

Programming in the PIC16 Family *

Charles B. Cameron

18 January 2007

Contents

1	Introduction	3
2	Assembly Language Source Programs	4
3	C-Language Control Structures	6
3.1	Assignment	7
3.2	Conditional Statements	9
3.3	Equality ($a = b$)	10
3.4	Inequality ($a \neq b$)	10
3.5	Strictly Less Than ($a < b$)	10
	3.5.1 Unsigned Integers	10
	3.5.2 Signed Integers	10
3.6	Less Than or Equal To ($a \leq b$)	12
	3.6.1 Unsigned Integers	12
	3.6.2 Signed Integers	12
3.7	Strictly Greater Than ($a > b$)	14
3.8	Greater Than or Equal To ($a \geq b$)	14
3.9	If . . . else	14
3.10	Do . . . while	15
3.11	While	16
3.12	For	17
3.13	Table Look-up	18
3.14	Switch Statement	19

*Course notes for EE461 Microprocessor-based Digital Design

List of Figures

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

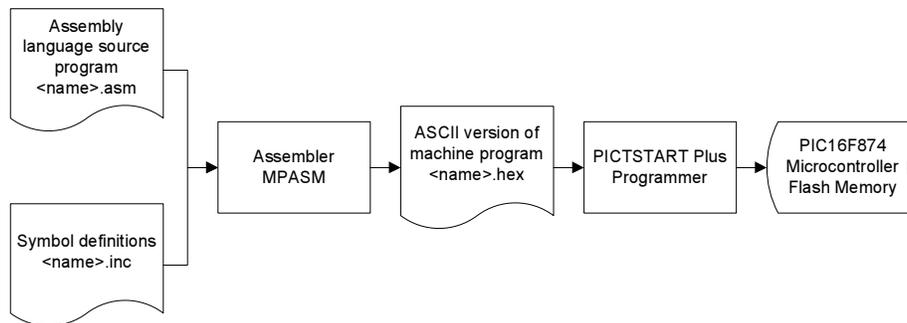


Figure 1: Generating a machine program. A source program with extension `.asm`, written in the PIC16F874 assembly language, is examined by the assembler, which also examines any included files, with extension `.inc`, specified by the source program. The result is a file of ASCII characters that represent the hexadecimal values of the machine program. The PICTSTART 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 PIC16F874 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.

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`. This is another text file. 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 PICTSTART 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 PIC16F874 microcontroller and
- Directives that tailor the operation of the assembler without generating any machine-language equivalents.

The *PIC16F87XA Data Sheet* 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 *PIC16F87XA 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 *PIC16F87XA Data Sheet* for each of the 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 160 of the *PIC16F87XA 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.

1	X	equ	H'4A'	; X = H'4A' = D'58'
2	Y	equ	D'301'	; Y = D'301' = H'12D'
3	Z	equ	H'22'	; Location 22 is scratch space
4	STATUS	equ	H'03'	; STATUS register has address 3.
5	W	equ	0	; W register specifier
6	F	equ	1	
7	RP1	equ	6	
8	RP0	equ	5	
9		...		
10		bcf	STATUS,RP1	; Set bank 1
11		bsf	STATUS,RP0	
12		movlw	X	; Load X
13		addlw	Y	; Add Y to it
14		movwf	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 $301_{10} = 12D_{16} = 000100101101_2$ but extracts only the least significant eight bits: $00101101_2 = 2D_{16} = 45_{10}$. 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 *PIC16F87XA 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.

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 *PIC16F87XA 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 only computes three things: an eight-bit output and the three status bits. The only decisions that can be made are 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. And 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 these three 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

```
1      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:

1	xinit	equ	D'35'	; <i>Initial value for x</i>
2	x	equ	H'22'	; <i>A memory location for x</i>
3	...			
4		movlw	xinit	; <i>Retrieve x's initial value</i>
5		movwf	x	; <i>Store the value in x's location</i>

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 $2^{16} = 65\,536$ or for signed integers in the range from $-32\,768$ to $32\,767$. 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 $y = 5120_{16}$ when they are both expressed in the two's-complement system. We could allocate eight-bit storage for $x1$ and $x0$ and store $2A_{16}$ in the location for $x1$ and $F8_{16}$ in the location for $x0$. Similarly, we could store 51_{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.

1	<code>x1</code>	<code>equ</code>	<code>H'22'</code>	<code>;</code>	<i>A memory location for x1</i>
2	<code>x0</code>	<code>equ</code>	<code>H'23'</code>	<code>;</code>	<i>A memory location for x0</i>
3	<code>y1</code>	<code>equ</code>	<code>H'24'</code>	<code>;</code>	<i>A memory location for y1</i>
4	<code>y0</code>	<code>equ</code>	<code>H'25'</code>	<code>;</code>	<i>A memory location for y0</i>
5	<code>z1</code>	<code>equ</code>	<code>H'27'</code>	<code>;</code>	<i>A memory location for z1</i>
6	<code>z0</code>	<code>equ</code>	<code>H'28'</code>	<code>;</code>	<i>A memory location for z0</i>
7	<code>temp</code>	<code>equ</code>	<code>H'26'</code>	<code>;</code>	<i>Temporary storage</i>
8	<code>...</code>				
9		<code>clrf</code>	<code>z1</code>	<code>;</code>	<i>Set z1 to 0</i>
10		<code>clrf</code>	<code>temp</code>	<code>;</code>	<i>Set temp to 0</i>
11		<code>movf</code>	<code>x0,W</code>	<code>;</code>	<i>Get x0</i>
12		<code>addwf</code>	<code>y0,W</code>	<code>;</code>	<i>Add y0</i>
13		<code>movwf</code>	<code>z0</code>	<code>;</code>	<i>Store sum</i>
14		<code>btfs</code>	<code>STATUS,C</code>	<code>;</code>	<i>Was there a carry?</i>
15		<code>incf</code>	<code>z1,F</code>	<code>;</code>	<i>Yes, so increment z1</i>
16		<code>movf</code>	<code>x1,W</code>	<code>;</code>	<i>Get x1</i>
17		<code>addwf</code>	<code>z1,F</code>	<code>;</code>	<i>Add it into z1</i>
18		<code>btfs</code>	<code>STATUS,C</code>	<code>;</code>	<i>Was there a carry?</i>
19		<code>incf</code>	<code>temp,F</code>	<code>;</code>	<i>Yes, record the fact</i>
20		<code>movf</code>	<code>y1,W</code>	<code>;</code>	<i>Get y1</i>
21		<code>addwf</code>	<code>z1,F</code>	<code>;</code>	<i>Add it into z1</i>
22		<code>btfs</code>	<code>STATUS,C</code>	<code>;</code>	<i>Was there a carry?</i>
23		<code>incf</code>	<code>temp,F</code>	<code>;</code>	<i>Yes, record the fact</i>
24		<code>bcf</code>	<code>STATUS,C</code>	<code>;</code>	<i>Assume no carry</i>
25		<code>btfs</code>	<code>temp,0</code>	<code>;</code>	<i>Bit 0 of temp is 1</i>
26				<code>;</code>	<i>if there was a carry</i>
27		<code>bsf</code>	<code>STATUS,C</code>	<code>;</code>	<i>so set the carry bit</i>

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, but others are more difficult. This section looks at arithmetic comparisons of the eight-bit values which can be stored in PIC16 registers. For equality ($=$) and inequality (\neq), it does not matter whether numbers are regarded as unsigned integers or signed integers. However, for inequalities ($<$, \leq , $>$, and \geq), it matters a great deal.

In each of the conditional statement subsections below, we assume that the variables a , b , and d have been defined as registers in the PIC16 microprocessor's program memory.¹ For example, here is a way to place the decimal value 38 in the memory at address 20 hexadecimal and give that address the symbolic name a : It entails declaring constants and reserved memory locations using the `equ` directive.

```
1 a      equ H'20'   ; Put "a" in location 0x20
2 avalue equ D'38'   ; "a" will be initialized
3                ; to 38 (decimal)
4 ...
5        movlw  avalue
6        movwf  a     ; initializes the storage
7                ; location for "a"
```

¹Not all of the programs use d .

3.3 Equality ($a = b$)

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

```
1      movf    b,W
2      subwf   a,W
3      btfsc   STATUS,Z          ; Do next if a = b
```

3.4 Inequality ($a \neq b$)

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

```
1      movf    a,W
2      subwf   b,W
3      btfss   STATUS,Z          ; Do next if a <> b
```

3.5 Strictly Less Than ($a < b$)

3.5.1 Unsigned Integers

If a and b are unsigned integers, then subtracting $a - b$ will clear the carry flag C and set the sign bit N (bit 7) of the result if $a < b$. If either of these conditions fails, then $a \geq b$. The N bit is true for negative results. Unsigned numbers, of course, do not use this bit for this purpose. However, the hardware for unsigned integers is identical to that for signed integers: we take advantage of this fact and look for “negative” results that do not generate a carry.

```
1  N      equ 7      ; Sign bit is the most significant bit
2  ...
3  movf   b,W
4  subwf  a,W
5  btfsc  STATUS,C   ; Clear carry bit?
6  goto   NotStrictlyLessThan
7  movwf  a_temp     ; Sign bit = 1?
8  btfss  a_temp,N
9  goto   NotStrictlyLessThan
10 StrictlyLessThan ; Both conditions were true
11 ...
12 goto   Done
13 NotStrictlyLessThan ; 1 or 2 conditions were false
14 ...
15 Done
```

3.5.2 Signed Integers

If a and b are signed integers, then we also have to consider the overflow bit, V . Unfortunately, PIC16 processors do not compute it. Overflow occurs (that

is, $V = 1$) if the result of the subtraction has the wrong arithmetic sign, an indication that the difference is too big to fit in the eight bits available for signed integers.

There are two ways this can happen.

1. If $a \geq 0$ and $b < 0$, $a - b$ should be positive: if it's negative, overflow occurred.
2. If $a < 0$ and $b \geq 0$, $a - b$ should be negative: if it's positive, overflow occurred.

A Boolean expression for this can be written if we calculate $d = a - b$. Then

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

With V available, $a < b$ whenever V and N differ, a condition captured by $V \oplus N$ or, equivalently, $\overline{V} \cdot N + V \cdot \overline{N}$. It can be shown by, say, the use of truth tables that this is equivalent to the expression $a_7 \overline{b_7} + a_7 d_7 + \overline{b_7} d_7$. One approach to detecting this is to look for any of the conditions in its complement, $\overline{a_7} \overline{d_7} + b_7 \overline{d_7} + \overline{a_7} b_7$. If any of these terms is true, the test $a < b$ fails.

1	N	equ 7	; Sign bit is the most significant bit
2		...	
3		movf	b,W
4		subwf	a,W
5		movwf	d
6	FirstTest		; a > 0 and d > 0?
7		btfs	a,N
8		goto	SecondTest
9		btfs	d,N
10		goto	NotStrictlyLessThan
11	SecondTest		; b < 0 and d > 0?
12		btfs	b,N
13		goto	ThirdTest
14		btfs	d,N
15		goto	NotStrictlyLessThan
16	ThirdTest		; a > 0 and b < 0?
17		btfs	a,N
18		goto	StrictlyLessThan
19		btfs	b,N
20		goto	NotStrictlyLessThan
21	StrictlyLessThan		
22		...	
23		goto	Done
24	NotStrictlyLessThan		
25		...	
26	Done		

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

3.6.1 Unsigned Integers

This test is almost the same as for $a < b$, except we now should test the Z bit, too, because $a \leq b$ if $z = 1$.

```
1 N      equ 7      ; Sign bit is the most significant bit
2      ...
3      movf    b,W
4      subwf   a,W
5      btfsc   STATUS,Z      ; Zero result (equality)?
6      goto    LessThanOrEqual
7      btfsc   STATUS,C      ; Clear carry bit?
8      goto    NotStrictlyLessThan
9      movwf   a_temp      ; Sign bit = 1?
10     btfss   a_temp,N
11     goto    NotStrictlyLessThan
12 LessThanOrEqual
13     ...
14     goto    Done
15 NotLessThanOrEqual
16     ...
17 Done
```

3.6.2 Signed Integers

As with unsigned integers, this test is almost the same as for $a < b$, except we now should test the Z bit, too, because $a \leq b$ if $z = 1$.

```
1 N      equ 7      ; Sign bit is the most significant bit
2      ...
3      movf    b,W
4      subwf   a,W
5      movwf   d
6 ZerothTest      ; d = 0?
7      btfsc   STATUS,Z
8      goto    LessThanOrEqual
9 FirstTest       ; a > 0 and d > 0?
10     btfsc   a,N
11     goto    SecondTest
12     btfss   d,N
13     goto    NotLessThanOrEqual
14 SecondTest     ; b < 0 and d > 0?
15     btfss   b,N
16     goto    ThirdTest
17     btfss   d,N
```

```
18         goto    NotLessThanOrEqual
19 ThirdTest ; a > 0 and b < 0?
20         btfsc  a,N
21         goto    LessThanOrEqual
22         btfsc  b,N
23         goto    NotLessThanOrEqual
24 LessThanOrEqual
25         ...
26         goto    Done
27 NotLessThanOrEqual
28         ...
29 Done
```

3.7 Strictly Greater Than ($a > b$)

Use the same method as for $a < b$, but interchange the rôles of a and b .

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

Use the same method as for $a \leq b$, but interchange the rôles of a and b .

3.9 If ... else ...

C language construct:

```
1      if (x == c) {
2          Do block 1;
3      } else {
4          Do block 2;
5      }
```

PIC16 Family equivalent:

1	movf	x,W	<i>; Retrieve x</i>
2	sublw	c	<i>; Subtract c-x</i>
3	btfs	STATUS,Z	<i>; If equal, do Block 1</i>
4	goto	Block2	<i>; Otherwise, do Block 2</i>
5	Block1:		
6		...	
7	goto	Next	<i>; Bypass Block 2</i>
8	Block2:		
9		...	
10	Next:		

3.10 Do ... while

C language construct:

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

PIC16 Family equivalent:

```
1 Block :  
2     ...  
3     movlw    k  
4     subwf   x,W           ; Calculate x-k.  
5     btfsc   STATUS,C      ; Skip if k > x  
6     goto    Block        ; x >= k, so repeat Block  
7 Next :
```

3.11 While ...

C language construct:

```
1      while (x >= k){
2          Block;
3      }
```

PIC16 Family equivalent:

```
1 Start:
2     movf  x,W           ; is x >= k?
3     movwf temp         ; save k in temp
4     movlw k            ; Calculate x - k
5     subwf temp,W
6     btfss STATUS,C     ; x >= k, do Block
7     goto  Next         ; x < k, skip Block
8 Block:
9     ...
10    goto  Start
11 Next:
```

3.12 For ...

C language construct:

```
1     for (i=0; i<n; ++i) {
2         Block;
3     }
```

PIC16 Family equivalent:

```
1     clrfsf    i           ; i = 0
2 Test:
3     movf     i,W
4     movwf   temp        ; save n in temp
5     movf     n,W        ; Calculate i - n
6     subwf   temp,W
7     btfsc   STATUS,C    ; i < n, do Block
8     goto    Next       ; i >= n, skip Block
9 Block:
10    ...
11    movlw   1           ; ++i
12    addwf   i,F
13    goto   Test
14 Next:
```

3.13 Table Look-up

C language construct:

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

where *list* is *n* 8-bit characters and *i* is an index in the range $[0, n - 1]$.

PIC16 Family equivalent:

```
1      movf    i,W      ; Put the index i in W
2      call   LookUp   ; Lookup the ith element
3      movwf  x        ; Store it in x
4      ...
5
6 LookUp:
7      addwf  PCL      ; Add index to PCL
8      retlw  L0       ; 0th element of list
9      retlw  L1       ; 1st element of list
10     retlw  L2       ; 2nd element of list
11     ...
12     retlw  LN_1     ; (n-1)st element of list
```

3.14 Switch Statement

C language construct:

```
1      switch(x) {
2          case n0:
3              Block_n0
4              break;
5          case n1:
6              Block_n1
7              break;
8          ...
9          case nk:
10             Block_nk
11             break;
12         default:
13             DefaultBlock
14     }
```

PIC16 Family equivalent:

1	movf	x,W	; Put x in temp
2	movwf	temp	
3	movf	n0,W	; Compute x - n0
4	subwf	temp,W	
5	btfs	STATUS,Z	; x = n0
6	goto	Blockn0	; x <> n0
7			
8	movf	x,W	; Put x in temp
9	movwf	temp	
10	movf	n1,W	; Compute x - n1
11	subwf	temp,W	
12	btfs	STATUS,Z	; x = n1
13	goto	Blockn1	; x <> n1
14			
15		...	
16			
17	movf	x,W	; Put x in temp
18	movwf	temp	
19	movf	nk,W	; Compute x - nk
20	subwf	temp,W	
21	btfs	STATUS,Z	; x = nk
22	goto	Blocknk	; x <> nk
23			
24			
25	Default:		
26		...	

```
27     goto    Next
28
29 Blockn0 :
30     ...
31     goto  Next
32
33 Blockn1 :
34     ...
35     goto  Next
36
37     ...
38
39 Blocknk :
40     ...
41
42
43 Next :
```