJMP SEVEN
OTT:    JNB P3.2, ONE
        JNB P3.3, TWO
        LJMP THREE
FF:     JNB P3.3, FOUR
        LJMP FIVE
ONE:    JB P0.5, RETYLW
        JB P0.0, RETYLW
        SETB P0.0
        RET
TWO:    JB P0.6, RETYLW
        JB P0.1, RETYLW
        SETB P0.1
        RET
THREE:  JB P0.7, RETYLW
        JB P0.2, RETYLW
        SETB P0.2
        RET
FOUR:   JB P1.5, RETYLW
        JB P1.0, RETYLW
        SETB P1.0
        RET
FIVE:   JB P1.6, RETYLW
        JB P1.1, RETYLW
        SETB P1.1
        RET
SIX:    JB P1.7, RETYLW
        JB P1.2, RETYLW
        SETB P1.2
        RET
SEVEN:  JB P2.5, RETYLW
        JB P2.0, RETYLW
        SETB P2.0
        RET
EIGHT:  JB P2.6, RETYLW
        JB P2.1, RETYLW
        SETB P2.1
        RET
NINE:   JB P2.7, RETYLW
        JB P2.2, RETYLW
```

This CheckYellow sub-routine will determine the input from A3-A0 in Port 3 and light up the Yellow LED in the game board.
Indicate from lower 3 bits of P0, P1 and P2 according to the number in sequence.

```
                SETB P2.2
                RET
RETYLW:LJMP YLW
RCHK:
CROSSR:         MOV A, P0
                ANL A, #11100000B
                MOV R7, A
                CJNE R7, #11100000B, CROSSR1
                LJMP WINR
CROSSR1:        MOV A, P1
                ANL A, #11100000B
                MOV R7, A
                CJNE R7, #11100000B, CROSSR2
                LJMP WINR
CROSSR2:        MOV A, P2
                ANL A, #11100000B
                MOV R7, A
                CJNE R7, #11100000B, VERR
                LJMP WINR
VERR:   MOV A, #00H
        MOV C, P0.5
        MOV ACC.7, C
        MOV C, P1.5
        MOV ACC.6, C
        MOV C, P2.5
        MOV ACC.5, C
        CJNE A, #11100000B, VERR1
        LJMP WINR
VERR1:  MOV A, #00H
        MOV C, P0.6
        MOV ACC.7, C
        MOV C, P1.6
        MOV ACC.6, C
        MOV C, P2.6
        MOV ACC.5, C
        CJNE A, #11100000B, VERR2
        LJMP WINR
VERR2:  MOV A, #00H
        MOV C, P0.7
        MOV ACC.7, C
        MOV C, P1.7
        MOV ACC.6, C
```

RChk performs the task of checking all the possibilities of winning the games for the Red player, checking 3 lines horizontal lay and vertically, and 2 diagonal lines.

```
        MOV C, P2.7
        MOV ACC.5, C
        CJNE A, #11100000B, DIAGR
        LJMP WINR
DIAGR:  MOV A, #00H
        MOV C, P0.5
        MOV ACC.7, C
        MOV C, P1.6
        MOV ACC.6, C
        MOV C, P2.7
        MOV ACC.5, C
        CJNE A, #11100000B, DIAGR1
        LJMP WINR
DIAGR1: MOV A, #00H
        MOV C, P0.7
        MOV ACC.7, C
        MOV C, P1.6
        MOV ACC.6, C
        MOV C, P2.5
        MOV ACC.5, C
        CJNE A, #11100000B, RETR
        LJMP WINR
RETR:   RET
YCHK:
CROSSY:         MOV A, P0
                ANL A, #00000111B
                MOV R7, A
                CJNE R7, #00000111B, CROSSY1
                LJMP WINY
CROSSY1:        MOV A, P1
                ANL A, #00000111B
                MOV R7, A
                CJNE R7, #00000111B, CROSSY2
                LJMP WINY
CROSSY2:        MOV A, P2
                ANL A, #00000111B
                MOV R7, A
                CJNE R7, #00000111B, VERY
                LJMP WINY
VERY:   MOV A, #00H
        MOV C, P0.0
        MOV ACC.7, C
```

YChk performs the task of checking all the possibilities of winning the games for the Yellow player, checking 3 lines horizontally, and vertically, and 2 diagonal lines.

```
        MOV C, P1.0
        MOV ACC.6, C
        MOV C, P2.0
        MOV ACC.5, C
        CJNE A, #11100000B, VERY1
        LJMP WINY
VERY1:  MOV A, #00H
        MOV C, P0.1
        MOV ACC.7, C
        MOV C, P1.1
        MOV ACC.6, C
        MOV C, P2.1
        MOV ACC.5, C
        CJNE A, #11100000B, VERY2
        LJMP WINY
VERY2:  MOV A, #00H
        MOV C, P0.2
        MOV ACC.7, C
        MOV C, P1.2
        MOV ACC.6, C
        MOV C, P2.2
        MOV ACC.5, C
        CJNE A, #11100000B, DIAGY
        LJMP WINY
DIAGY:  MOV A, #00H
        MOV C, P0.0
        MOV ACC.7, C
        MOV C, P1.1
        MOV ACC.6, C
        MOV C, P2.2
        MOV ACC.5, C
        CJNE A, #11100000B, DIAGY1
        LJMP WINY
DIAGY1: MOV A, #00H
        MOV C, P0.2
        MOV ACC.7, C
        MOV C, P1.1
        MOV ACC.6, C
        MOV C, P2.0
        MOV ACC.5, C
        CJNE A, #11100000B, RETY
        LJMP WINY
```

```
RETY:     RET
CHKDRAW:  CJNE R0, #09H, RETC
          LJMP DRAW
RETC:     RET
```

Check if draw

```
WINY:     SETB P2.3
          LJMP START
WINR:     SETB P2.4
          LJMP START
DRAW:     SETB P2.3
          SETB P2.4
          LJMP START
```

Yellow wins: P2.3 will be set.
Red wins: P2.4 will be set.
Draw Game: Both P2.3 and P2.4 will be set.

```
AITHINK1:  JB P2.2, CHKL1
           MOV P3, #00001001B
BACK:      SETB P0.3
           RET
CHKL1:     MOV A, P0
           ANL A, #00000111B
           MOV R2, A
           CJNE R2, #00000110B, NEXT1
           JB P0.5, NEXT1
           MOV P3, #00001000B
           LJMP BACK
NEXT1:     CJNE R2, #00000101B, NEXT2
           JB P0.6, NEXT2
           MOV P3, #00000100B
           LJMP BACK
NEXT2:     CJNE R2, #00000011B, NEXT3
           JB P0.7, NEXT3
           MOV P3, #00001100B
           LJMP BACK
NEXT3:     MOV A, P1
           ANL A, #00000111B
           MOV R2, A
           CJNE R2, #00000110B, NEXT4
           JB P1.5, NEXT4
           MOV P3, #00000010B
           LJMP BACK
NEXT4:     CJNE R2, #00000101B, NEXT5
           JB P1.6, NEXT5
           MOV P3, #00001010B
```

The following organizes the A.I. to perform the task of blocking the opponent from winning while getting a chance to win.

The method used is to move bits from the coordinate and count the possibility to block and to fill up all the remaining vacancies.

```
                LJMP BACK
NEXT5:          CJNE R2, #00000011B, NEXT6
                JB P1.7, NEXT6
                MOV P3, #00000110B
                LJMP BACK
NEXT6:          MOV A, P2
                ANL A, #00000111B
                MOV R2, A
                CJNE R2, #00000110B, NEXT7
                JB P2.5, NEXT7
                MOV P3, #00001110B
                LJMP BACK
NEXT7:          CJNE R2, #00000101B, NEXT8
                JB P2.6, NEXT8
                MOV P3, #00000001B
                LJMP BACK
NEXT8:          CJNE R2, #00000011B, NEXT9
                JB P2.7, NEXT9
                MOV P3, #00001001B
                LJMP BACK
NEXT9:          MOV A, #00H
                MOV C, P0.0
                MOV ACC.7, C
                MOV C, P1.0
                MOV ACC.6, C
                MOV C, P2.0
                MOV ACC.5, C
                CJNE A, #01100000B, NEXT10
                JB P0.5, NEXT10
                MOV P3, #00001000B
                LJMP BACK
NEXT10:         CJNE A, #10100000B, NEXT11
                JB P1.5, NEXT11
                MOV P3, #00000010B
                LJMP BACK
NEXT11:         CJNE A, #11000000B, NEXT12
                JB P2.5, NEXT12
                MOV P3, #00001110B
                LJMP BACK
NEXT12:         MOV A, #00H
                MOV C, P0.1
                MOV ACC.7, C
```

```
                MOV C, P1.1
                MOV ACC.6, C
                MOV C, P2.1
                MOV ACC.5, C
                CJNE A, #01100000B, NEXT13
                JB P0.6, NEXT13
                MOV P3, #00000100B
                LJMP BACK
NEXT13:         CJNE A, #10100000B, NEXT14
                JB P1.6, NEXT14
                MOV P3, #00001010B
                LJMP BACK
NEXT14:         CJNE A, #11000000B, NEXT15
                JB P2.6, NEXT15
                MOV P3, #00000001B
                LJMP BACK
NEXT15:         MOV A, #00H
                MOV C, P0.2
                MOV ACC.7, C
                MOV C, P1.2
                MOV ACC.6, C
                MOV C, P2.2
                MOV ACC.5, C
                CJNE A, #01100000B, NEXT16
                JB P0.7, NEXT16
                MOV P3, #00001100B
                LJMP BACK
NEXT16:         CJNE A, #10100000B, NEXT17
                JB P1.7, NEXT17
                MOV P3, #00000110B
                LJMP BACK
NEXT17:         CJNE A, #11000000B, NEXT18
                JB P2.7, NEXT18
                MOV P3, #00001001B
                LJMP BACK
NEXT18:         MOV A, #00H
                MOV C, P0.0
                MOV ACC.7, C
                MOV C, P1.1
                MOV ACC.6, C
                MOV C, P2.2
                MOV ACC.5, C
```

```
                CJNE A, #01100000B, NEXT19
                JB P0.5, NEXT19
                MOV P3, #00001000B
                LJMP BACK
NEXT19:         CJNE A, #10100000B, NEXT20
                JB P1.6, NEXT20
                MOV P3, #00001010B
                LJMP BACK
NEXT20:         CJNE A, #11000000B, NEXT21
                JB P2.7, NEXT21
                MOV P3, #00001001B
                LJMP BACK
NEXT21:         MOV A, #00H
                MOV C, P0.2
                MOV ACC.7, C
                MOV C, P1.1
                MOV ACC.6, C
                MOV C, P2.0
                MOV ACC.5, C
                CJNE A, #01100000B, NEXT22
                JB P0.7, NEXT22
                MOV P3, #00001100B
                LJMP BACK
NEXT22:         CJNE A, #10100000B, NEXT23
                JB P1.6, NEXT23
                MOV P3, #00001010B
                LJMP BACK
NEXT23:         CJNE A, #11000000B, NEXT24
                JB P2.5, NEXT24
                MOV P3, #00001110B
                LJMP BACK
NEXT24:         MOV A, P0
                ANL A, #11100000B
                MOV R1, A
                CJNE R1, #11000000B, NEXT25
                JB P0.0, NEXT25
                MOV P3, #00001000B
                LJMP BACK
NEXT25:         CJNE R1, #10100000B, NEXT26
                JB P0.1, NEXT26
                MOV P3, #00000100B
                LJMP BACK
```

```
NEXT26:   CJNE R1, #01100000B, NEXT27
          JB P0.2, NEXT27
          MOV P3, #00001100B
          LJMP BACK
NEXT27:   MOV A, P1
          ANL A, #11100000B
          MOV R1, A
          CJNE R1, #11000000B, NEXT28
          JB P1.0, NEXT28
          MOV P3, #00000010B
          LJMP BACK
NEXT28:   CJNE R1, #10100000B, NEXT29
          JB P1.1, NEXT29
          MOV P3, #00001010B
          LJMP BACK
NEXT29:   CJNE R1, #01100000B, NEXT30
          JB P1.2, NEXT30
          MOV P3, #00000110B
          LJMP BACK
NEXT30:   MOV A, P2
          ANL A, #11100000B
          MOV R1, A
          CJNE R1, #11000000B, NEXT31
          JB P2.0, NEXT31
          MOV P3, #00001110B
          LJMP BACK
NEXT31:   CJNE R1, #10100000B, NEXT32
          JB P2.1, NEXT32
          MOV P3, #00000001B
          LJMP BACK
NEXT32:   CJNE R1, #01100000B, NEXT33
          JB P2.2, NEXT33
          MOV P3, #00001001B
          LJMP BACK
NEXT33:   MOV A, #00H
          MOV C, P0.5
          MOV ACC.7, C
          MOV C, P1.5
          MOV ACC.6, C
          MOV C, P2.5
          MOV ACC.5, C
          CJNE A, #01100000B, NEXT34
```

```
        JB P0.0, NEXT34
        MOV P3, #00001000B
        LJMP BACK
NEXT34: CJNE A, #10100000B, NEXT35
        JB P1.0, NEXT35
        MOV P3, #00000010B
        LJMP BACK
NEXT35: CJNE A, #11000000B, NEXT36
        JB P2.0, NEXT36
        MOV P3, #00001110B
        LJMP BACK
NEXT36: MOV A, #00H
        MOV C, P0.6
        MOV ACC.7, C
        MOV C, P1.6
        MOV ACC.6, C
        MOV C, P2.6
        MOV ACC.5, C
        CJNE A, #01100000B, NEXT37
        JB P0.1, NEXT37
        MOV P3, #00000100B
        LJMP BACK
NEXT37: CJNE A, #10100000B, NEXT38
        JB P1.1, NEXT38
        MOV P3, #00001010B
        LJMP BACK
NEXT38: CJNE A, #11000000B, NEXT39
        JB P2.1, NEXT39
        MOV P3, #00000001B
        LJMP BACK
NEXT39: MOV A, #00H
        MOV C, P0.7
        MOV ACC.7, C
        MOV C, P1.7
        MOV ACC.6, C
        MOV C, P2.7
        MOV ACC.5, C
        CJNE A, #01100000B, NEXT40
        JB P0.2, NEXT40
        MOV P3, #00001100B
        LJMP BACK
NEXT40: CJNE A, #10100000B, NEXT41
```

```
                JB P1.2, NEXT41
                MOV P3, #00000110B
                LJMP BACK
NEXT41:         CJNE A, #11000000B, NEXT42
                JB P2.2, NEXT42
                MOV P3, #00001001B
                LJMP BACK
NEXT42:         MOV A, #00H
                MOV C, P0.5
                MOV ACC.7, C
                MOV C, P1.6
                MOV ACC.6, C
                MOV C, P2.7
                MOV ACC.5, C
                CJNE A, #01100000B, NEXT43
                JB P0.0, NEXT43
                MOV P3, #00001000B
                LJMP BACK
NEXT43:         CJNE A, #10100000B, NEXT44
                JB P1.1, NEXT44
                MOV P3, #00001010B
                LJMP BACK
NEXT44:         CJNE A, #11000000B, NEXT45
                JB P2.2, NEXT45
                MOV P3, #00001001B
                LJMP BACK
NEXT45:         MOV A, #00H
                MOV C, P0.7
                MOV ACC.7, C
                MOV C, P1.6
                MOV ACC.6, C
                MOV C, P2.5
                MOV ACC.5, C
                CJNE A, #01100000B, NEXT46
                JB P0.2, NEXT46
                MOV P3, #00001100B
                LJMP BACK
NEXT46:         CJNE A, #10100000B, NEXT47
                JB P1.1, NEXT47
                MOV P3, #00001010B
                LJMP BACK
NEXT47:         CJNE A, #11000000B, NEXT90
```

```
                JB P2.0, NEXT90
                MOV P3, #00001110B
                LJMP BACK
NEXT90:         JB P0.5, NEXT91
                JB P0.0, NEXT91
                MOV P3, #00001000B
                LJMP BACK
NEXT91:         JB P2.5, NEXT92
                JB P2.0, NEXT92
                MOV P3, #00001110B
                LJMP BACK
NEXT92:         JB P0.7, SET01
                JB P0.2, SET01
                MOV P3, #00001100B
                LJMP BACK

SET01:          JB P1.1, SET02
                JB P1.6, SET02
                MOV P3, #00001010B
                LJMP BACK
SET02:          JB P1.0, SET03
                JB P1.5, SET03
                MOV P3, #00000010B
                LJMP BACK
SET03:          JB P2.1, SET04
                JB P2.6, SET04
                MOV P3, #00000001B
                LJMP BACK
SET04:          JB P1.2, SET05
                JB P1.7, SET05
                MOV P3, #00000110B
                LJMP BACK
SET05:          JB P0.1, SET06
                JB P0.6, SET06
                MOV P3, #00000100B
                LJMP BACK
SET06:          JB P0.0, SET07
                JB P0.5, SET07
                MOV P3, #00001000B
                LJMP BACK
SET07:          JB P0.2, SET08
                JB P0.7, SET08
```

```
                MOV P3, #00001100B
                LJMP BACK
SET08:          JB P2.0, SET09
                JB P2.5, SET09
                MOV P3, #00001110B
                LJMP BACK
SET09:          MOV P3, #00001001B
                LJMP BACK


AITHINK2:       CJNE R0, #01H, FIX
                JB P1.1, FIND
                JB P1.6, FIND
                MOV P3, #10100000B
BACK1:          SETB P0.4
                RET
FIND:           JNB P0.2, FIND1
                JB P2.0, FIND1
                JB P2.5, FIND1
                MOV P3, #11100000B
                LJMP BACK1
FIND1:          JNB P2.0, FIND2
                JB P0.2, FIND2
                JB P0.7, FIND2
                MOV P3, #11000000B
                LJMP BACK1
FIND2:          JNB P2.2, FIND3
                JB P0.0, FIND3
                JB P0.5, FIND3
                MOV P3, #10000000B
                LJMP BACK1
FIND3:          JNB P1.1, ALT
                JB P2.2, ALT
                JB P2.7, ALT
                MOV P3, #10010000B
                LJMP BACK1
ALT:            MOV P3, #10100000B
                LJMP BACK1
FIX:            MOV A, P0
                ANL A, #00000111B
                MOV R1, A
                MOV A, P0
```

```
                ANL A, #11100000B
                MOV R2, A
                CJNE R2, #11000000B, NEXT01
                JB P0.0, NEXT01
                JB P0.5, NEXT01
                MOV P3, #10000000B
                LJMP BACK1
NEXT01:         CJNE R1, #00000110B, NEXT02
                JB P0.5, NEXT02
                JB P0.0, NEXT02
                MOV P3, #10000000B
                LJMP BACK1
NEXT02:         CJNE R2, #10100000B, NEXT03
                JB P0.1, NEXT03
                JB P0.6, NEXT03
                MOV P3, #01000000B
                LJMP BACK1
NEXT03:         CJNE R1, #00000101B, NEXT04
                JB P0.6, NEXT04
                JB P0.1, NEXT04
                MOV P3, #01000000B
                LJMP BACK1
NEXT04:         CJNE R2, #01100000B, NEXT05
                JB P0.2, NEXT05
                JB P0.7, NEXT05
                MOV P3, #11000000B
                LJMP BACK1
NEXT05:         CJNE R1, #00000011B, CHKL02
                JB P0.7, CHKL02
                JB P0.2, CHKL02
                MOV P3, #11000000B
                LJMP BACK1
CHKL02:         MOV A, P1
                ANL A, #00000111B
                MOV R1, A
                MOV A, P1
                ANL A, #11100000B
                MOV R2, A
                CJNE R2, #11000000B, NEXT011
                JB P1.0, NEXT011
                JB P1.5, NEXT011
                MOV P3, #00100000B
```

```
                LJMP BACK1
NEXT011:CJNE R1, #00000110B, NEXT012
                JB P1.5, NEXT012
                JB P1.0, NEXT012
                MOV P3, #00100000B
                LJMP BACK1
NEXT012:CJNE R2, #10100000B, NEXT013
                JB P1.1, NEXT013
                JB P1.6, NEXT013
                MOV P3, #10100000B
                LJMP BACK1
NEXT013:CJNE R1, #00000101B, NEXT014
                JB P1.6, NEXT014
                JB P1.1, NEXT014
                MOV P3, #10100000B
                LJMP BACK1
NEXT014:CJNE R2, #01100000B, NEXT015
                JB P1.2, NEXT015
                JB P1.7, NEXT015
                MOV P3, #01100000B
                LJMP BACK1
NEXT015:CJNE R1, #00000011B, CHKL03
                JB P1.7, CHKL03
                JB P1.2, CHKL03
                MOV P3, #01100000B
                LJMP BACK1
CHKL03:         MOV A, P2
                ANL A, #00000111B
                MOV R1, A
                MOV A, P2
                ANL A, #11100000B
                MOV R2, A
                CJNE R2, #11000000B, NEXT021
                JB P2.0, NEXT021
                JB P2.5, NEXT021
                MOV P3, #11100000B
                LJMP BACK1
NEXT021:CJNE R1, #00000110B, NEXT022
                JB P2.5, NEXT022
                JB P2.0, NEXT022
                MOV P3, #11100000B
                LJMP BACK1
```

```
NEXT022:CJNE R2, #10100000B, NEXT023
        JB P2.1, NEXT023
        JB P2.6, NEXT023
        MOV P3, #00010000B
        LJMP BACK1
NEXT023:CJNE R1, #00000101B, NEXT024
        JB P2.6, NEXT024
        JB P2.1, NEXT024
        MOV P3, #00010000B
        LJMP BACK1
NEXT024:CJNE R2, #01100000B, NEXT025
        JB P2.2, NEXT025
        JB P2.7, NEXT025
        MOV P3, #10010000B
        LJMP BACK1
NEXT025:CJNE R1, #00000011B, CHKL04
        JB P2.7, CHKL04
        JB P2.2, CHKL04
        MOV P3, #10010000B
        LJMP BACK1
CHKL04: MOV A, #00H
        MOV C, P0.5
        MOV ACC.7, C
        MOV C, P1.5
        MOV ACC.6, C
        MOV C, P2.5
        MOV ACC.5, C
        CJNE A, #01100000B, NEXT034
        JB P0.0, NEXT034
        JB P0.5, NEXT034
        MOV P3, #10000000B
        LJMP BACK1
NEXT034:CJNE A, #10100000B, NEXT035
        JB P1.0, NEXT035
        JB P1.5, NEXT035
        MOV P3, #00100000B
        LJMP BACK1
NEXT035:CJNE A, #11000000B, NEXT033
        JB P2.0, NEXT033
        JB P2.5, NEXT033
        MOV P3, #11100000B
        LJMP BACK1
```

```
NEXT033:MOV A, #00H
        MOV C, P0.0
        MOV ACC.7, C
        MOV C, P1.0
        MOV ACC.6, C
        MOV C, P2.0
        MOV ACC.5, C
        CJNE A, #01100000B, NEXT031
        JB P0.5, NEXT031
        JB P0.0, NEXT031
        MOV P3, #10000000B
        LJMP BACK1
NEXT031:CJNE A, #10100000B, NEXT032
        JB P1.5, NEXT032
        JB P1.0, NEXT032
        MOV P3, #00100000B
        LJMP BACK1
NEXT032:CJNE A, #11000000B, CHKL05
        JB P2.5, CHKL05
        JB P2.0, CHKL05
        MOV P3, #11100000B
        LJMP BACK1
CHKL05: MOV A, #00H
        MOV C, P0.6
        MOV ACC.7, C
        MOV C, P1.6
        MOV ACC.6, C
        MOV C, P2.6
        MOV ACC.5, C
        CJNE A, #01100000B, NEXT044
        JB P0.1, NEXT044
        JB P0.6, NEXT044
        MOV P3, #01000000B
        LJMP BACK1
NEXT044:CJNE A, #10100000B, NEXT045
        JB P1.1, NEXT045
        JB P1.6, NEXT045
        MOV P3, #10100000B
        LJMP BACK1
NEXT045:CJNE A, #11000000B, NEXT043
        JB P2.1, NEXT043
        JB P2.6, NEXT043
```

```
                MOV P3, #00010000B
                LJMP BACK1
NEXT043:MOV A, #00H
                MOV C, P0.1
                MOV ACC.7, C
                MOV C, P1.1
                MOV ACC.6, C
                MOV C, P2.1
                MOV ACC.5, C
                CJNE A, #01100000B, NEXT041
                JB P0.6, NEXT041
                JB P0.1, NEXT041
                MOV P3, #01000000B
                LJMP BACK1
NEXT041:CJNE A, #10100000B, NEXT042
                JB P1.6, NEXT042
                JB P1.1, NEXT042
                MOV P3, #10100000B
                LJMP BACK1
NEXT042:CJNE A, #11000000B, CHKL06
                JB P2.6, CHKL06
                JB P2.1, CHKL06
                MOV P3, #00010000B
                LJMP BACK1
CHKL06:         MOV A, #00H
                MOV C, P0.7
                MOV ACC.7, C
                MOV C, P1.7
                MOV ACC.6, C
                MOV C, P2.7
                MOV ACC.5, C
                CJNE A, #01100000B, NEXT054
                JB P0.2, NEXT054
                JB P0.7, NEXT054
                MOV P3, #11000000B
                LJMP BACK1
NEXT054:CJNE A, #10100000B, NEXT055
                JB P1.2, NEXT055
                JB P1.7, NEXT055
                MOV P3, #01100000B
                LJMP BACK1
NEXT055:CJNE A, #11000000B, NEXT053
```

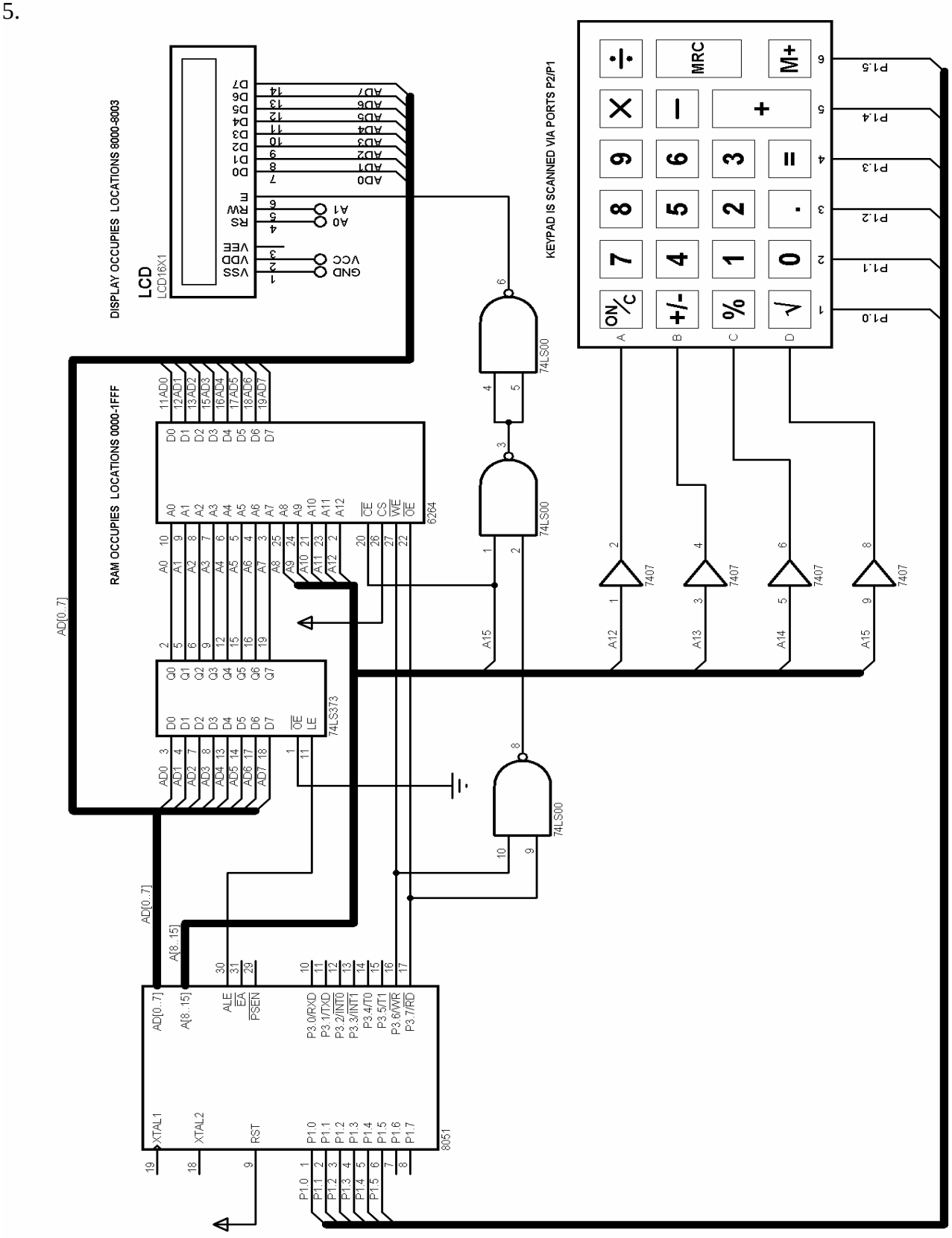
```
                JB P2.2, NEXT053
                JB P2.7, NEXT053
                MOV P3, #10010000B
                LJMP BACK1
NEXT053:MOV A, #00H
                MOV C, P0.2
                MOV ACC.7, C
                MOV C, P1.2
                MOV ACC.6, C
                MOV C, P2.2
                MOV ACC.5, C
                CJNE A, #01100000B, NEXT051
                JB P0.7, NEXT051
                JB P0.2, NEXT051
                MOV P3, #11000000B
                LJMP BACK1
NEXT051:CJNE A, #10100000B, NEXT052
                JB P1.7, NEXT052
                JB P1.2, NEXT052
                MOV P3, #01100000B
                LJMP BACK1
NEXT052:CJNE A, #11000000B, CHKL07
                JB P2.7, CHKL07
                JB P2.2, CHKL07
                MOV P3, #10010000B
                LJMP BACK1
CHKL07:         MOV A, #00H
                MOV C, P0.5
                MOV ACC.7, C
                MOV C, P1.6
                MOV ACC.6, C
                MOV C, P2.7
                MOV ACC.5, C
                CJNE A, #01100000B, NEXT064
                JB P0.0, NEXT064
                JB P0.5, NEXT064
                MOV P3, #10000000B
                LJMP BACK1
NEXT064:CJNE A, #10100000B, NEXT065
                JB P1.1, NEXT065
                JB P1.6, NEXT065
                MOV P3, #10100000B
```

```
                LJMP BACK1
NEXT065:CJNE A, #11000000B, NEXT063
                JB P2.2, NEXT063
                JB P2.7, NEXT063
                MOV P3, #10010000B
                LJMP BACK1
NEXT063:MOV A, #00H
                MOV C, P0.0
                MOV ACC.7, C
                MOV C, P1.1
                MOV ACC.6, C
                MOV C, P2.2
                MOV ACC.5, C
                CJNE A, #01100000B, NEXT061
                JB P0.5, NEXT061
                JB P0.0, NEXT061
                MOV P3, #10000000B
                LJMP BACK1
NEXT061:CJNE A, #10100000B, NEXT062
                JB P1.6, NEXT062
                JB P1.1, NEXT062
                MOV P3, #10100000B
                LJMP BACK1
NEXT062:CJNE A, #11000000B, CHKL08
                JB P2.7, CHKL08
                JB P2.2, CHKL08
                MOV P3, #10010000B
                LJMP BACK1
CHKL08:         MOV A, #00H
                MOV C, P0.7
                MOV ACC.7, C
                MOV C, P1.6
                MOV ACC.6, C
                MOV C, P2.5
                MOV ACC.5, C
                CJNE A, #01100000B, NEXT074
                JB P0.2, NEXT074
                JB P0.7, NEXT074
                MOV P3, #11000000B
                LJMP BACK1
NEXT074:CJNE A, #10100000B, NEXT075
                JB P1.1, NEXT075
```

```
                JB P1.6, NEXT075
                MOV P3, #10100000B
                LJMP BACK1
NEXT075:CJNE A, #11000000B, NEXT073
                JB P2.0, NEXT073
                JB P2.5, NEXT073
                MOV P3, #11100000B
                LJMP BACK1
NEXT073:MOV A, #00H
                MOV C, P0.2
                MOV ACC.7, C
                MOV C, P1.1
                MOV ACC.6, C
                MOV C, P2.0
                MOV ACC.5, C
                CJNE A, #01100000B, NEXT071
                JB P0.7, NEXT071
                JB P0.2, NEXT071
                MOV P3, #11000000B
                LJMP BACK1
NEXT071:CJNE A, #10100000B, NEXT072
                JB P1.6, NEXT072
                JB P1.1, NEXT072
                MOV P3, #10100000B
                LJMP BACK1
NEXT072:CJNE A, #11000000B, SET1
                JB P2.5, SET1
                JB P2.0, SET1
                MOV P3, #11100000B
                LJMP BACK1
SET1:           JB P1.1, SET2
                JB P1.6, SET2
                MOV P3, #10100000B
                LJMP BACK1
SET2:           JB P1.0, SET3
                JB P1.5, SET3
                MOV P3, #00100000B
                LJMP BACK1
SET3:           JB P2.1, SET4
                JB P2.6, SET4
                MOV P3, #00010000B
                LJMP BACK1
```

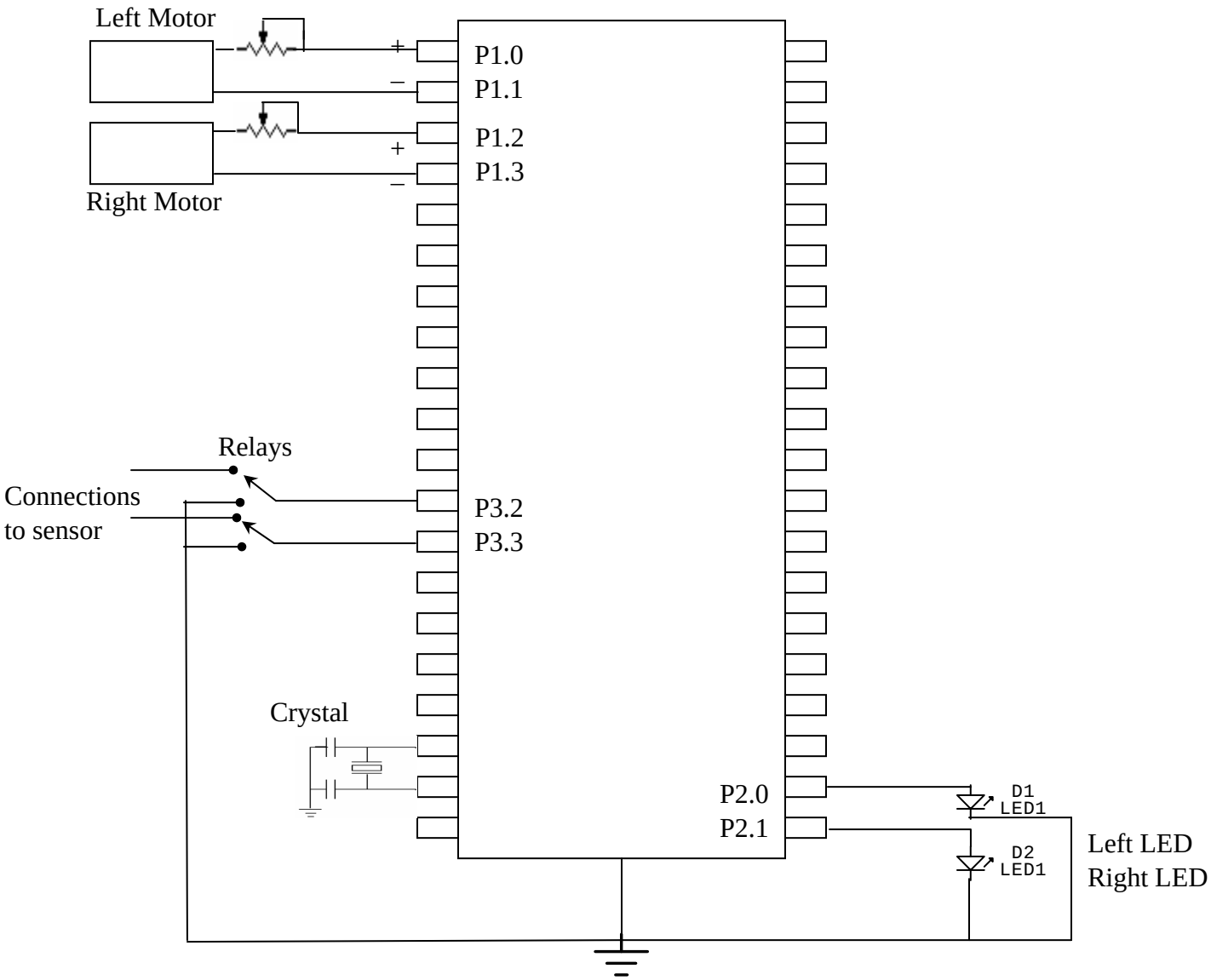
```
SET4:   JB P1.2, SET5
        JB P1.7, SET5
        MOV P3, #01100000B
        LJMP BACK1
SET5:   JB P0.1, SET6
        JB P0.6, SET6
        MOV P3, #01000000B
        LJMP BACK1
SET6:   JB P0.0, SET7
        JB P0.5, SET7
        MOV P3, #10000000B
        LJMP BACK1
SET7:   JB P0.2, SET8
        JB P0.7, SET8
        MOV P3, #11000000B
        LJMP BACK1
SET8:   JB P2.0, SET9
        JB P2.5, SET9
        MOV P3, #11100000B
        LJMP BACK1
SET9:   MOV P3, #10010000B
        LJMP BACK1
        END
```

5.


DISPLAY OCCUPIES LOCATIONS 8000-8003
LCD
LCD16X1
VSS
VDD
VEE
RS
RW
E
D0
D1
D2
D3
D4
D5
D6
D7
GND
VCC
A0
A1
AD0
AD1
AD2
AD3
AD4
AD5
AD6
AD7
KEYPAD IS SCANNED VIA PORTS P2/P1
ON/C
+/-
%
√
7
8
9
÷
4
5
6
×
1
2
3
−
0
.
=
+
MRC
M+
P1.0
P1.1
P1.2
P1.3
P1.4
P1.5
RAM OCCUPIES LOCATIONS 0000-1FFF
6264
74LS373
74LS00
7407
AD[0..7]
A[8..15]
A12
A13
A14
A15
8051
XTAL1
XTAL2
RST
ALE
EA
PSEN
P3.0/RXD
P3.1/TXD
P3.2/INT0
P3.3/INT1
P3.4/T0
P3.5/T1
P3.6/WR
P3.7/RD
P1.0
P1.1
P1.2
P1.3
P1.4
P1.5
P1.6
P1.7

6.
```
ADDITION:   MOV A,22H
            ADD A,24H          ;add low bytes
            MOV 22H,A
            MOV A,23h
            ADDC A,#0DH        ;add high bytes with carry
            JNC DONE
            MOV 23H, #0        ;overflow, put 00 in result
            SJMP OUT
DONE:       MOV 23H, A         ;store result
OUT:        RET
```

7.



8. (a) To solve this problem, we could imagine that we ourselves were walking through a maze. The choice is ultimately up to the individual, but the second author would personally prefer to test the furthest branches first because usually the furthest branches would be shorter or would have less internal branches, and so would be easier to keep track off. This perception is also due to his experience of reading interactive novels in which he adopted the same approach in getting to the end of the story.

   (b) The technique used to memorize previously traversed paths is very much dependent on the choice made in the previous section (a). The mouse could keep a record of the number of branches and their corresponding directions (left, right, front), and then when it comes across further internal branches, should connect these to the outer branches in very much the same way as the directory structure within our computer. Then, moving inwards out (based on the choice made in section (a) previously), all branches that come to a dead end are eliminated from the list of branches traversed.

9. Since smart cards are essentially tiny computers with microcontrollers and memory, we can embed photos in smart cards in the same way that we store photos in the computer. The difference lies in the limited memory size in smart cards, so photos for smart cards should be saved in compressed formats such as JPEG.

10. Pseudo code:

```
BEGIN
      REPEAT
            BEGIN
                  [try current shift to decrypt ciphertext]
                  [compare decrypted value with entire dictionary]
                  IF [decrypted_value == dictionary_entry]
                        [key = current_shift]
            END
      UNTIL [all 26 possible shifts tested]
END
```

11.
```
#include <reg51.h>
#include <stdio.h>

char * dictionary[3]={"lets", "date", "tonight"};
char * cipher1 = {"atih"};
char * cipher2 = {"spit"};
char * cipher3 = {"idcxvwi"};
char plain1[4];
char plain2[4];
char plain3[7];
int i, j, count, length, match = 0, key;

int getlength(char * cipher)
{
      int len = 0;
      while (cipher[len] != 0)
            len++;
      return len;
}

void decrypt(char * cipher, char * plain, int shift)
{
      for (j=0;j<getlength(cipher);j++)
      {
            plain[j]=cipher[j]-shift;
            if (plain[j]<'a')
                  plain[j]=plain[j]+26;
      }
}

bit compare(char * plain)
{
      for (j=0;j<getlength(plain1);j++)
            if (plain[j] != dictionary[0][j])
                  return 0;
            return 1;
}

void attack(void)
{
      for (i=0; i<26; i++)    //try all possible shifts
      {
```

```
            decrypt(cipher1, plain1, i);
            if ( compare(plain1))
            {
                  printf("match!\n");
                  key = i;
                  break;
            }
      }
}

main()
{
    SCON  = 0x50;       /* SCON: mode 1, 8-bit UART, enable rcvr      */
    TMOD |= 0x20;       /* TMOD: timer 1, mode 2, 8-bit reload        */
    TH1   = 221;        /* TH1:  reload value for 1200 baud @ 16MHz   */
    TR1   = 1;          /* TR1:  timer 1 run                          */
    TI    = 1;          /* TI:   set TI to send first char of UART    */

      attack();
      decrypt(cipher1, plain1, key);
      printf("Plaintext: %s ",plain1);
      while(1);
}
```

12. The AES has been implemented in 8051 assembly language by several sources. Check http://www.nist.gov/aes for further information on the AES.

Chapter 14 - 8051 Derivatives

1. Check out the websites of 8051 derivative manufacturers including Atmel, Maxim, Philips, Siemens.

2. Maxim's DS5250 microcontroller includes the following security features:
   - meets physical security requirements of FIPS-140
   - program memory integrity checking
   - supports DES encryptions
   - Programmable attack countermeasures
   - Secure bootstrap loader

   Philip's 83C852 also has security features that make it ideal as a smart card.

3. Other enhancements to the 8051 include built-in pulse-width modulated (PWM) outputs such as Philip's 8XC055, and built-in watchdog timers such as Atmel's 89S8252, Maxim's 80C320, Philip's 8XC524 and Siemens' 90C502.