Unit 2: Instruction set and programming of PIC 18

Addressing modes:-

The CPU can access data in various ways. The data could be in a register, or in memory, or provided as an immediate value. These various ways of accessing data are called *addressing modes*. In this chapter we discuss PIC18 addressing modes in the context of some examples.

The various addressing modes of a microprocessor are determined when it is designed, and therefore cannot be changed by the programmer. The PIC18 provides a total of three distinct addressing modes. They are as follows:

1. Immediate
2. Direct
3. Register indirect

1. Immediate addressing mode:-

In this addressing mode, the operand is a literal constant. In immediate addressing mode, as the name implies, the operand comes immediately after the opcode when the instruction is assembled. Notice that immediate data is called a *literal* in the PIC. This addressing mode can be used to load information into WREG and selected registers, but not to any file register. The immediate addressing mode is also used for arithmetic and logic instructions. Examine the following examples.

```
MOVLW  0x25          ;load 25H into WREG
SUBLW  D'62'         ;subtract WREG from 62
ANDLW  B'01000000'   ;AND WREG with 40H
```

2. Direct addressing mode:-

In direct addressing mode, the operand data is in a RAM memory location whose address is known, and this address is given as a part of the instruction. Contrast this with immediate addressing mode in which the operand data itself is provided with the instruction. While the letter "L" in the instruction means literal (immediate), the letter "F" in the instruction signifies the address of the file register location. See the example below, and note the letter F in the instructions.

```
MOVLW  0x56          ;WREG = 56H (immediate addressing mode)
MOVWF  0x40          ;copy WREG into fileReg RAM location 40H
MOVFF  0x40,0x50     ;copy data from loc 40H to 50H.
```

3. Register Indirect addressing mode:-

In the register indirect addressing mode, a register is used as a pointer to the data RAM location. In the PIC18, three registers are used for this purpose: FSR0, FSR1, and FSR2. FSR stands for *file select register* and must not be confused with SFR (special function register). The FSR is a 12-bit register allowing access to the entire 4096 bytes of data RAM space in the PIC18. We use LFSR (load FSR) to load the RAM address. In other words, when FSRx are used as pointers, they must be loaded first with the RAM addresses as shown below.

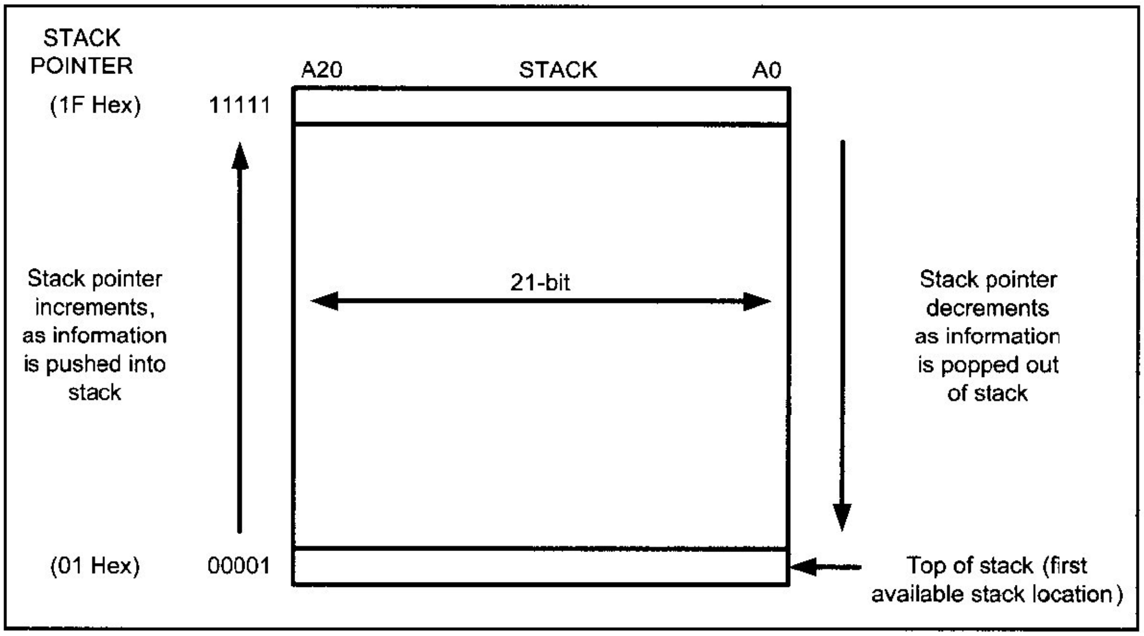
```
LFSR 0, 0x30        ;load FSR0 with 0x30
LFSR 1, 0x40        ;load FSR1 with 0x40
LFSR 2, 0x6F        ;load FSR2 with 0x6F
```

Because FSR0, FSR1, and FSR2 are 12-bit registers they cannot fit into the SFR address space unless they are split into pieces of an 8-bit size. That is exactly what PIC18 has done. The FSR registers have the low-byte and high-byte parts called FSRxL and FSRxH, as shown in the SFR table of Table 6-1. In Table 6-1 we see FSR0L and FSR0H, representing the low and high parts of the 12-bit FSR0 register. Note that the FSRxH is only 4-bit and the upper 4 bits are not used. Another register associated with the register indirect addressing mode is the INDF (indirect register). Each of the FSR0, FSR1, and FSR2 registers has an INDF register associated with it, and these are called INDF0, INDF1, and INDF2. When we move data into INDFx we are moving data into a RAM location pointed to by the FSR. In the same way, when we read data from the INDF register, we are reading data from a RAM location pointed to by the FSR. This is shown below.

```
LFSR  0, 0x30       ;FSR0 = 30H RAM location pointer
MOVWF INDF0         ;copy contents of WREG into RAM
                    ;location whose address is held by
                    ;12-bit FSR0 register
```

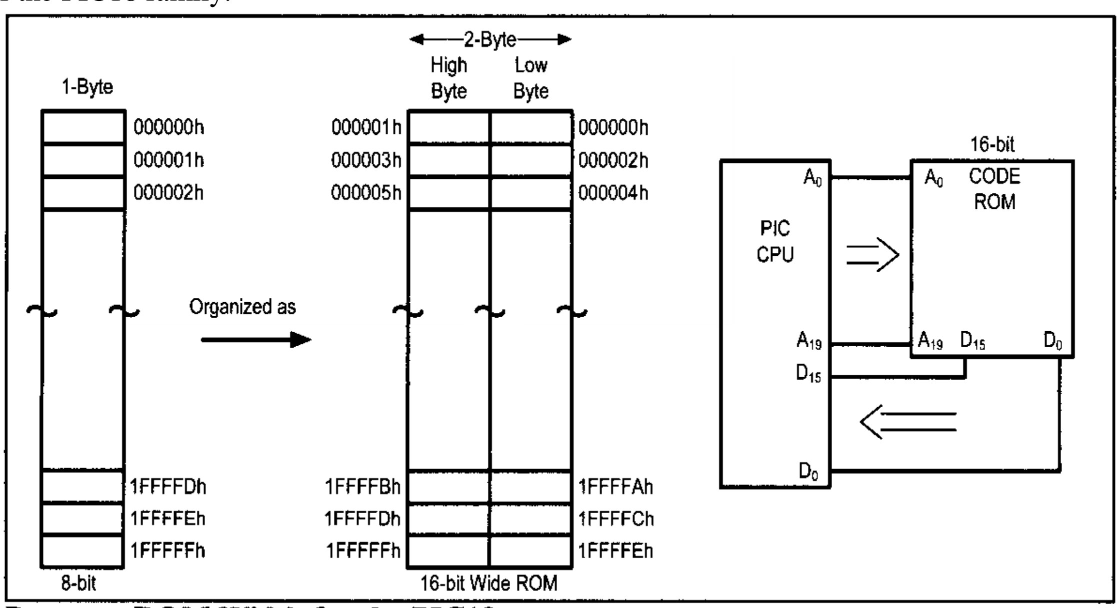
Stack and Stack pointer:-

The stack is read/write memory (RAM) used by the CPU to store some very critical information temporarily. This information usually is an address, but it could be data as well. The CPU needs this storage area because there are only a limited number of registers. The stack in the PIC18 is 21-bit because the program counter is 21-bit. This means that it is used for the CALL instruction to make sure that the PIC knows where to come back to after execution of the called subroutine. A 21-bit stack can take values of 00000 to 1FFFFFH, just like the program counter. If the stack is RAM, there must be a register inside the CPU to point to it. The register used to access the stack is called the SP (stack pointer) register. The PIC18 has a 5-bit stack pointer, which can take values of 00 to 1FH. That gives us a total of 32 locations where each location is 21 bits wide. This is shown in Figure 3-7. When the PIC18 is powered up, the SP register contains value 0. This means that stack location 1 is the first location used for the stack because the SP points to the last-used location. That means that location 0 of the stack is not available and we have only 31 stack locations in the PIC18.



ROM width in PIC 18:-

ROM is the memory where code of the program is stored. If we have 16 address lines, it wil give us 2^{16} locations, which is 64K bytes of memory space with an address map of 0000-FFFF. CPU's with 8-bit data will fetch one byte at a time. To bring in more code information into CPU we can increase the width of the data bus to 16 bits. For the PIC18, the internal data bus between the code ROM and the CPU is 16 bits. Therefore, the 64K ROM space is shown as 32K x 16 using 16-bit word size. The widening of the data path between the program ROM and the CPU is another way in which the PIC designers increased the processing power of the PIC18 family.



Program ROM Width for the PIC18

PIC 18 time dealy and delay calculation:-

In creating a time delay using Assembly language instructions, one must be mindful of two factors that can affect the accuracy of the delay:

1. The crystal frequency: The frequency of the crystal oscillator connected to the OSC1 and OSC2 input pins is one factor in the time delay calculation. The duration of the clock period for the instruction cycle is a function of this crystal frequency.
2. The PIC design: Since the 1970s, both the field of IC technology and the architectural design of microprocessors have seen great advancements. Due to the limitations of IC technology and limited CPU design experience for many years, the instruction cycle duration was longer. Advances in both IC technology and CPU design in the 1980s and 1990s have made the single instruction cycle a common feature of many microcontrollers. Indeed, one way to increase performance without losing code compatibility with the older generation of a given family is to reduce the number of instruction cycles it takes to execute an instruction. One might wonder how microprocessors such as PIC are able to execute an instruction in one cycle. There are three ways to do that: (a) Use Harvard architecture to get the maximum amount of code and data into the CPU, (b) use RISC architecture features such as fixed-size instructions, and finally (c) use pipelining to overlap fetching and execution of instructions. We have examined the Harvard and RISC architectures in Chapter 2. Next, we discuss pipelining.

Pipelining:-

In early microprocessors such as the 8085, the CPU could either fetch or execute at a given time. In other words, the CPU had to fetch an instruction from memory, then execute it, and then fetch the next instruction, execute it, and so on. The idea of pipelining in its simplest form is to allow the CPU to fetch and execute at the same time, as shown in Figure 3-9.

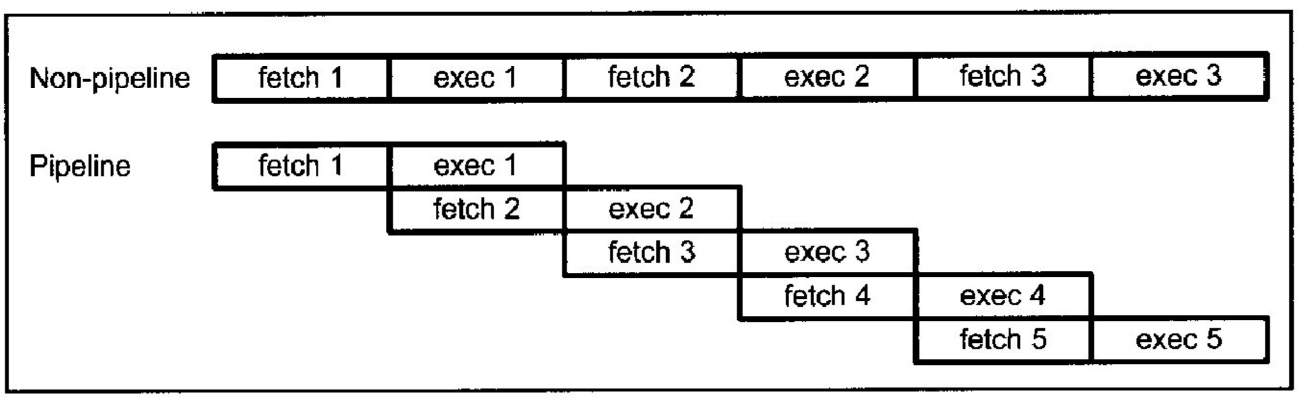


Figure 3-9. Pipeline vs. Non-pipeline

Instruction cylce time for PIC:-

It takes a certain amount of time for the CPU to execute an instruction. In the PIC, this time is referred to as *instruction cycles* (referred to as machine cycles in some other CPUs). Because all the instructions in the PIC18 are either 2-byte or 4-byte, most instructions take no more than one or two instruction cycles to execute. (Notice, however, that some instructions such as BTFSS could take up to three instruction cycles.) Appendix A provides a list of PIC18 instructions and their cycles. In the PIC family, the length of the instruction cycle depends on the frequency of the oscillator connected to the PIC system. The crystal oscillator, along with on-chip circuitry, provide the clock source for the PIC CPU (see Chapter 8). In the PIC18, one instruction cycle consists of four oscillator periods. Therefore, to calculate the instruction cycle for the PIC, we take 1/4 of the crystal frequency, then take its inverse, as shown in Example 3-14.

Example 3-14

The following shows the crystal frequency for three different PIC-based systems. Find the period of the instruction cycle in each case.

(a) 4 MHz (b) 16 MHz (c) 20 MHz

Solution:

(a) 4/4 = 1 MHz; instruction cycle is 1/1 MHz = 1 µs (microsecond)
(b) 16 MHz/4 = 4 MHz; instruction cycle = 1/4 MHz = 0.25 µs = 250 ns (nanosecond)
(c) 20 MHz/4 = 5 MHz; instruction cycle = 1/5 MHz = 0.2 µs = 200 ns

Branch Penalty:-

The overlapping of fetch and execution of the instruction is widely used in today's microcontrollers such as PIC. For the concept of pipelining to work, we need a buffer or queue in which an instruction is prefetched and ready to be executed. In some circumstances, the CPU must flush out the queue. For example, when a branch instruction is executed, the CPU starts to fetch codes from the new memory location and the code in the queue that was fetched previously is discarded. In this case, the execution unit must wait until the fetch unit fetches the new instruction. This is called a branch penalty. The penalty is an extra instruction cycle to fetch the instruction from the target location instead of executing the instruction right below the branch. Remember that the instruction below the branch has already been fetched and is next in line to be executed when the CPU branches to a different address. This means that while the vast majority of PIC instructions take only one instruction cycle, some instructions take two or three instruction cycles. These are GOTO, BRA, CALL, and all the conditional branch instructions such as BNZ, BC, and so on. The conditional branch instruction can take only one instruction cycle if it does not jump. For example, the BNZ will jump if Z = 0 and that takes two instruction cycles. If Z = 1, then it falls through and it takes only one instruction cycle. See Examples 3-15 and 3-16.

Example 3-15

For a PIC18 system of 4 MHz, find how long it takes to execute each of the following instructions:

```
(a) MOVLW        (b)  DECF         (c) MOVWF
(d) ADDLW        (e)  NOP          (f) GOTO
(g) CALL         (h)  BNZ
```

Solution:

The machine cycle for a system of 4 MHz is 1 μs, as shown in Example 3-14. Appendix A shows instruction cycles for each of the above instructions. Therefore, we have:

Instruction	***Instruction cycles***	***Time to execute***
(a) MOVLW 0x55	1	1 ×1 μs = 1 μs
(b) DECF MYREG	1	1 ×1 μs = 1 μs
(c) MOVWF	1	1 ×1 μs = 1 μs
(d) ADDLW	1	1 ×1 μs = 1 μs
(e) NOP	1	1 ×1 μs = 1 μs
(f) GOTO	2	2 ×1 μs = 2 μs
(g) CALL	2	2 ×1 μs = 2 μs
(h) BNZ	2/1	(2 μs taken, 1 μs if it falls through)

Example 3-16

Find the size of the delay of the code snippet below if the crystal frequency is 4 MHz:

Solution:

From Appendix A, we have the following machine cycles for each instruction of the DELAY subroutine:

```
                                    Instruction Cycle
MYREG EQU   0x08          ;use location 08 as counter

DELAY   MOVLW    0xFF               1
        MOVWF    MYREG              1

AGAIN   NOP                         1
        NOP                         1
        DECF     MYREG,F            1
        BNZ      AGAIN              2

        RETURN                      1
```

Therefore, we have a time delay of [(255 × 5) + 1 + 1 + 1] × 1 μs = 1278 μs.
Notice that BNZ takes two instruction cycles if it jumps back, and takes only one when falling through the loop. That means the above number should be 1277 μs.

Delay calculation for PIC:-

As seen in the last section, a delay subroutine consists of two parts: (1) setting a counter, and (2) a loop. Most of the time delay is performed by the body of the loop, as shown in Examples 3-17 and 3-18.

Example 3-18

Find the size of the delay in the following program if the crystal frequency is 4 MHz:

```
MYREG EQU   0x08                    ;use location 08 as counter

        ORG     0
BACK    MOVLW   0x55                ;load WREG with 55H
        MOVWF   PORTB               ;send 55H to port B
        CALL    DELAY               ;time delay
        MOVLW   0xAA                ;load WREG with AA (in hex)
        MOVWF   PORTB               ;send AAH to port B
        CALL    DELAY
        GOTO    BACK                ;keep doing this indefinitely

;——————  this is the delay subroutine
        ORG     300H                ;put time delay at address 300H
DELAY   MOVLW   0xFA                ;WREG = 250, the counter
        MOVWF   MYREG
AGAIN   NOP                         ;no operation wastes clock cycles
        NOP
        NOP
        DECF    MYREG, F
        BNZ     AGAIN               ;repeat until MYREG becomes 0
        RETURN                      ;return to caller
        END                         ;end of asm file
```

Solution:

From Appendix A, we have the following machine cycles for each instruction of the DELAY subroutine:

```
                                  Instruction Cycle

DELAY   MOVLW   0xFA                   1
        MOVWF   MYREG                  1

AGAIN   NOP                            1
        NOP                            1
        NOP                            1
        DECF    MYREG, F               1
        BNZ     AGAIN                  2

        RETURN                         1
```

Therefore, we have a time delay of $[(250 \times 6) + 1 + 1 + 1] \times 1\ \mu s = 1503\ \mu s$.

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

Loop inside a loop delay:-

Another way to get a large delay is to use a loop inside a loop, which is also called a *nested loop*. See Example 3-18. Compare that with Example 3-19 to see the disadvantage of using many NOPs.

Example 3-18

For a instruction cycle of 1 μs, find the time delay in the following subroutine:

```
R2      EQU     0x7
R3      EQU     0x8
DELAY                              Instruction Cycle
        MOVLW      D'200'      1
        MOVWF      R2          1
AGAIN   MOVLW      D'250'      1
        MOVWF      R3          1
HERE    NOP                    1
        NOP                    1
        DECF       R3, F       1
        BNZ        HERE        2
        DECF       R2, F       1
        BNZ        AGAIN       2
        RETURN                 1
```

Solution:

For the HERE loop, we have (5 × 250) 1 μs = 1250 μs. The AGAIN loop repeats the HERE loop 200 times; therefore, we have 200 × 1250 μs = 250000 μs, if we do not include the overhead. However, the following instructions of the outer loop add to the delay:

```
AGAIN   MOVLW      D'250'      1
        MOVWF      R3          1
        .....
        DECF       R2, F       1
        BNZ        AGAIN       2
```

The above instructions at the beginning and end of the AGAIN loop add 5 × 200 × 1 μs = 1000 μs to the time delay. We should also subtract 200 μs for the times BNZ HERE falls through. As a result we have 250000 + 1000 – 200 = 250800 μs = 250.8 milliseconds for the total time delay associated with the above DELAY subroutine. Notice that in the case of a nested loop, as in all other time delay loops, the time is approximate because we have ignored the first few instructions and the last instruction, RETURN, in the subroutine. NOP is a 2-byte instruction. There are 11 instructions in the above DELAY program, and all the instructions are 2-byte instructions. That means that the loop delay takes 22 bytes of ROM code space.

Example 3-19

Find the time delay for the following subroutine, assuming a crystal frequency of 4 MHz. Discuss the disadvantage of this over Example 3-18.

```
MYREG EQU    0x8
                                        Machine Cycle

DELAY     MOVLW     D'200'                    1
          MOVWF     MYREG                     1

AGAIN     NOP                                 1
          NOP                                 1
          NOP                                 1
          NOP                                 1
          NOP                                 1
          NOP                                 1
          NOP                                 1
          NOP                                 1
          NOP                                 1
          NOP                                 1
          NOP                                 1
          NOP                                 1
          DECF      MYREG, F                  1
          BNZ       AGAIN                     2

          RETURN                              1
```

Solution:

The time delay inside the AGAIN loop is $[200(13 + 2)] \times 1\ \mu s = 3000\ \mu s$. NOP is a 2-byte instruction, even though it does not do anything except to waste cycle time. There are 17 instructions in the above DELAY program, and all the instructions are 2-byte instructions. This means the loop delay takes 34 bytes of ROM code space, and gives us only a 3000 μs delay. That is the reason we use a nested loop instead of NOP instructions to create a time delay. Chapter 9 shows how to use PIC timers to create delays much more efficiently.

I/O PORTS

Depending on the device selected, there are up to five general purpose I/O ports available on PIC18FXX8 devices. Some pins of the I/O ports are multiplexed with an alternate function from the peripheral features on the device. In general, when a peripheral is enabled, that pin may not be used as a general purpose I/O pin. Each port has two registers for its operation:

• TRIS register (Data Direction register)
• PORT register (reads the levels on the pins of the device)

PORTA:-

PORTA is a 7-bit wide, bidirectional port. The corresponding Data Direction register is TRISA. Setting a TRISA bit (= 1) will make the corresponding PORTA pin an input. Clearing a TRISA bit (= 0) will make the corresponding PORTA pin an output.

The RA4 pin is multiplexed with the Timer0 module clock input to become the RA4/T0CKI pin. The other PORTA pins are multiplexed with analog inputs and the analog VREF+ and VREF- inputs. The operation of each pin is selected by clearing/setting the control bits in the ADCON1 register (A/D Control Register 1). On a Power-on Reset, these pins are configured as analog inputs and read as '0'.

PORTB:-

PORTB is an 8-bit wide, bidirectional port. The corresponding Data Direction register is TRISB. Setting a TRISB bit (= 1) will make the corresponding PORTB pin an input. Clearing a TRISB bit (= 0) will make the corresponding PORTB pin an output.

Four of the PORTB pins (RB7:RB4) have an interruption- change feature.

PORTC:-

PORTC is an 8-bit wide, bidirectional port. The corresponding Data Direction register is TRISC. Setting a TRISC bit (= 1) will make the corresponding PORTC pin an input. Clearing a TRISC bit (= 0) will make the corresponding PORTC pin an output.

PORTC is multiplexed with several peripheral functions.

Pin	Multiplexed	Function
RC0	Yes	Timer1 Oscillator for Timer1/Timer3
RC1	Yes	Timer1 Oscillator for Timer1/Timer3
RC2	No	-----
RC3	Yes	SPI™/I2C™ Master Clock
RC4	Yes	I2C Data Out
RC5	Yes	SPI Data Out
RC6	Yes	USART Async Xmit, Sync Clock
RC7	Yes	USART Sync Data Out

PORTD:-

PORTD is an 8-bit wide, bidirectional port. The corresponding Data Direction register for the port is TRISD. Setting a TRISD bit (= 1) will make the corresponding PORTD pin an input. Clearing a TRISD bit (= 0) will make the corresponding PORTD pin an output

PORTD can be configured as an 8-bit wide, microprocessor port (Parallel Slave Port or PSP. PORTD is also multiplexed with the analog comparator module and the ECCP module.

PORTE:-

PORTE is a 3-bit wide, bidirectional port. PORTE has three pins (RE0/AN5/RD, RE1/AN6/WR/C1OUT and RE2/AN7/CS/C2OUT which are individually configurable as inputs or outputs. The corresponding Data Direction register for the port is TRISE. Setting a TRISE bit (= 1) will make the corresponding PORTE pin an input. Clearing a TRISE bit (= 0) will make the corresponding PORTE pin an output.

The TRISE register also controls the operation of the Parallel Slave Port through the control bits in the upper half of the register. When the Parallel Slave Port is active, the PORTE pins function as its control inputs. PORTE pins are also multiplexed with inputs for the A/D converter and outputs for the analog comparators. When selected as an analog input, these pins will read as ‘0’s.

I/O PORT programming:-

Initializing a PORT:-

```
CLRF PORTA        ; Initialize PORTA by clearing output data latches
CLRF PORTB        ; Initialize PORTB by clearing output data latches
CLRF PORTC        ; Initialize PORTC by clearing output data latches
MOVLW 0CFh        ; Move CF h into WREG
MOVWF TRISA       ; Move WREG to TRISA; Set RA3:RA0 as inputs; RA5:RA4 as outputs
CLRF TRISB        ; Clear TRISB register; make PORTB as output
SETG TRISC        ; Setting TRISC register; make PORTC as input
```

Toggling PORT:-

Following code will send 55h and AAh continuously to PORTD or toggle the PORTD

```
      CLRF PORTD        ; Initialize PORTD by clearing output data latches
      CLRF TRISD        ; Clear TRISD register; make PORTD as output
Back: MOVLW 55h         ; Move 55h to WREG register
      MOVWF PORTD       ; Move WREG to PORTD; i.e. PORTD=55h
      CALL Delay        ; time delay
      MOVLW 0AAh        ; Move AAh to WREG register
      MOVWF PORTD       ; Move WREG to PORTD; i.e. PORTD=AAh
      CALL Delay        ; time delay
      GOTO Back
```

Write a test program for the PIC18 chip to toggle all the bits of PORTB, PORTC, and PORTD every 1/4 of a second. Assume a crystal frequency of 4 MHz.

Solution:

```
;tested with MPLAB for the PIC18F458 and XTAL = 4 MHz

      list P=PIC18F458
#include P18F458.INC

R1 equ 0x07
R2 equ 0x08

ORG 0
      CLRF   TRISB        ;make Port B an output port
      CLRF   TRISC        ;make Port C an output port
      CLRF   TRISD        ;make Port D an output port
      MOVLW  0x55         ;WREG = 55h
      MOVWF  PORTB        ;put 55h on Port B pins
      MOVWF  PORTC        ;put 55h on Port C pins
      MOVWF  PORTD        ;put 55h on Port D pins
L3    COMF   PORTB,F      ;toggle bits of Port B
      COMF   PORTC,F      ;toggle bits of Port C
      COMF   PORTD,F      ;toggle bits of Port D
      CALL   QDELAY       ;quarter of a second delay
      BRA    L3


;-----------1/4 SECOND DELAY
QDELAY
      MOVLW D'200'
      MOVWF R1
D1    MOVLW D'250'
      MOVWF R2
D2    NOP
      NOP
      DECF R2, F
      BNZ D2
      DECF R1, F
      BNZ D1
      RETURN
      END
```

Calculations:

4 MHz / 4 = 1 MHz
1 / 1 MHz = 1 μs
Delay = 250 × 200 × 5 MC × 1 μs = 250,000 μs (if we include the overhead, we will have 250,800. See Example 3-17 in the previous chapter.)

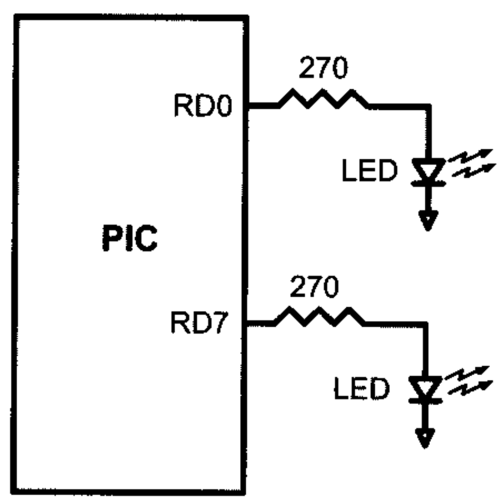
I/O bit manipulation:-

Sometimes we need to access only 1 or 2 bits of the port instead of the entire 8 bits. The PIC I/O ports have capability to access individual bits of the port without alternating the rest of the bits in that port.

An LED is connected to each pin of Port D. Write a program to turn on each LED from pin D0 to pin D7. Call a delay module before turning on the next LED.

Solution:

```
CLRF   TRISD        ;make PORTD an output port
BSF    PORTD,0      ;bit set turns on RD0
CALL   DELAY        ;delay before next one
BSF    PORTD,1      ;turn on RD1
CALL   DELAY        ;delay before next one
BSF    PORTD,2
CALL   DELAY
BSF    PORTD,3
CALL   DELAY
BSF    PORTD,4
CALL   DELAY
BSF    PORTD,5
CALL   DELAY
BSF    PORTD,6
CALL   DELAY
BSF    PORTD,7
CALL   DELAY
```



Write the following programs:
(a) Create a square wave of 50% duty cycle on bit 0 of Port C.
(b) Create a square wave of 66% duty cycle on bit 3 of Port C.

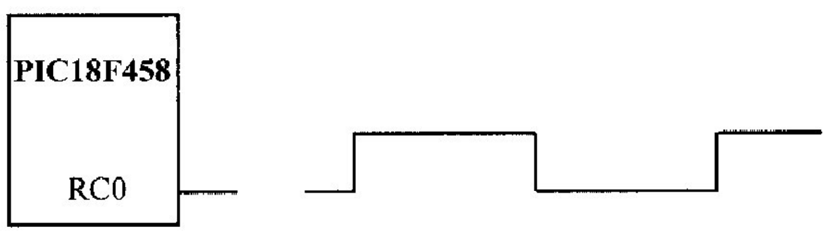
Solution:

(a) The 50% duty cycle means that the "on" and "off" states (or the high and low portions of the pulse) have the same length. Therefore, we toggle RC0 with a time delay between each state.

```
        BCF     TRISC,0       ;clear TRIS bit for RC0 = out
HERE    BSF     PORTC,0       ;set to HIGH RC0 (RC0 = 1)
        CALL    DELAY         ;call the delay subroutine
        BCF     PORTC,0       ;RC0 = 0
        CALL    DELAY
        BRA     HERE          ;keep doing it
```

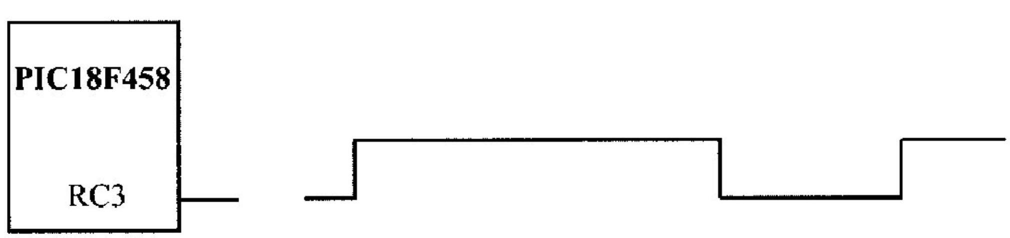
Another way to write the above program is:

```
        BCF     TRISC,0       ;make RC0 = out
HERE    BTG     PORTC,0       ;complement bit 0 of PORTC
        CALL    DELAY         ;call the delay subroutine
        BRA     HERE          ;keep doing it
```



(b) A 66% duty cycle means that the "on" state is twice the "off" state.

```
        BCF     TRISC,3       ;clear TRISC3 bit for output
BACK    BSF     PORTC,3       ;RC3 = 1
        CALL    DELAY         ;call the delay subroutine
        CALL    DELAY         ;twice for 66%
        BCF     PORTC,3       ;RC3 = 0
        CALL    DELAY         ;call delay once for 33%
        BRA     BACK          ;keep doing it
```



Write a program to perform the following:

(a) Keep monitoring the RB2 bit until it becomes HIGH;

(b) When RB2 becomes HIGH, write value 45H to Port C, and also send a HIGH-to-LOW pulse to RD3.

Solution:

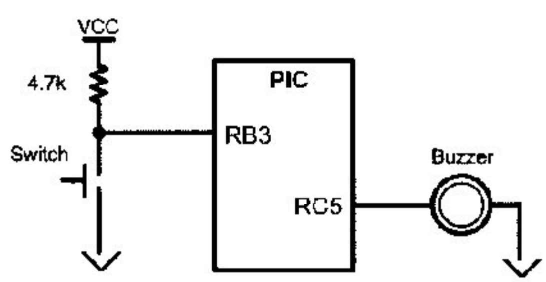
```
      BSF    TRISB,2      ;make RB2 an input
      CLRF   TRISC        ;make PORTC an output port
      BCF    PORTD,3      ;make RD3 an output
      MOVLW  0x45         ;WREG = 45h
AGAIN BTFSS  PORTB,2      ;bit test RB2 for HIGH
      BRA    AGAIN        ;keep checking if LOW
      MOVWF  PORTC        ;issue WREG to Port C
      BSF    PORTD,3      ;bit set fileReg RD3 (H-to-L)
      BCF    PORTD,3      ;bit clear fileReg RD3 (L)
```

In this program, instruction "`BTFSS PORTB,2`" stays in the loop as long as RB2 is LOW. When RB2 becomes HIGH, it skips the branch instruction to get out of the loop, and writes the value 45H to Port C. It also sends a HIGH-to-LOW pulse to RD3.

Assume that bit RB3 is an input and represents the condition of a door alarm. If it goes LOW, it means that the door is open. Monitor the bit continuously. Whenever it goes LOW, send a HIGH-to-LOW pulse to port RC5 to turn on a buzzer.

Solution:

```
      BSF    TRISB,3      ;make RB3 an input
      BCF    TRISC,5      ;make RC5 an output
HERE  BTFSC  PORTB,3      ;keep monitoring RB3 for HIGH
      BRA    HERE         ;stay in the loop
      BSF    PORTC,5      ;make RC5 HIGH
      BCF    PORTC,5      ;make RC5 LOW for H-to-L
      BRA    HERE
```



BCD (binary coded decimal) number system

BCD stands for *binary coded decimal*. BCD is needed because in everyday life we use the digits 0 to 9 for numbers, not binary or hex numbers. Binary representation of 0 to 9 is called BCD (see Figure 5-1). In computer literature, one encounters two terms for BCD numbers: (1) unpacked BCD, and (2) packed BCD. We describe each one next.

Digit	BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Figure 5-1. BCD Code

Unpacked BCD

In unpacked BCD, the lower 4 bits of the number represent the BCD number, and the rest of the bits are 0. Example: "0000 1001" and "0000 0101" are unpacked BCD for 9 and 5, respectively. Unpacked BCD requires 1 byte of memory, or an 8-bit register, to contain it.

Packed BCD

In packed BCD, a single byte has two BCD numbers in it: one in the lower 4 bits, and one in the upper 4 bits. For example, "0101 1001" is packed BCD for 59H. Only 1 byte of memory is needed to store the packed BCD operands. Thus one reason to use packed BCD is that it is twice as efficient in storing data.

ASCII numbers

On ASCII keyboards, when the key "0" is activated, "011 0000" (30H) is provided to the computer. Similarly, 31H (011 0001) is provided for key "1", and so on, as shown in Table 5-3.

It must be noted that BCD numbers are universal although ASCII is standard in the United States (and many other countries). Because the keyboard, printers, and monitors all use ASCII, how does data get converted from ASCII to BCD, and vice versa? These are the subjects covered next.

Table 5-3: ASCII and BCD Codes for Digits 0–9

Key	ASCII (hex)	Binary	BCD (unpacked)
0	30	011 0000	0000 0000
1	31	011 0001	0000 0001
2	32	011 0010	0000 0010
3	33	011 0011	0000 0011
4	34	011 0100	0000 0100
5	35	011 0101	0000 0101
6	36	011 0110	0000 0110
7	37	011 0111	0000 0111
8	38	011 1000	0000 1000
9	39	011 1001	0000 1001

Packed BCD to ASCII conversion

In many systems we have what is called a *real-time clock* (RTC). The RTC provides the time of day (hour, minute, second) and the date (year, month, day) continuously, regardless of whether the power is on or off (see Chapter 16). This data, however, is provided in packed BCD. For this data to be displayed on a device such as an LCD, or to be printed by the printer, it must be in ASCII format.

To convert packed BCD to ASCII, you must first convert it to unpacked BCD. Then the unpacked BCD is tagged with 011 0000 (30H). The following demonstrates converting packed BCD to ASCII. See also Example 5-32.

Packed BCD	*Unpacked BCD*	*ASCII*
29H	02H & 09H	32H & 39H
0010 1001	0000 0010 &	0011 0010 &
	0000 1001	0011 1001

Assume that register WREG has packed BCD. Write a program to convert packed BCD to two ASCII numbers and place them in file register locations 6 and 7.

Solution:

```
BCD_VAL EQU 0x29
L_ASC   EQU 0x06  ;set aside file register location
H_ASC   EQU 0x07  ;set aside file register location

        MOVLW BCD_VAL     ;WREG = 29H, packed BCD
        ANDLW 0x0F        ;mask the upper nibble (W = 09)
        IORLW 0x30        ;make it an ASCII, W = 39H ('9')
        MOVWF L_ASC       ;save it (L_ASC = 39H ASCII char)
        MOVLW BCD_VAL     ;W = 29H get BCD data once more
        ANDLW 0xF0        ;mask the lower nibble (W = 20H)
        SWAPF WREG,W      ;swap nibbles (WREG = 02H)
        IORLW 0x30        ;make it an ASCII, W = 32H ('2')
        MOVWF H_ASC       ;save it (H_ASC = 32H ASCII char)
```

ASCII to packed BCD conversion

To convert ASCII to packed BCD, you first convert it to unpacked BCD (to get rid of the 3), and then combine it to make packed BCD. For example, for 4 and 7 the keyboard gives 34 and 37, respectively. The goal is to produce 47H or "0100 0111", which is packed BCD. This process is illustrated next.

Key	*ASCII*	*Unpacked BCD*	*Packed BCD*
4	34	00000100	
7	37	00000111	01000111 which is 47H

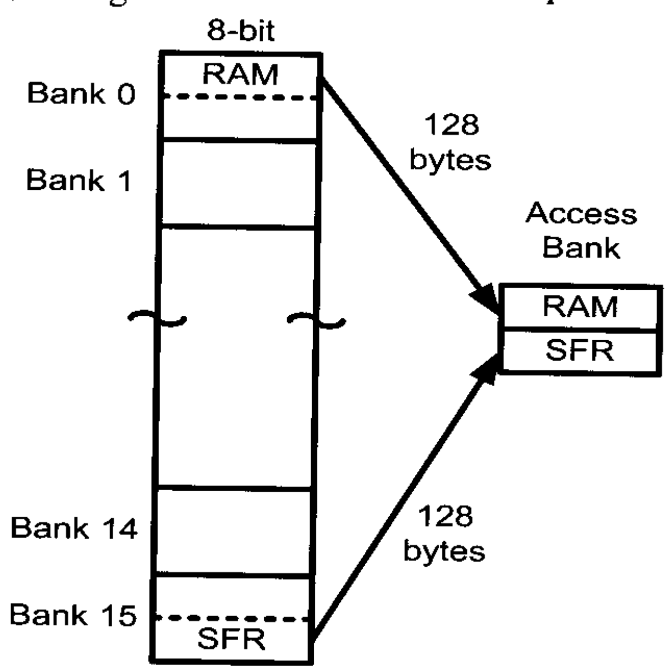
```
MYBCD EQU 0x20    ;set aside location in file register

      MOVLW A'4'        ;WREG = 34H, hex for ASCII char 4
      ANDLW 0x0F        ;mask upper nibble (WREG = 04)
      MOVWF MYBCD       ;save it in MYBCD loc
      SWAPF MYBCD,F     ;MYBCD = 40H
      MOVLW A'7'        ;WREG = 37H, hex for ASCII char 7
      ANDLW 0x0F        ;mask upper nibble (WREG = 07)
      IORWF MYBCD,F     ;MYBCD = 47H, a packed BCD
```

After this conversion, the packed BCD numbers are processed and the result will be in packed BCD format. As we saw earlier in this chapter, a special instruction, "DAW", requires that the data be in packed BCD format.

Bank switching:-

The PIC18 microcontroller has a maximum of 4K of data RAM space. Although not all members of the family have the entire RAM installed, every member of the family has at least the access bank for the file register. The file register RAM is divided into banks of 256 bytes each, which gives us a total of 16 banks in the PIC18. The minimum bank that every PIC18 has is called the access bank, as we discussed in Chapter 2. The access bank is made of 128 bytes of lower addresses and 128 bytes of higher addresses. While the lower 128 bytes of address space 000–07FH are used for general-purpose RAM, the higher 128 bytes are dedicated to the SFRs (special function registers) residing in address space F80–FFFH. The vast majority of the PIC18 chips we see on the Microchip web site have more than just the access bank. In this section we show how to use bank switching to take advantage of the entire data RAM space of the PIC18.



The BSR register for bank switching:-

we use the BSR (bank select register) to choose the desired bank. The BSR is an 8-bit register and is part of the SFRs. Of the 8 bits of the BSR, only 4 least-significant bits are used in the PIC18. The upper 4 bits are set to zero and are ignored by the PIC18. The 4-bit BSR gives us 16 banks, and because each bank is 256 bytes, we cover the entire 4096 ($16 \times 256 = 4096$) bytes of the data RAM file register using bank switching. The 4K (4096) bytes of the data RAM are organized as banks 0 to F, where the lowest bank, 0, has an address of 00–FFH, and the highest bank is bank F with the addresses of F00–FFFH. In the PIC18, the last 128 bytes of bank F are always set aside for the SFRs, while general purpose registers always start at address 0 of bank 0. Upon power-on reset, BSR = 0 (0000 binary), which indicates that only the lowest addresses of data RAM, from 000 to 0FFH, can be used for the general-purpose register in addition to the SFRs, which always reside in the last half of bank F. Similarly, if we make BSR = 1 (0001 binary), then PIC18 selects bank 1 using the 100–1FFH addresses in addition to the SFRs, which use only the last half of the bank with addresses of F80–FFFH. To select bank 2, we load BSR with the value 02 (0010 binary), which allows access to the bank addresses 200–2FF

Write a program to copy the value 55H into RAM memory locations 340H to 345H using:

(a) direct addressing mode.
(b) a loop.

Solution:
(a)

```
MOVLB  0x3           ;BANK 3
MOVLW  0x55          ;load WREG with value 55H
MOVWF  0x40, 1       ;copy WREG to RAM location 340H
MOVWF  0x41, 1       ;copy WREG to RAM location 341H
MOVWF  0x42, 1       ;copy WREG to RAM location 342H
MOVWF  0x43, 1       ;copy WREG to RAM location 343H
MOVWF  0x44, 1       ;copy WREG to RAM location 344H
```

(b)

```
COUNT EQU 0x10          ;loc 10h
      MOVLB  0x3        ;BANK 3
      MOVLW  0x5        ;WREG = 5
      MOVWF  COUNT      ;load the counter, count = 5
      LFSR   0,0x340    ;load pointer. FSR0 = 40H, RAM address
      MOVLW  0x55       ;WREG = 55h value to be copied
B1    MOVWF  INDF0,0    ;copy WREG to RAM loc FSR0 points to
      INCF   FSR0L      ;increment FSR0L pointer
      DECF   COUNT,F,0  ;decrement the counter
      BNZ    B1         ;loop until counter = zero
```

The following shows RAM contents after the above program is run:

340 = (55)
341 = (55)
342 = (55)
343 = (55)
344 = (55)