Chapter 3: Instruction Sets

Instruction sets could be said of programmer's interface to hardware. The CPU of the PIC is responsible for using the instructions (or program code) stored in the program memory to execute the functions and operations the instructions intend to do. The instructions are stored in the program memory in a format of machine code, or hex code. Assembly language is the instruction mnemonics for the machine codes. Assembler generates machine codes from Assembly code.

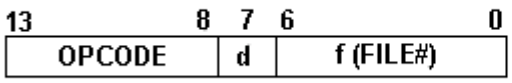
1. PIC16F877 Instruction

Each PIC16F877 instruction is a 14-bit word divided into an OPCODE, which specifies the instruction type, and one or more operands which further specify the operation of the instruction. There are three addressing modes in the 16F877 instruction: byte-oriented, bit-oriented, and literal and control operations. Byte-oriented instructions operate with a whole-byte data, like moving a byte of data in **W** register to a file register. Bit-oriented instructions are to check or change only a bit of a byte data. Literal operations involve with direct numerical value of loading or logical operation with **W** register. The literal operation is usually called 'an immediate addressing mode' in the more traditional microprocessor instructions.

Byte-oriented instructions

For byte-oriented instructions, 'f' represents a file register designator and 'd' represents a destination designator. The file register designator is used to specify which one of the file registers is to be used by the instruction. The destination designator specifies where the result of the operation is to be placed. If 'd' is '0', the result is placed in the **W** register. If 'd' is '1', the result is placed in the file register specified in the instruction. If we do not specify the destination, it is considered '1' (default value).

In the machine code level (in 14-bit word configuration), byte-oriented instructions are configured with 6 bits of Opcode, 1 bit for destination designator, followed by 7 bit file register address.



d=0 for desitination W
d=1 for destination f (default)
f= 7-bit file register address

The table below lists the instructions words of the byte-oriented operation. As we see at the last column of the table, the 6-bit Opcode portion is already given with a specific bit formation for each instruction mnemonic. The remaining 8 bits are determined by the destination of the operation and the file register the operation accesses to do the operation. The column with 'T' indicate the number instruction cycles needed to do the operation of each instruction.

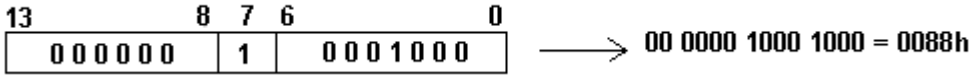
Table. Byte-oriented operation of 16F877 instructions.

Mnemonic	Description	T	Flag	Instruction word (OPCODE+operand)
addwf f,d	add W and f	1	C, Z	00 0111 dfff ffff
andwf f,d	and W with f	1	Z	00 0101 dfff ffff
clrf f	clear f (i.e., f=0)	1	Z	00 0001 1fff ffff
clrw	clear W	1	Z	00 0001 0xxx xxxx
comf f,d	complement f	1	Z	00 1001 dfff ffff

decf f,d	decrease f by 1	1	Z	00 0011 dfff ffff
decfsz f,d	decrease f by 1, skip if f=0	1(2)		00 1011 dfff ffff
incf f,d	increase f by 1	1	Z	00 1010 dfff ffff
incfsz f,d	increase f by 1, skip if f=0	1(2)		00 1111 dfff ffff
iorwf f,d	OR W with f	1	Z	00 0100 dfff ffff
movf f,d	move f	1	Z	00 1000 dfff ffff
movwf f	move W to f	1		00 0000 1fff ffff
nop	no operation	1		00 0000 0xx0 0000
rlf f,d	rotate left f through carry	1	C	00 1101 dfff ffff
rrf f,d	rotate right f through carry	1	C	00 1100 dfff ffff
subwf f,d	subtract W from f (i.e., f-W)	1	C, Z	00 0010 dfff ffff
swapf f,d	swap nibbles in f	1		00 1110 dfff ffff
xorwf f,d	XOR W with f	1	Z	00 0110 dfff ffff

Let's have an example of machine code generation from an instruction. Let's consider then the following instruction:
`movwf PORTD,` which moves a content in W register to PORTD register.

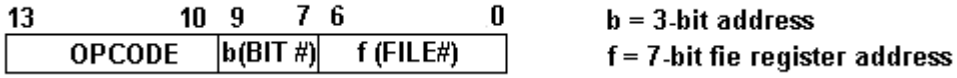
Since the OPCODE for `movwf` is set 000000 (see the byte-oriented operation table above), the destination is the default value of '1', and the file register address of PORTD is 0x08 (from the file register table in page 7), the corresponding machine code is 0088h:



If the file register is changed to PORTB, since the file register address of PORTB is 0x06, the corresponding machine code would be: 0086h.

Bit-oriented instructions

For bit-oriented instructions, 'b' represents a bit field designator which selects the number of the bit affected by the operation, while 'f' represents the file register in which the bit is located. In the machine code level (in 14-bit word configuration), bit-oriented instructions are configured with 4 bits of Opcode, 3 bits for the bit number (between 000b and 111b), followed by 7 bit file register address.



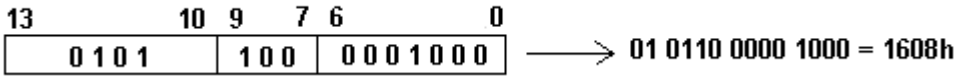
The table below lists the instructions words of the bit-oriented operations. As before, at the last column of the table, the 4-bit Opcode portion is already given with a specific bit formation for each instruction mnemonic. The remaining 10 bits are determined by the 3-bit bit number and the 7-bit file register the operation accesses to do the operation.

Table. Bit-Oriented Instruction for 16F877

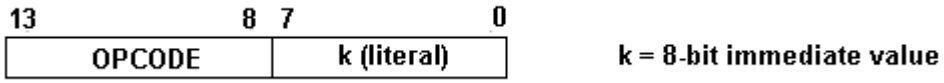
Mnemonic	Description	T	Flag	Instruction word (OPCODE+operand)
bcf f,b	clear f<b> bit (i.e., f<b)=0)	1		01 00bb bfff ffff
bsf f,b	set f<b> bit (i.e., f<b)=1)	1		01 01bb bfff ffff
btfsc f,b	test f<b> bit, skip if f<b>=0	1(2)		01 10bb bfff ffff
btfss f,b	test f<b> bit, skip if f<b>=1	1(2)		01 11bb bfff ffff

Let's have an example of machine code generation from a bit-oriented instruction. Let's consider the following instruction:
`bsf PORTD, 0x04`, which sets (or make '1') the fourth bit of the file register PORTD.

The Opcoe for `bsf` is already configured as 0101. The 3-bit bit number, which is 4 in decimal, is 100b, fill the next 3 bits of the instruction word. Lastly, the file register address for PORTD, which is 0x08 (or 0001000b) fills the last 7 bits of the word, to make the machine code 1608h.



Literal and Control Operations
For literal and control operations, 'k' represents an 8 or 9-bit constant or literal value. In the machine code level in 14-bit word configuration, they are configured with 6 bits of Opcode followed by 8 bit constant (or literal).



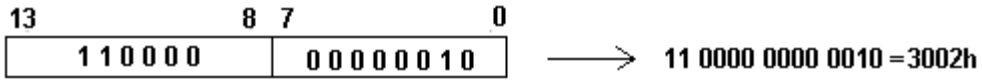
The table below lists the instructions words of the literal and control operations. As before, at the last column of the table, but unlike the previous two operations, the number of bits assigned to Opcode is fixed: some has 5, another 6, and others 3. The x marked bits in the Opcode are don't care values (1 or 0). In most cases, the Opcode is followed by a 8 bit literal.

Table. Literal and Control Operations of 16F877 Instructions

Mnemonic		Description	T	Flag	Instruction word (OPCODE+operand)
addlw	k	add k and W	1	C, Z	11 111x kkkk kkkk
andlw	k	and k and W	1	Z	11 1001 kkkk kkkk
call	k	call subroutine at address k	2		10 0kkk kkkk kkkk
clrwdt		clear watchdog timer	1		00 0000 0110 0100
goto	k	go to address k	2		10 1kkk kkkk kkkk
iorlw	k	OR k with W	1	Z	11 1000 kkkk kkkk
movlw	k	move k to W (i.e., W=k)	1		11 00xx kkkk kkkk
retfie		return from interrupt	2		00 0000 0000 1001
retlw	k	return with k in W	2		11 01xx kkkk kkkk
return		return from subroutine	2		00 0000 0000 1000
sleep		go into standby mode	1		00 0000 0110 0011
sublw	k	subtract W from k (i.e.,k-W)	1	C, Z	11 110x kkkk kkkk
xorlw	k	XOR k with W	1	Z	11 1010 kkkk kkkk

Let's have a simple example of machine code generation from a bit-oriented instruction. Let's consider the following instruction:
`movlw 0x02`, which loads a constant value of 2h to **W** register.

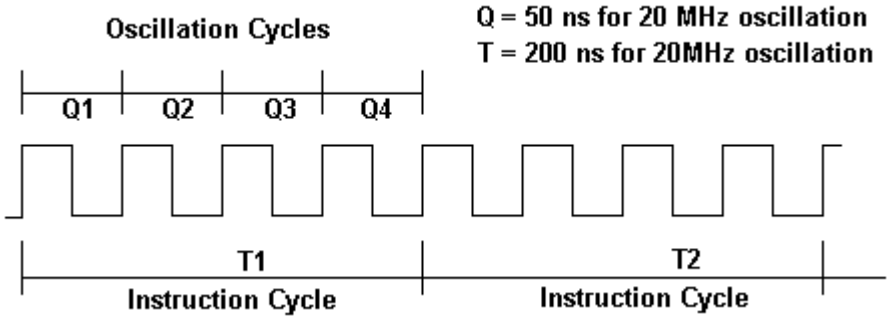
From the table, the Opcode for `movlw` is 110000, and the literal value component must be 00000010 (2h), which makes the corresponding machine code as 3002h.



2. Instruction Cycle and Execution Time

The above three tables showed the Mnemonics of 16F877 instructions and machine code formation. In addition, they indicated, under the 'T' column, the number of instruction cycle of the instructions. And we see that all instructions are executed within one single instruction cycle, unless a conditional test is true or the program counter is changed as a result of an instruction, in which the execution takes two instruction cycles.

By the way, one instruction cycle consists of four oscillator periods. Thus, for an oscillator frequency of 20 MHz, the normal instruction execution time is 0.2 μs (1μs = one millionth second). If a conditional test is true or the program counter is changed as a result of an instruction, the instruction execution time is 0.4 μs.



3. Coding Practice - Tricks and Tips

Let's have some practice of instruction sets. As we see that there are not many instructions, especially, arithmetic operations. So we must build some skills and tricks to combine the instructions for a desired operation. Here we will devise some tricks and tips for frequently met cases in embedded computing and PIC microcontroller interface for monitoring and control.

<u>Turn on/off an LED</u>

An LED is connected to a pin at one, say, PORTB<0>, and to a ground at the other.

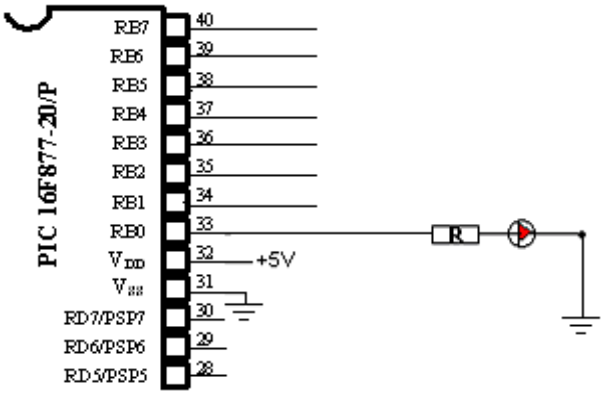


Fig 8. An LED connection to PORTB<0> of 16F877

After assigning PORTB<0> as output port by loading a proper value to TRISB register, we set or reset the single bit of PORTB<0> to turn on or off the LED. We use a bit-oriented file register operation as below for turning on/off:

```
bsf PORTB, 0x00 ;LED on
bcf PORTB, 0x00 ;LED off.
```

A small problem with the above two lines to turn on and off an LED is that you do not see it blink at all: the time between on and off is too short, 0.2μs precisely, to be recognized by eyes. To make it blink, we have to have some time delay between the lines. Making time delay, without using the Timer of 16F877, is discussed in this section. Making time delay using the built-in Timer is a subject of a separate chapter.

Variable Declaration

When we program in C, declaring and defining a variable is not even an issue. We declare a variable with a name and size like,

```
byte  temp
int   x
```

However, in PIC programming environment, the variable must physically, overtly, occupy the data memory (RAM) area. To declare variables whose values can be changed freely, we use `CBLOCK` and `ENDC` pair of MPLAB (which is the integrated Assembly programming, debugging, and simulation environment, and is a subject of another separate chapter.) directive. So to declare the above two variables as in C, we have to have the following lines before any line of PIC assembly code. Remember that 16F877 is an 8-bit microcontroller and any variable declared will be treated as an 8-bit file register, specifically general file register. The following program declares the two variables at the general purpose register area in bank 0.

```
CBLOCK  0x20      ;the starting address of the general purpose
                  ;register block is 20h
        temp      ;temp is a file register at address 20h
        x         ;x is a file register defined at address 21h
ENDC              ;end of a file register block
```

In the above variable declaration and its address in the RAM, we see that the block starts at 20h. You may have a question asking why 20h? Why not 10h? There is a good reason why the above example block starts from 20h: If you go again to the filer register map in page 7, you can see that there are four blocks (one block a bank) in four blanks of general purpose register, which is available for programmer. In bank 0, the block is assigned from address 20h to 7Fh. In bank 1, the user block is between A0h and EFh (not that it is not to FFh since F0h - FFh is overlapped with 70h-7Fh of the bank 1 block). In banks 2 and 3, the user block is much wider: 110h - 170h for bank2, and 190h - 1F0h for bank 3.

If your special purpose registers are, for example, in bank 1, you naturally use a block in bank 1 to declare your variables since accessing registers in the same bank is much easier in code

writing: it does not require to move to different banks. However, if your code accesses I/O port which are in bank 0 (except that the port's direction designation registers in bank 2, which you can work out before accessing the port), there is no reason not to have your variable declaration block in bank 0.

Content Check

Here the question is how we check or compare a byte content stored in **W** register or a file register. This seemingly easy question in a high level language environment like C is legitimate since there is no such instruction of `IF...AND IF...END` structure in PIC Assembly language. Therefore in microcomputer coding, we always look a pattern of the data bits, instead of the value of them, in a file register. Let's consider an example, which we treat thoroughly in a later chapter, of simple remote control application. Assume that an Infrared (IR) receiver is connected to a pin of a port and that it reads transmitted IR command sent from an IR remote controller, like our TV or VCR remote.

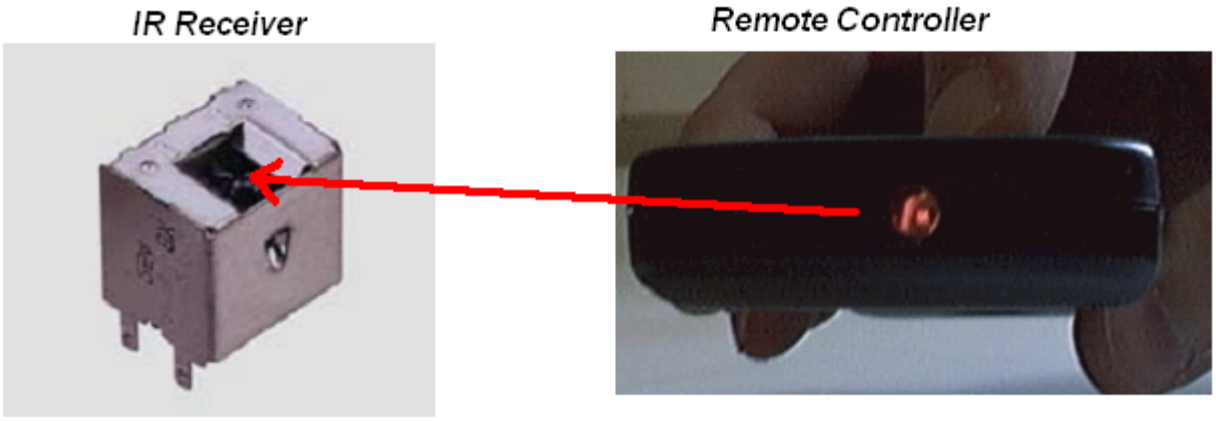


Fig 9. Remote Controller and an IR receiver pair

The command code part of the data from a Sony remote controller is tabulated in Table. Let's further simplify our situation by assuming that a byte information sent from the remote is, after a decoding process which is actually the main subject when we discuss on the IR remote control application, stored to **W** register. Now we want to check the IR command by examining IR command.

Table. IR command code for a Sony Remote Controller

IR command code	Button pressed from a remote
0 0 0 0 0 0 0 0	'1'
0 0 0 0 0 0 0 1	'2'
0 0 0 0 0 0 1 0	'3'
0 0 0 0 0 0 1 1	'4'
0 0 0 0 0 1 0 0	'5'
0 0 0 0 0 1 0 1	'6'
0 0 0 0 0 1 1 0	'7'
0 0 0 0 0 1 1 1	'8'
0 0 0 0 1 0 0 0	'9'

0 0 0 0 1 0 0 1	'0'

Let's further simplify our example by assuming that a byte information sent from the remote is, after a decoding process which is actually the main subject when we discuss on the IR remote control application, stored to **W** register. Now we want to check the IR command by examining **W** register.

So our job is to check the content of **W** register to find what command the remote controller sent to the IR receiver. Since **W** register should be checked at least 10 times to find the pressed button, and the checking may alter the original content, we have to save and store the original content to a memory location (RAM area). As we discussed above, let's define a general purpose register `COMreg` at bank 0 by declaring the register in the bank 0:

```
      CBLOCK  0x20
              COMreg
      ENDC
```

Now let's focus on the content check. First we move **W** to COMreg for a keepsake. And we restore to **W** every time we check the content. Let's assume that **W** now holds the original content. The content could be any combination of 8 bits out of 2^8=256 possibilities. And the number of our patterns of interest are only 10. If you check every bit to match the 10 patterns, you would have to check 256 checks for all 8 bits. This approach is not good. Then, what is a simpler one?

Let's explore `andlw k` instruction. The code `andlw 0xFF` keeps the content of **W** intact by the logical AND operation between **W** and the constant value FFh, since whatever byte data you have the AND operation with 1s will keep 1 as 1 and 0 as 0. On the other hand, `andlw 0x00` will clear **W**, and set the Z (zero flag) of STATUS register, since any bit AND-ed with 0 will cleared. The Z flag is the second bit of the STATUS register. Therefore, if the second bit of STATUS register is 1 (which says the result is zero) after the instruction `andlw 0xFF`, then we know that the content of **W** is 00h or 00000000b.

How about other numbers? And, `andlw 0xFE` or `andlw B'11111110'` would keep all the bits except the LSB (least significant bit), i.e., bit 0 of **W**. Therefore, if the result (by the Z flag) of `andlw 0xFE` is zero, then we know that **W** is either 00h=00000000b or 01h=00000001b. But the pattern of 00h=00000000b is already checked above, we can assume that W is 01h=00000001h.

Further, if the result of `andlw 0xFC` or `andlw B'11111100'` is zero, then we can see that the content of **W** is 03h=00000011b.

Now we can write a sample code to decode the remote.

```
;sample code for IR command decoding
;It is assumed that 1 byte IR information is stored in COMreg register

            list P = 16F877         ;Target processor = 16F877
Z     EQU   0x02        ;Z flag, STATUS<2>

      CBLOCK  0x20      ;declaration of COMreg register in bank 0
```

```
            COMreg
      ENDC

      clrf  STATUS            ;cleared the STATUS register
                              ;mainly to clear Z flag
      movf  COMReg,0          ;COMreg holds the IR command byte
                              ;and we bring the content to W
      andlw B'11111111'
      btfss STATUS, Z         ;W=0? (button 1)
      goto  next1             ;no.  Check other possibilities
      goto  One               ;Yes.  Button 1 is pressed.
next1 movf  COMreg,0          ; Retrieve the original byte from file register
      andlw B'11111110'
      btfss STATUS,Z          ;W=1? (button 2)
      goto  next2             ;No
      goto  Two               ;Yes.  Button 2 is pressed.
next2 movf  COMreg,0
      andlw B'11111101'
      btfss STATUS,Z          ;W=2? (button 3)
      goto  next3
      goto  Three             ;Yes. Button 3 is pressed
next3 movf  COMreg, 0
      andlw B'11111100'       ;
      btfss STATUS, ZERO      ;W=3? (button 4)
      goto  next4
      goto  Four
next4 movf  COMreg,0
```

The code continues on to find other buttons. The instruction `btfss` is to "***b***it ***t***est of a ***f***ile register, and ***s***kip if the bit is ***s***et." In other words, in the above code, if the Z flag is 1 ("set"), skip the next line and go to the second line after the instruction. If the Z flag is 0 ("reset"), just go to the immediate next line after the instruction. We will discuss more on `btfss` in the next example of coding. Also remember from the code that there is only one instruction to move a data from a file register to **W** register: `movf` with direction 0 as in

```
movf  COMreg,0
```

Monitoring Digital Input and Action on It

This case is when we receive a sensor input and act on the input. Usually the sensor is a digital device which generates a digital output, say +5V (High) for normal situation and 0V (Low) output for abnormal situation, or vice versa. In practice, let's consider a motion detection and warning system. We use a PIR (pyro-IR) motion detector to detect a motion and, once a motion is detected, we will alarm by turning on a buzzer. The motion detector's output is normal High (which means it generates High (+5V) signal when motion is not detected, and 0V when motion is detected). Alarming is done by turn on a relay which in turn close a circuit for a buzzer. In other words, +5V output for a relay will turn on the buzzer. Assume that, as illustrated, a PIR motion detector is connected to PORTB<1> and a relay to turn on a buzzer is connected to PORTD<2>. Note that the PIR motion detector has 3 pins for +5V and Ground connections and for the output signal.

There must be two main operations: keep monitoring the PORTB<1> if it goes to zero, and send High output to PORTD<2> in such case, otherwise Low output. The pseudo-code for this operation looks that:

1. Select PORTB<1> as input by setting TRISB<1>.
2. Select PORTD<2> as output by clearing TRISD<2>.
3. Clear PORTD<2> to turn off the relay (and the buzzer)
4. Check PORTB<1>
5. If PORTB<1>=1 then go to 3. If PORTB<1>=0 go to 6
6. Set PORTD<2> to turn on the relay (and buzzer)
7. Go to 4

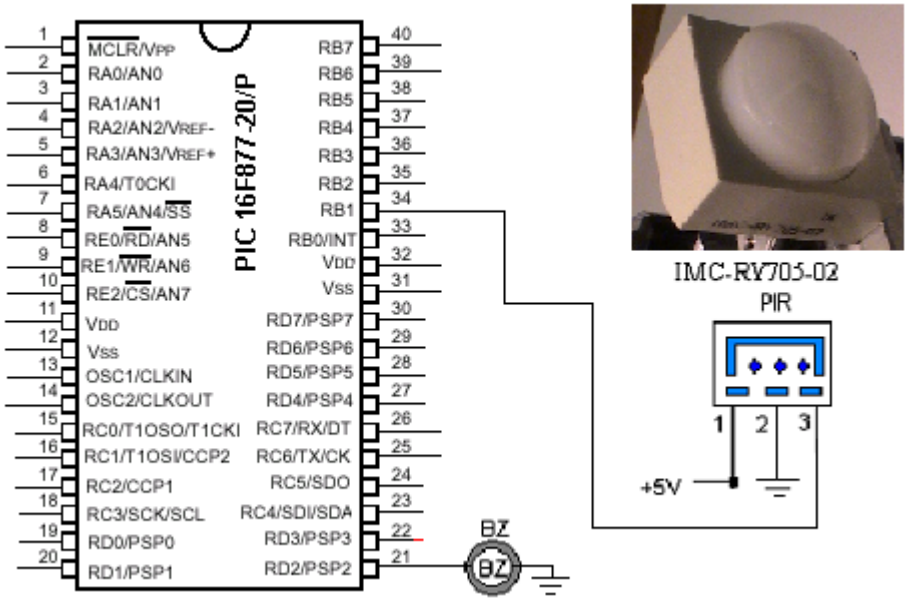


Fig. 10 Illustration of a motion detector and a buzzer (with relay)

Here our interest is how we check PORTB<1> and make decision based on the value of the bit. From the bit-oriented instruction, `btfss f,b` and `btfsc f,b` are very helpful in bit test and decision making, where `b` is for a bit number.

Instruction `btfss f, b` tests (after reading) the bit `b` of file register `f`, i.e., f<b>, and if f<b>=1 ('set"), then skip the line of code following the instruction. If f<b>=0 ("clear"), then the next line below the instruction is executed. Instruction `btfsc f,b` is the opposite instruction of `btfss f, b`.

So let's have another sample code utilized with `btfss f,b` or `btfsc f,b`. In the code below, especially the top portion, we can see that `EQU` can be used to declare the addresses of file registers, as we declare the address of PORTB is 0x06 and that of TRISB is 0x86 as they are mapped in the filer register map shown in page 7.

```
PORTB       EQU   0x06
TRISB       EQU   0x86
```

Also, we use `EQU` to equate a variable (in this case, the variable does not occupy a physical space as used in the `CBLOCK` and `CEND`) to a numerical value. In other words, in the code, `P1`, `P0`, `PIR`, and `BUZ` are file registers nor a physically occupying variable: they just replace numbers 6, 5, 1, and 2, respectively, in the code. Once this `EQU` is done, `PIR` is the same as number 1, for example. Therefore, the instruction

```
bsf   STATUS, P0
```

is the same as

```
bsf   STATUS, 0x05.
```

The sample code follows.

```
;sample code for motion detection and alarm
;
            list P = 16F877         ;Target processor = 16F877
STATUS      EQU   0x03  ;SFR declaration
PORTB       EQU   0x06
TRISB       EQU   0x86
PORTD       EQU   0x08
TRISD       EQU   0x86
P1          EQU   0x06  ;STATUS<6>
P0          EQU   0x05  ;STATUS<5> for bank selection
PIR         EQU   0x01  ;motion detector is connected to bit 1 of PORTB
BUZ         EQU   0x02  ;buzzer is (via relay) connected to bit 2 of PORTD


            bsf   STATUS, P0        ;bank 1

            bsf   TRISB, PIR        ;PORTB<1> as input
                                    ;all other bits are don't care
                                    ;and use initial value
            bcf   TRISD, BUZ        ;PORTD<2> as output

            bcf   STATUS, P0        ;bank 0

            bcf   PORTD, BUZ        ;Alarm off

                                    ;Monitoring PIR
                                    ;Motion detection is indicated
                                    ;by PORTB<1>=0
AGAIN       btfsc PORTB, PIR        ;Is PORTB<1>=0?
            goto  AGAIN             ; No, then keep monitoring
            bsf   PORTD, BUZ        ;Yes, then buzz
            goto  AGAIN             ;go and monitor again
            end
```

Loops and Repetition

In many cases, including arithmetic operation, we need a loop or repetition control flow. Since we do not have `do` loop as in FORTRAN or `for` loop as in C, we have to devise a way to do a loop operation. Consider an LED blinking routine. We want to blink and LED (on followed by

off) for 10 times and then turn off the LED completely. Of course, there must be some time delay to keep an LED on for, say, 100 ms. Apparently we need a time delay subroutine. But in this discussion, let's assume that there is a 100 ms time delay subroutine already built. Details of time delay will be discussed in the next sub-section. Now the main question is how to count (or repeat) the On-delay-off-delay sequence and stop this sequence after ten times.

The answer is around the instruction `decfsz f`, which is to "***dec***rease the content of a ***f***ile register by 1, and then check the content to make a decision: ***s***kip the next line in code if the content after the decrement is reduced to ***z***ero, but if the content is not zero, execute the immediate next line after the instruction". So if we put number 10d (or 0Ah) to a file register, then by using the instruction, we can count from 10 to 1 or ten times. Note that this instruction is performed only on a content stored in a file register: you cannot perform this on a content in **W** register. Also, you have to remember that there is no direct load instruction to put a constant number to a file register. This means you have to load a number, say 0Ah, to **W** register and then move the content from **W** to a file register.

So here comes a pseudo-code for blinking an LED 10 times.
1. Load numeral 10 to **W** register
2. Move **W** to a file register (so we have to declare a GPR before the start line of code)
3. Turn on the LED for 100ms
4. Turn off the LED for 100ms
5. `decfsz` the file register
6. If the content of the file register is zero, then stop
7. If the content of the file register is not zero, go to 3.

Let's convert the pseudo-code to an actual PIC assembly code. As explained before, in addition to the three file register address declaration, using `EQU`, three variables (`P1`, `P0`, and `LED`) are replacing the numbers 6, 5, and 1, respectively. Unlike the above three variables which do not occupy a RAM space but only replace numbers, the variable `TEMP` occupies a RAM space at the address 20h.

```
;sample code for a blinking LED 10 times
;
            list P = 16F877         ;Target processor = 16F877
STATUS      EQU   0x03  ;SFR declaration
PORTB       EQU   0x06
TRISB       EQU   0x86
P1          EQU   0x06  ;STATUS<6>
P0          EQU   0x05  ;STATUS<5> for bank selection
LED         EQU   0x01  ;LED is connected pin 1 of PORTB

            CBLOCK  0x20
                  TEMP        ;declaration of a GPR at bank 0
            ENDC

            bcf   STATUS, P1
            bsf   STATUS, P0        ;bank 1
            bcf   TRISB, LED        ;PORTB<1> as output
            bcf   STATUS, P0        ;bank 0
            movlw 0x0A              ;load 10 to W
            movwf TEMP              ;TEMP = 10
```

```
AGAIN       bsf   PORTB, LED        ;Turn on LED
            call  delay100ms        ;Keep on for 100ms
            bcf   PORTB, LED        ;Turn off LED
            call  delay100ms        ;Keep off for 100ms
            decfsz      TEMP        ;Decrease and Test TEMP
            goto  AGAIN             ;if not zero, continue
            end                     ;if zero, end
```

In the code above, we see that `call` is used to call a subroutine. Also, the subroutine name is immediately follow the `call` instruction. Subroutines and subroutine call is discussed next.

Time delay

Now let's deal with the unfinished business of time delay. Getting time delay can be done two ways: one is to use the timer module of the PIC chip (timer 0, timer 1, and timer 2) and the other is to utilize the number of instruction cycles needed for a loop operation.

As we discussed before, most instruction of PIC takes one instruction cycle (T), and one instruction cycle, with 20 MHZ oscillations, has duration of 0.2μs. So by having five nop ("No Operation") instructions, for example, we could get 1 μs time delay:

```
      bsf   PORTB, LED  ;LED on (it is assumed that LED is EQUed above to
                        ; indicate a number)
      nop               ;it takes 0.2us to execute this line
      nop               ;now total 0.4us
      nop               ;0.6us
      nop               ;0.8us
      nop               ;1.0 us  (LED is on for 1us)
      bcf   PORTB, LED  ;LED off
      nop               ;it takes 0.2us to execute this line
      nop               ;now total 0.4us
      nop               ;0.6us
      nop               ;0.8us
      nop               ;1.0 us  (LED is off for 1us)
```

If we want to have 100μs, we would have to have 500 lines of `nop`, and it definitely is not a good way of writing code. Let's apply that `decfsz` again here. From the instruction table, we see that the instruction cycle for `decfsz` is either 1 or 2. According to the PIC manual, it takes 2 cycles when the test condition is satisfied or when PC (program counter) is modified. In other words, when skip happens it adds one more cycle to the usual 1 cycle instruction. Therefore, the following code, when `TEMP`=1, would take 4 cycles (`decfsz`(1 cycle), skip(1 cycle) since the content is not zero, and `goto B`(2 cycles)).

```
A           decfsz      TEMP        ;Decrease and Test TEMP
            goto        A
            goto        B
```

With `TEMP=3`, it would take (we are following the instruction order),

```
                        ;TEMP=3
            decfsz      ;1 cycle and  TEMP=2
            goto A      ;2 cycles
            decfsz      ;1 cycles and  TEMP=1
            goto A      ;2 cycles
```

```
            decfsz      ;1 cycle and TEMP=0
                        ; 1 cycle for skipping the next line (since TEMP=0)
            goto B      ;2 cycles
```

therefore, total instruction cycles of 10.

As you see above, the pair `decfsz` and `goto` makes 3 instruction cycles. This pair can be used as a basic time delay routine block. Now we can expand this to make the 100μs delay subroutine. First, we know that 100μs delay needs 500 instruction cycles: 100[μs]=0.2[μs/cycle]*500[cycle].

Let's consider the following block of instructions. The number of instruction cycles is included in the comment spaces. From here, we have to determine what value the file register `Kount100us` should contain to achieve 500 instruction cycles. In other words, for the first line of the code below, we have to decide what number we have to load to **W** register.

```
            movwf       Kount100us  ;(1)
R100us      decfsz      Kount100us  ;(1)
            goto        R100us      ;(2) if condition is not met
                                    ;(1) skip if condition is met
            return                  ;(2) end of subroutine
```

The number of instruction cycles can be formulated as below:
T_{total} = 1(`movwf`)+3(`decfsz` & `goto`)* (Content of `Kount100us` register) + 1 (skip) +2 (`return`) = 500.

Or we can rearrange the above formula into:
500 = 3* [`Kount100us`] + 4

If we add one more line at the very top of the code for loading a literal to **W**, `movlw`, then the final formula for the register content to make 100 μs delay would be:
500 = 3* [`Kount100us`] + 5

From the formula, we find the value for `Kount100us` as 165d or 0xA5. Now we are ready to write a 100us delay subroutine. Note that the subroutine name is the starting label of the subroutine.

```
;Subroutine Delay100us ==============
;Need 500 instruction cycles since 1 instruction cycles takes 0.2 us
;The formula for 500 instruction cycles:
;  500 =3*165 + 5
;  this number (165) must be stored into a file register

delay100us  movlw       0xA5        ;165 in decimal
            movwf       Kount100us
R100us      decfsz      Kount100us
            goto        R100us
            return

;end of subroutine =================
```

Now we can make longer delays from the 100μs delay subroutine. For 1ms delay, all we have to do is to call the 100μs delay subroutine 10 times. So a 1ms delay subroutine would look like this:

```
;Subroutine Delay1ms ==============
;Need to call delay100us for 10 times
;
delay1ms    movlw       0x0A        ;10 in decimal
            movwf       Kount1ms
R1ms        call        delay100us
            decfsz      Kount1ms
            goto        R1ms
            return

;end of subroutine =================
```

Actually, the above subroutine takes more than 1ms time delay. Do you see why? Let's count the exact number of instruction cycles.

T = 1 (`movlw`) + 1 (`movwf`) + 10*{500(`call delay100us`) +1 (`decfsz`)+2(`goto`)} + 1(`decfsz`)+1(skip)+2(`return`) = 10*{503}+6=5036 [cycles] --->1.007 [ms].

In most situations, this small error could easily be accepted.

If you can accept this minor errors, you can easily build a subroutine for 100ms delay by calling 100 times the 1ms delay subroutine. Other time delay can be further developed using these time delay subroutines. The curious can measure the actual time delay by measuring the pulse duration of the LED port (without LED of course) by digital storage oscilloscope.