

Programming in the PIC16 Family *

Charles B. Cameron

15 September 2009

Contents

1	Introduction	3
2	Assembly Language Source Programs	4
3	C-Language Control Structures	7
3.1	Assignment	7
3.2	Conditional Statements	9
3.3	Equality ($a = b$) and Inequality ($a \neq b$)	13
3.4	Strictly Less Than ($a < b$)	13
3.5	Less Than or Equal To ($a \leq b$)	13
3.6	Strictly Greater Than ($a > b$)	15
3.7	Greater Than or Equal To ($a \geq b$)	15
3.8	If . . . else	16
3.9	Do . . . while	18
3.10	While	19
3.11	For	20
3.12	Table Look-up	21
3.13	Switch Statement	22

*Course notes for EE361 Microprocessor-based Digital Design

List of Figures

1	Generating a machine program	3
---	--	---

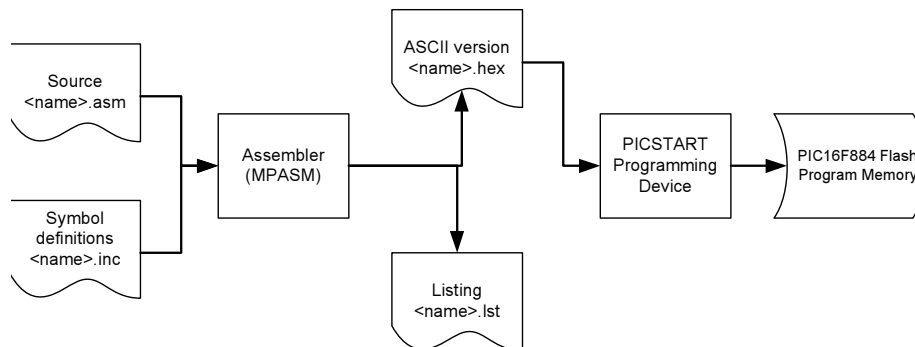


Figure 1: Generating a machine program. A source program with extension `.asm`, written in the PIC16F884 assembly language, is examined by the assembler, which also examines any included files, with extension `.inc`, specified by the source program. The result is a file of ASCII characters that represent the hexadecimal values of the machine program. The PICSTART Plus programmer converts them into the corresponding pattern of zeroes and ones and stores them serially in the flash memory of a processor. The processor can then be inserted into a suitable circuit where it executes the stored programming, gathering inputs and generating outputs as specified by the program.

1 Introduction

Figure 1 illustrates the steps required in creating a program for execution by a PIC16F884 microcontroller when the program is written in PIC16 assembly language. The assembly language program is written as a text file with ASCII characters and saved with the file extension `.asm`. The file may specify the inclusion of other files having extension `.inc`. These are most commonly used to hold definitions of symbols used by the programmer in writing the program. An example of such a `.inc` file is the file `P16F884.inc`, normally found in the `C:/Program Files/Microchip/MPASM Suite` directory.

The program and the included files can be created using any text editor. However, Microchip provides a text editor with its free development environment, MPLAB, that is well-suited to the task. This editor recognizes the key words that correspond to the 35 instructions of the PIC16 family of processors and highlights them in a special color. In addition, the MPLAB program makes it easy to perform the next step shown in Figure 1: the assembly of the program by the MPASM assembler. The assembler scans the assembly language program and its included files and converts them into equivalent machine code. The output of the assembler is another text file with extension `.hex`. However, its contents consist mostly of the 16 hexadecimal characters `0...9A...F`.

The MPLAB program also permits this file to be sent to a device called the PICSTART Plus programmer. It converts the `.hex` file into a serial string of ones and zeros and loads them into the flash memory of the processor itself.

Once the processor's memory contains the program, it can be placed within a suitable circuit. When power is applied, the stored program starts to execute, processing its inputs and generating outputs as specified by the stored program itself.

In this document we focus on the assembly language source program, the `.asm` file.

2 Assembly Language Source Programs

The lines of an assembly language source program are of essentially two kinds:

- Instructions that get translated by the assembler into machine-language equivalents for later execution by the PIC16F884 microcontroller and
- Directives that tailor the operation of the assembler without generating any machine-language equivalents.

The *PIC16F88X Data Sheet*¹ describes the machine-language instructions and their assembly-language equivalents in detail. However, it is mostly silent about assembler directives. Since both are important, we will discuss them both here but we will be careful to distinguish between them.

Anything appearing on a program line to the right of a semicolon (;) is disregarded by the assembler because it is a comment. It is there to make it easier for people to read the program. Use them to explain *why* you are doing something, not just as a restatement of what the adjacent instruction does.

Much use is made in assembly-language programs of labels. A label can be recognized by the assembler because it is placed in the left margin of the program source file. If it is not in the left margin then it must be something else, thinks the assembler. A label is just a symbolic name for a number. The number might refer to a program memory address. It might be a piece of data. It might even be an ASCII character, which of course is really a seven-bit number. What it is in fact is defined by the manner in which it is used and by what is in the programmer's mind when he thinks of it. The assembler only thinks of it as a name for a number.

If a symbol is not a label, then it appears anywhere but at the left margin. If it is one of the 35 instructions described in the *PIC16F88X Data Sheet* then the assembler will recognize it and try to generate a machine instruction from the line where the instruction appears. If it is not one of the 35 instructions, then the assembler decides whether it is one of the numerous directives it knows about. If it is neither of these, then the assembler gives up and issues an error message.

If the symbol is one of the 35 valid instructions then it is one of three kinds, depending on how many operands it needs: one, two, or three. The number of operands is specified in the *PIC16F88X Data Sheet* for each of the

¹ References to the *PIC16F88X Data Sheet* are actually references to the *PIC16F882/883/884/886/887 Data Sheet*, which is its full title.

35 instructions. For example, the `andwf` instruction requires two operands: `f` and `d`. Some instructions require no operands at all. One kind of instruction requires just one operand, `f`. Another kind requires just `k`. And another kind requires two operands: `f` and `b`.

To understand these requires knowing what these operands really are. Table 15-2 on page 230 of the *PIC16F88X Data Sheet* has a column showing the 14-bit pattern generated by the assembler for each of the 35 different kinds of instruction. The number of times the letters *d*, *f*, *k*, and *b* appear in the 14-bit pattern tells you how many bits the assembler requires from you before it can fill in the required bits. For example, the `BTFSS` instruction needs three bits for the `b` field and seven bits for the `f` field. To do this, the assembler reads the operands and tries to squeeze them into the available space.

Of course, if you have numeric operands, your code will be all but unreadable. You will use symbolic names in almost all cases. So the assembler will look up the value of the symbol you have used as an operand and squeeze *that* into the available space.

Here is a program fragment that illustrates this process.

X	<code>equ</code>	H'4A'	; X = H'4A' = D'58'
Y	<code>equ</code>	D'301'	; Y = D'301' = H'12D'
Z	<code>equ</code>	H'22'	; Location 22 is scratch space
STATUS	<code>equ</code>	H'03'	; STATUS register has address 3.
W	<code>equ</code>	0	; W register specifier
F	<code>equ</code>	1	; F register specifier
RP1	<code>equ</code>	6	; RP1 bit in the STATUS register
RP0	<code>equ</code>	5	; RP0 bit in the STATUS register
	<code>...</code>		
	<code>bcf</code>	STATUS,RP1	; Set bank 1
	<code>bsf</code>	STATUS,RP0	
	<code>movlw</code>	X	; Load X into W
	<code>addlw</code>	Y	; Add Y to it
	<code>movwf</code>	Z	; Put the result in Z

The five `equ` directives tell the assembler to associate symbols with numeric values. For example, $X = 58_{10} = 4A_{16}$.

The first `bcf` instruction needs a seven-bit `f`-field and a three-bit `b`-field. The assembler uses the seven bits 000 0011 for the `f`-field because these are the least significant seven bits of 03_{16} . It uses the three bits 110 for the `b`-field because these are the least significant three bits of $6_{16} = 0110_2$.

Similarly, the second `bcf` instruction uses 000 0011 for the `f`-field and 101 for the `b`-field.

Taken together, these two instructions cause the RP1 and RP0 bits of the STATUS register to be set to the value 01_2 . When direct addressing is used, this means that bank 1 is in use. All data addresses require nine bits. The remaining seven bits will come from the instruction that is using direct addressing.

The `movlw` instruction needs eight bits for the `k`-field. It sees that the symbol `X` has the value $4A_{16} = 0100\ 1010_2$ and so these are the eight bits it uses. Using an eight-bit constant that has been stored in an instruction is known as immediate addressing: the data are immediately available, right in the instruction.

The `addlw` instruction contains a surprise. The assembler sees that the symbol `Y` has the value $301_{10} = 12D_{16} = 0001\ 0010\ 1101_2$ but extracts only the least significant eight bits: $0010\ 1101_2 = 2D_{16} = 45_{10}$. When the processor executes the instruction, it knows nothing of the discarded bits. So the arithmetic operation performed is not $58 + 301 = 359$. Rather, it is $58 + 45 = 103$. The problem is that the programmer tried to cram a nine-bit value into an eight-bit field and this is impossible.

The final instruction is `movwf`. It requires a seven-bit `f`-field. Symbol `Z` has the value 22_{16} . Reference to Figure 2-4 in the *PIC16F88X Data Sheet* shows that this is one of the general purpose registers in bank 0. You may use these registers for any purpose. Here, the location is being set aside to hold the value of a variable `Z`. Note that the symbols `X` and `Y` are *values* associated with variables `X` and `Y` whereas the symbol `Z` is an *address* associated with memory where the value of variable `Z` will be stored. Because the values of `X` and `Y` are fixed in program memory, they are not changeable except when the program is written, so they are not really variables at all in the usual sense. (The only way to distinguish between variables and constants is by context. Byte-oriented instructions get their data from data memory using an address: they use variables. Literal instructions get their data from the program word itself: they use constants.)

Some of the instructions cause the values of the `Z`, `DC`, and `C` bits in the `STATUS` register to be determined. Table 15-2 in the *PIC16F88X Data Sheet* shows these instructions. If a status bit is not mentioned, then its value is not altered by the instruction. This means that a program might not have to examine a status bit immediately after it has been determined, as long as no intervening instruction alters it.

The arithmetic and logic unit (ALU) computes four things: an eight-bit output and the three status bits. The only decisions that can be based on the results of an ALU operation are those based on the three status bits. For example, if the eight-bit output of the ALU is 00000000_2 , then the `Z` bit will be set to the value 1; otherwise it will be reset to the value 0. What is more, this determination will only occur in the case of instructions that determine `Z`, such as `ADDWF` and `ANDWF`. The `DC` and `C` bits are only determined by the two addition instructions and the two subtraction instructions.

Any decision that the program needs to make based on the output of the ALU has to be based somehow on one or more of the three status bits. So a program cannot decide to do something if, say, the ALU outputs the value 73. But it can subtract 73 from the output of the ALU, test the answer to see if it is 0, and do different things depending on whether it is 0 or not.

3 C-Language Control Structures

After having struggled through a course to learn the C or C++ language, students have been exposed to a variety of techniques to put structure into their programs. They have learned how to use `for`, `do`, `while`, and `switch` statements, for example, and they know how to use simple data structures like one-dimensional arrays (vectors).

Upon encountering assembly language programming for the first time, they often see it as an entirely new discipline and set aside all the structured programming techniques they knew they had to use with higher level languages.

We now consider how to relate the way one might program a task in C or C++ and the equivalent code in the PIC16 family of microprocessors.

In each section, we present a fragment of code from the C programming language (a subset of C++) and an equivalent fragment of PIC16 assembly language code.

3.1 Assignment

In the C programming language, an example of an assignment is

```
x = 35;
```

For numbers that are not too large, this works very well in C. In many modern machines, C treats signed and unsigned integers as 32-bit numbers. In the PIC16 family, registers only have eight bits, not 32. Therefore it is much more common to run afoul of the processor's limitations with PIC16 assembly language programming than it is in C.

This particular assignment can be translated into PIC16 assembly language very easily:

<code>xinit</code>	<code>equ</code>	<code>D'35'</code>	<code>; Initial value for x</code>
<code>x</code>	<code>equ</code>	<code>H'22'</code>	<code>; A memory location for x</code>
<code>...</code>			
	<code>movlw</code>	<code>xinit</code>	<code>; Retrieve x's initial value</code>
	<code>movwf</code>	<code>x</code>	<code>; Store the value in x's location</code>

(Note that a decimal value such as 27 can be specified as a string like this: `D'27'`). A hexadecimal number such as $2A_{16}$ can be specified with a similar string, `H'2A'`. A binary string such as 0110 1101 can be specified as `B'01101101'`).

But what if you want to store a number that will not fit into an eight-bit word? The usual approach is to allocate storage for as many words as needed. For example, two eight-bit words would provide an aggregate of 16 bits, enough for unsigned integers in the range from 0 to $2^{16} - 1 = 65535$ or for signed integers in the range from -32768 to 32767 . Of course, the PIC16 does not know how to do arithmetic on 16-bit numbers. It only knows how to do addition and subtraction with eight-bit quantities. A sequence of additions can add this capability to the PIC16.

Suppose, for example, that we want to add the 16-bit signed number $x = 11\,000$ to the 16-bit signed number $y = -12\,000$ to get $z = x + y$. Converting these numbers to hexadecimal makes it clear that $x = 2AF8_{16}$ and $y = D120_{16}$ when they are both expressed in the two's-complement system. We could allocate eight-bit storage for $x1$ and $x0$, storing $2A_{16}$ in the location for $x1$ and $F8_{16}$ in the location for $x0$. Similarly, we could store $D1_{16}$ in the location for $y1$ and 20_{16} in the location for $y0$.

To perform the required sum, we would need to do the following, all of which can be done by the PIC16:

1. Set $z1$ to 0.
2. Add $x0$ and $y0$ to form $z0$.
3. If this resulted in a carry, increment $z1$.
4. Add $x1$ to $z1$.
5. If this resulted in a carry, the final result has a carry, too.
6. Add $y1$ to $z1$.
7. If this resulted in a carry, the final result has a carry, too.
8. The result is the final carry, if any, and the two bytes $z1$ and $z0$.

This is a lot for a beginning programmer to do. Here is a program that will do it.

```

x1      equ    H'22'    ; A memory location for x1
x0      equ    H'23'    ; A memory location for x0
y1      equ    H'24'    ; A memory location for y1
y0      equ    H'25'    ; A memory location for y0
z1      equ    H'27'    ; A memory location for z1
z0      equ    H'28'    ; A memory location for z0
temp    equ    H'26'    ; Temporary storage
STATUS  equ    3
C       equ    0
W       equ    0
F       equ    1

      clrf    z1        ; Set z1 to 0
      clrf    temp      ; Set temp to 0
      movf    x0,W      ; Get x0
      addwf   y0,W      ; Add y0
      movwf   z0        ; Store sum
      btfsc  STATUS,C   ; Was there a carry?
      incf   z1,F       ; Yes, so increment z1
      movf   x1,W       ; Get x1

```

```

addwf z1,F      ; Add it into z1
btfsc STATUS,C ; Was there a carry?
incf  temp,F   ; Yes, record the fact
movf  y1,W     ; Get y1
addwf z1,F      ; Add it into z1
btfsc STATUS,C ; Was there a carry?
incf  temp,F   ; Yes, record the fact
bcf   STATUS,C ; Assume no carry
btfsc temp,0   ; Bit 0 of temp is 1
                ; if there was a carry
bsf   STATUS,C ; so set the carry bit

```

For the most part, this is a straightforward implementation of the algorithm. The extra location `temp` is used to hold the carry bit temporarily. This works because there can only be one carry out of the sum entailing the most significant byte, not two, so `temp` can only be either a 0 or a 1.

3.2 Conditional Statements

It is easy to evaluate certain arithmetic conditionals in the PIC16 architecture. This section looks at arithmetic comparisons of the eight-bit values which can be stored in PIC16 registers.

We will look at comparisons of two kinds of eight-bit numbers: unsigned and two's-complement. The PIC16 calculates three status bits whenever it performs addition or subtraction. These are the `C`, `DC`, and `Z` bits, located in bits 0, 1, and 2 of the `STATUS` register, respectively. Only the `C` and `Z` bits are of interest for our present purpose.

Some processors calculate additional status bits when arithmetic instructions are performed. These are the `V` bit, indicating arithmetic overflow, and the `N` bit, indicating the sign of a two's-complement number. Neither of these bits is of any interest when we consider eight-bit numbers to be unsigned, but they are essential when attempting to compare two's-complement numbers.

The `C` bit is the result of a carry out of bit 7 when addition or subtraction is carried out with the PIC16.

The `Z` bit is true (1) when all eight bits of the result of an addition or subtraction operation are zeroes.

When the sum $d = a + b$ is calculated, the `V` bit can be calculated as

$$V = d_7 \cdot \overline{a_7} \cdot \overline{b_7} + \overline{d_7} \cdot a_7 \cdot b_7.$$

When the difference $d = a - b$ is calculated, the `V` bit can be calculated as

$$V = d_7 \cdot \overline{a_7} \cdot b_7 + \overline{d_7} \cdot a_7 \cdot \overline{b_7}.$$

After either a sum $d = a + b$ or a difference $d = a - b$ is calculated, the `N` bit is given by

$$N = d_7.$$

The calculation of V following a subtraction is required in order to evaluate all of the two's-complement comparisons:

$$a < b$$

$$a \leq b$$

$$a = b$$

$$a \geq b$$

and

$$a > b.$$

To evaluate any of these conditions in the PIC16, we perform the subtraction $d = a - b$ first, and then inspect the resulting status bits. Since the PIC16 does not calculate V or N for us, we must manufacture them whenever we want to perform two's-complement comparisons. We do this with software, which will be shown presently. First, though, the table below provides logic equations for each of the five comparisons and for the two cases, unsigned and two's-complement. It is assumed that the four status bits, Z , C , V , and N , already are available. Of course, the PIC processor calculates Z and C for us, placing those bits in the STATUS register.

Comparison	Unsigned	Two's-complement
$a < b$	\bar{C}	$V \oplus N$
$a \leq b$	$\bar{C} + Z$	$Z + (V \oplus N)$
$a = b$	Z	Z
$a \geq b$	C	$\overline{V \oplus N}$
$a > b$	$C \cdot \bar{Z}$	$\bar{Z} \cdot (V \oplus N)$

For the two's-complement case, assume that we have allocated a byte `dd` to hold the difference $d = a - b$ and a byte `STATUS2` to hold

- the V bit in bit-position 0,
- the N bit in bit-position 1,
- the GT bit in bit-position 2 (true when $a > b$, that is, when $\bar{Z} \cdot \overline{(V \oplus N)}$ is true, and
- the LT bit in bit-position 3 (true when $a < b$, that is, when $V \oplus N$ is true).

It should be clear that $a \geq b$ is equivalent to $a \not< b \equiv \overline{LT}$. Similarly, $a \leq b$ is equivalent to $a \not> b \equiv \overline{GT}$.

STATUS2	equ	0x20	; Location of extra status bits
V	equ	0	; Location of V bit in STATUS2
N	equ	1	; Location of N bit in STATUS2
GT	equ	2	; Location of GT bit in STATUS2
LT	equ	3	; Location of LT bit in STATUS2

With these definitions available, and assuming locations for a , b , and d (a , b , and dd , respectively), we can compute the extra status bits after first computing $d = a - b$.

```

; Calculate  $d = a - b$ 
movf    b,W
subwf   a,W
movwf   dd

; Calculate N
bcf     STATUS2,N    ; Assume N is false.
btfs    dd,7         ; Check the assumption
bsf     STATUS2,N    ; It was false, so set N

; Calculate  $V = d_7 \cdot \bar{a}_7 \cdot b_7 + \bar{d}_7 \cdot a_7 \cdot \bar{b}_7$ 
bcf     STATUS2,V    ; Assume V is false.
btfss   dd,7         ; Negative result ( $d_7 = 1$ )?
goto    CalculateVWhenResultIsPositive ; No!

; Yes: Check for  $d_7 \cdot \bar{a}_7 \cdot b_7$ 
CalculateVWhenResultIsNegative
btfs    a,7         ;  $a_7 = 0$ ?
goto    DoneWithV   ; No
btfs    b,7         ; Yes.  $b_7 = 1$ ?
bsf     STATUS2,V   ; Yes, so V is true.
goto    DoneWithV   ; Finished calculating V

; Check for  $\bar{d}_7 \cdot a_7 \cdot \bar{b}_7$ 
CalculateVWhenResultIsPositive
btfss   a,7         ;  $a_7 = 1$ ?
goto    DoneWithV   ; No, finished calculating V
btfss   b,7         ; Yes.  $b_7 = 0$ ?
bsf     STATUS2,V   ; Yes, so V is true.
DoneWithV

; Calculate  $LT = V \oplus N$ 
bcf     STATUS2,LT   ; Assume  $\bar{LT}$ 
btfs    STATUS2,V    ; Check V
goto    CheckWithVTrue
CheckWithVFalse ; Calculate  $\bar{V} \cdot N$ 

```

```

btfs    STATUS2,N
bsf     STATUS2,LT ; a < b is true
goto    DoneWithLT
CheckWithVTrue ; Calculate  $V \cdot \bar{N}$ 
btfs    STATUS2,N
bsf     STATUS2,LT ; a < b is true
DoneWithLT

; Calculate  $GT = \bar{Z} \cdot \bar{LT}$ 
bcf     STATUS2,GT ; Assume  $\overline{GT}$ 
btfs    STATUS2,LT
goto    DoneWithGT
btfs    STATUS,Z
bsf     STATUS2,GT
DoneWithGT

```

3.3 Equality ($a = b$) and Inequality ($a \neq b$)

Subtracting $a - b$ will set the zero flag Z if $a = b$ and will reset it otherwise.

```
    movf    b,W
    subwf   a,W           ; Compute a-b
    btfs    STATUS,Z     ; Do next if a = b
    goto    HandleTheCaseForEquality
HandleTheCaseForInequality
    ...     ; Do whatever is needed if a ≠ b.
    goto    AllDone
HandleTheCaseForEquality
    ...     ; Do whatever is needed if a = b.
AllDone
```

This program fragment works for both the unsigned and the two's-complement cases.

3.4 Strictly Less Than ($a < b$)

After subtracting $a - b$, and calculating the extra status bits N and V as shown above, we check for \bar{C} in the unsigned case and $V \oplus N$ in the two's-complement case.

For the unsigned case:

```
    movf    b,W
    subwf   a,W
    btfs    STATUS,C     ; Do next if a < b
```

For the two's-complement case, assume that the LT bit has already been computed, as shown above.

```
    btfs    STATUS2,LT   ; a < b?
    goto    HandleTheCaseForALessThanB ; Yes.
HandleTheCaseForNotALessThanB           ; No
    ...     ; Do whatever is needed if a ≮ b.
    goto    AllDone
HandleTheCaseForALessThanB
    ...     ; Do whatever is needed if a < b.
AllDone
```

3.5 Less Than or Equal To ($a \leq b$)

In the unsigned case, subtracting $b - a$ will clear the carry flag C if $a > b$ and will set it otherwise. So C will be set if $a \not> b$, which is equivalent to $a \leq b$.

```
    movf    a,W
    subwf   b,W
```

```
btfs    STATUS,C          ; Do next if a ≤ b
```

For the two's-complement case, assume that the **GT** bit has already been computed, as shown above.

```
btfs    STATUS2,GT       ; a > b?  
goto    HandleTheCaseForALessThanOrEqualToB ; No.  
HandleTheCaseForAGreaterThanB                ; Yes  
...     ; Do whatever is needed if a > b.  
goto    AllDone  
HandleTheCaseForALessThanOrEqualToB  
...     ; Do whatever is needed if a ≤ b.  
AllDone
```

3.6 Strictly Greater Than ($a > b$)

Subtracting $b - a$ will clear the carry flag C if $a > b$ and will set it otherwise.

```
movf    a,W
subwf   b,W
btfss   STATUS,C           ; Do next if a > b
```

For the two's-complement case, assume that the GT bit has already been computed, as shown above.

```
btfsc   STATUS2,GT         ; a > b?
goto    HandleTheCaseForAGreaterThanB ; Yes.
HandleTheCaseForALessThanOrEqualToThanB ; No
...     ; Do whatever is needed if a ≤ b.
goto    AllDone
HandleTheCaseForAGreaterThanB
...     ; Do whatever is needed if a > b.
AllDone
```

3.7 Greater Than or Equal To ($a \geq b$)

Subtracting $a - b$ will clear the carry flag C if $b > a$ and will set it otherwise. So C will be set if $b \not> a$, which is equivalent to $b \leq a$.

```
movf    b,W
subwf   a,W
btfsc   STATUS,C           ; Do next if a ≥ b
```

For the two's-complement case, assume that the LT bit has already been computed, as shown above.

```
btfss   STATUS2,LT         ; a < b?
goto    HandleTheCaseForAGreaterThanOrEqualToB ;
      Yes.
HandleTheCaseForALessThanThanB ;
      No
...     ; Do whatever is needed if a ≱ b.
goto    AllDone
HandleTheCaseForAGreaterThanOrEqualToB
...     ; Do whatever is needed if a ≥ b.
AllDone
```

3.8 If ... else ...

C language construct:

```
    if (x == c) {
        Do block 1;
    } else {
        Do block 2;
    }
```

PIC16 Family equivalent:

Here is an example where the condition requires comparing the contents of a variable x stored in location x to the contents of a variable c stored in location cc .

```
    movf    x,W           ; Retrieve x
    sublw   cc            ; Subtract: c-x
    btfss   STATUS,Z      ; If c=x, do Block 1
    goto    Block2        ; Otherwise, do Block 2
Block1:
    ; Do things pertaining to Block 1's task
    ...
    goto    Next          ; Bypass Block 2
Block2:
    ; Do things pertaining to Block 2's task
    ...
Next:
    ; Block 1 and Block 2 are behind us. Do tasks
    ; which should follow whichever one of them
    ; was performed.
```

More elaborate conditions amount to logical combinations of simple conditions. For example, suppose that logic variables u , v , and w are available and that we want to evaluate the condition $u \cdot \overline{v+w}$. This requires evaluating a series of individual conditions in sequence and deciding that the condition is false as soon as it is discovered that this is the case. Evaluating $\overline{v+w}$ could be done as follows, if we assume that the variables are true when their values are all ones and false if all of their bits are zeroes:

```
    mov     v,W           ; Compute v+w (logical or).
    iorwf   w,W
    movwf   temp          ; Save the result in temporary
                           ; storage so that it can
                           ; be complemented.
    comf    temp,W        ; Compute  $\overline{v+w}$ .
    btfss   STATUS,Z      ; If  $\overline{v+w}$ , Z is true
    goto    Block2
```

```

movf    temp,W
andwf   u,W      ; Compute  $u \cdot \overline{v+w}$ .
btfsc   STATUS,Z ; If  $u \cdot \overline{v+w}$  is true, Z is false
goto    Block2
Block1
        ; Do things pertaining to Block 1's task
        ...
goto    Next      ; Bypass Block 2
Block2:
        ; Do things pertaining to Block 2's task
        ...
Next:
        ; Block 1 and Block 2 are behind us. Do tasks
        ; which should follow whichever one of them
        ; was performed.

```

3.9 Do ... while

C language construct:

The significant characteristic of a `Do...while` structure is that the condition is tested *before* each execution of the block, which means it might never be executed at all if the condition is not met initially.

In the following example, the condition is to see whether an unsigned number x is less than or equal to an unsigned constant k .

```
do {  
    Block;  
} while (x >= k);
```

PIC16 Family equivalent:

```
Block:  
    ; Do the things pertaining to Block 1's task  
    ...  
    ; See if the block should be repeated  
    movlw    k  
    subwf   x,W           ; Calculate  $x - k$ .  
    btfsc   STATUS,C      ; Skip if  $k > x$   
    goto    Block        ;  $x \geq k$ , so repeat Block  
Next:  
    ; The Block of code has been executed. Carry on.
```

3.10 While ...

C language construct:

The significant characteristic of a `While...` structure is that the condition is tested *after* each execution of the block, which ensures that the block is executed at least once.

In the following example, the condition is to see whether an unsigned number x is greater than or equal to an unsigned constant k .

```
Block; }
```

PIC16 Family equivalent:

```
StartOfBlock :  
    movlw    k  
    subwf    x,W           ; Calculate  $x - k$   
    btfs    STATUS,C      ;  $x \geq k$ , do Block  
    goto    Next          ;  $x < k$ , skip Block  
Block :  
    ; Do the things that belong in this Block of code  
    ...  
    goto    StartOfBlock  
Next :  
    ; The Block of code is complete. Carry on.  
    ...
```

3.11 For ...

C language construct:

The significant characteristic of a `For ...` structure is that an initial condition is established, a condition is evaluated before each iteration, and some instructions are calculated at the end of each iteration, usually to alter data used in evaluating the condition subsequently.

In the following example, the initial condition is to set an unsigned variable *i* the value 0. The conditional code is executed repeatedly as long as *i* is less than the contents of another unsigned variable, *n*. After each iteration, the value of *i*.

```
for (i=0; i<n; ++i)
{
    Block;
}
```

PIC16 Family equivalent:

```
i      equ H'20'    ; A memory location for x1
temp   equ H'21'    ; Temporary storage
n      equ H'22'    ; Temporary storage
STATUS equ 3
C      equ 0
W      equ 0
F      equ 1

    clrf    i        ; i = 0
Test:
    movf   n,W        ; Calculate i - n
    subwf  i,W
    btfsc  STATUS,C   ; i < n, do Block
    goto   Next       ; i >= n, skip Block
Block:
    ...
    incf   i,F
    goto   Test
Next:
```

3.12 Table Look-up

C language construct:

```
x = list[i];
```

where *list* consists of *n* 8-bit characters and *i* is an index in the range $[0, n - 1]$. Care must be taken to ensure that index *i* not exceed $n - 1$. If $i \geq n$, then code beyond that last **retlw** instruction will be executed. Almost always, this is a programming error.

PIC16 Family equivalent:

	movf	i,W	; Put the index <i>i</i> in <i>W</i>
	call	LookUp	; Lookup the <i>ith</i> element
	movwf	x	; Store it in <i>x</i>
		...	
LookUp:			
	addwf	PCL,F	; Add index to <i>PCL</i>
	retlw	L0	; 0th element of list
	retlw	L1	; 1st element of list
	retlw	L2	; 2nd element of list
		...	
	retlw	LN_1	; (<i>n-1</i>)st element of list

3.13 Switch Statement

C language construct:

```
switch(x) {
  case n0:
    /* Block n0, executed if x = n0 */
    break;
  case n1:
    /* Block n1, executed if x = n1 */
    break;
  ...
  case nkminus1:
    /* Block nk-1, executed if x = nk-1
       */
    break;
  default:
    /* Default block, executed if
       x ≠ nj for any j ∈ {0, k-1} */
}
```

PIC16 Family equivalent:

```
movf    n0,W           ; Compute x - n0
subwf   x,W
btfsc   STATUS,Z       ; Skip if x ≠ n0
goto    Blockn0        ; Do Block n0 if x = n0
movf    n1,W           ; Compute x - n1
subwf   x,W
btfsc   STATUS,Z       ; Skip x ≠ n1
goto    Blockn1        ; Do Block n1 if x = n1
...
movf    nkminus1,W    ; Compute x - nk-1
subwf   x,W
btfsc   STATUS,Z       ; Skip if x ≠ nk-1
goto    Blocknkminus1 ; Do Block n - 1 if x = nk-1
Default:
        ; Do whatever is needed if no case is satisfied.
...
goto    Next
Blockn0:
        ; Do the things pertaining to the case where
        x = n0
...
goto    Next
Blockn1:
```

```
        ; Do the things pertaining to the case where
          x = n1
    ...
    goto Next
    ...
Blocknkminus1:
        ; Do the things pertaining to the case where
          x = nk-1
    ...
Next:
        ; The Switch statement is complete. Carry on.
```