CHAPTER 13

Assembler 101

At the end of this chapter, you won't be an expert in assembler language programming. In fact, if this is your first introduction to PIC assembler, you won't even qualify as a novice. What I hope to do, however, is to illuminate a few of the more important assembler programming principles to prepare the way for Chapters 14 and 15 where we will learn assembler routines that do things otherwise impossible with MBasic and speed up certain operations.

The Basics

Let's start with the basics. To keep the discussion to a manageable length, we'll limit our discussion to the 16F876/877/876A/877A series devices. For more details on these devices, as well as other PICs, consult the sources identified in the References section.

What is Assembler?

As we learned in Chapter 2, the MBasic programs you write are compiled into an assembler format, linked with library functions by a linker ultimately converted into machine instruction executable by a PIC.

Assembler is a human compatible form of directly addressing a PIC's built-in instruction set. A PIC's instruction set—and mid-range PICs such as the 16F87x/87xA devices have only 35 instructions—works with data and hardware at the most elementary level. Typical assembler instructions involve moving bytes from one storage location to another, or setting and clearing bits. Even the most complex mid-range PIC instructions do nothing more than add or subtract 8-bit values. Consequently, one line of MBasic code may correspond to dozens, or even hundreds of lines of assembler code.

We may illustrate this difference by comparing MBasic, assembler and machine instructions to accomplish a very simple task: `A = 123`, where `A` is a byte variable.

MBasic	*Assembler*	*Machine Code*
`A=123`	`movlw .123`	`11 00 00 01111011`
	`movwf A`	`00 00 00 11001000`

It's easy to write `A=123`. It's more difficult to write the two assembler instructions and it's far more difficult and time consuming to manually construct the machine code instructions. Fortunately, we'll never have to deal with machine code; MBasic's integrated compiler and assembler will mask that complexity from us.

What do these two cryptic assembler instructions mean?

`movlw .123`—means to load the 8 bits corresponding to the decimal number 123 into the PIC's working register, W. The period symbol tells the assembler that the number is a decimal number, not a hexadecimal (written in the form 0xNN, or a binary number, written in the form b'NNNNNNNN'). In PIC assembler terminology, copying a value from one location to another is called a *move*. A constant is

called a *literal* and the working register is called *w*. Hence, the assembler operation code name (usually called simply an opcode) is **movlw**, standing for move a literal to register w.

movwf A—means to copy the value currently in working register w into the file location identified by the value of the constant **A**. (We'll assume for the moment that **A** represents the storage register (file) at physical address $48.) The opcode name **movwf** is constructed from its actions; to move a value from register w to a file.

Not so fast, you're probably thinking. You just said **A** is a constant with a value of $48; but not one page earlier you said **A** is a variable that we are going to set to have the value 123. Make up your mind, which is it? The most accurate description of **A** is that it is neither a variable nor a constant, but rather it is an *address* at which a byte is stored. That *address* is static and is assigned—by the compiler, or by you if you write only in assembler—and does not change, so in that regard **A** is a constant. However, the *value* stored at *address* **A** can change. Hence, **A** can be considered a variable when we refer to the contents of the memory at address **A**, or a constant when we refer to the memory address itself.

Terminology

When starting out to learn something new, the hardest part is often understanding the terminology. The underlying idea may be simple, but it's described in unfamiliar words. Or, the words may be familiar, but they are used in a fashion that divorces them from their normal meaning. We've already thrown around the words, register, file, literal, move, opcode, and address, to mention but a few.

To understand the terminology, we must understand the PIC's architecture. First, remember what we learned in Chapter 1; the PIC follows the Harvard architecture and has separate *program* memory and *data* memory. Program memory holds—for mid-range PICs—a series of 14-bit machine code statements, similar to those we noted earlier. The data memory is 8-bits wide and holds not only variables of the type we use in MBasic and assembler, but also other variables controlling the PIC's functioning, its hardware and its features.

Data Memory

Let's look first at how the data memory is organized in a 16F876/877/876A/877A device. Figure 13-1 provides a conceptual view of the data memory. The data memory is organized into four *banks*, with each bank having 128 addresses, giving a maximum of 512 unique addresses. Each address represents one of three types of 8-bit wide register files:

- ***General-Purpose Register***—user RAM. The '87x chips we're discussing have 368 general-purpose registers.
- ***Special Function Register***—a file that may be accessed by the user (in almost all cases, both read and write) but it is used internally by the PIC for a function. For example, all ports are special function registers, and thus we may both read from and write to, for example, PortB.
- ***Unimplemented Address***—an address that is dead because it is not physically implemented. The '87x chips we're discussing have 15 dead addresses.

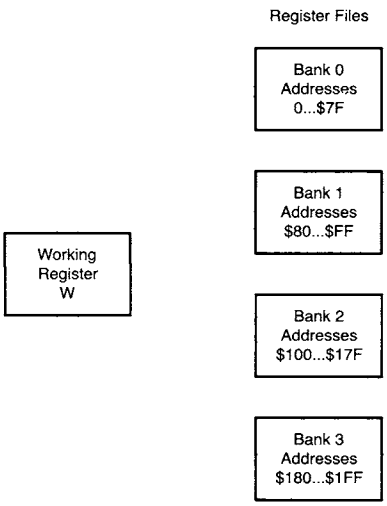


Figure 13-1: Conceptual view of 16F87x data memory.

Microchip calls these memory locations *register files*, or *registers*, or *files*. Don't get confused; whether we call it a register, or a file, it is still the location of an 8-bit information storage unit, identified by a numerical address in the range 0...511. (Banking makes the actual addressing a bit more complex, as we'll see later.)

The *working* or *W* register is special—it's an 8-bit memory location built into the PIC's central processing core and tied to the arithmetic and logic unit. It doesn't have an address in the same sense that the register files do; rather it's identified through opcodes as the source of data or the destination of a particular action.

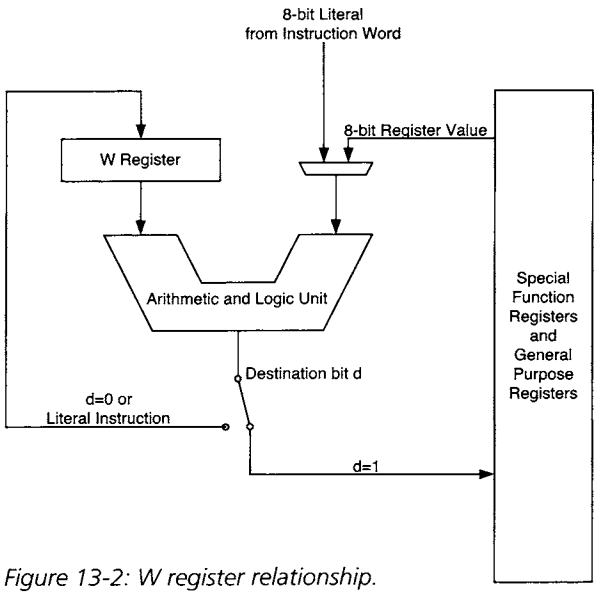


Figure 13-2: W register relationship.

Many logic and arithmetic function require two operands; such as addition, subtraction and logic operations. One operand may be a constant and the other a register value, or both may be register values. The arithmetic and logic unit portion of the PIC performs these functions and, as illustrated in Figure 13-2 , *one of the two input operands must be held in the* **W** *register*. This is a simple, but inviolate rule for mid-range PICs—if the opcode has two operands, one must be in the **W** register. Where the result of the operation is stored is determined by the opcode's destination bit and may be either the **W** register or one of the operand registers. It can't be a third register—it's either the **W** register or one of the operand registers.

Suppose we have two files, **A** and **B** and that we wish to add their values together and store the result in file **C**. In MBasic, this operation is **C = A + B**. (We'll assume that somehow the letters **A**, **B** and **C** are linked to the addresses of three general-purpose registers. You do understand that when we talk about adding **A** to **B** and putting the result into **C**, we are referring to the contents of the memory locations **A**, **B** and **C**, right? If not go back and read the last pages again.) Addition is a two-operand function and, as we learned, two operand functions require one operand to be held in **W**. Consequently, there is no single machine instruction, or opcode, that lets us add **A** and **B** and store the result at **C**. Rather, we must first move the contents of **A** into **W**, then add **B** to the contents of **W**, storing the result in **W** and lastly move **W**'s contents to **C**.

Each Special Function Register has been given a name by Microchip and each bit within each Special Function Register has a name. It's up to us to assign names to General-Purpose Registers we use to hold variables.

The following is a detailed register file map for the 16F876/A/877/A devices.

Bank 0	Hex Adr	Bank 1	Hex Adr	Bank 2	Hex Adr	Bank 3	Hex Adr
Indirect addr.(*)	00	Indirect addr.(*)	80	Indirect addr.(*)	100	Indirect addr.(*)	180
TMR0	01	OPTION_REG	81	TMR0	101	OPTION_REG	181
PCL	02	PCL	82	PCL	102	PCL	182
STATUS	03	STATUS	83	STATUS	103	STATUS	183
FSR	04	FSR	84	FSR	104	FSR	184
PORTA	05	TRISA	85		105		185
PORTB	06	TRISB	86	PORTB	106	TRISB	186
PORTC	07	TRISC	87		107		187
PORTD(1)	08	TRISD(1)	88		108		188

Bank 0	Hex Adr	Bank 1	Hex Adr	Bank 2	Hex Adr	Bank 3	Hex Adr
PORTE(1)	09	TRISE(1)	89		109		189
PCLATH	0A	PCLATH	8A	PCLATH	10A	PCLATH	18A
INTCON	0B	INTCON	8B	INTCON	10B	INTCON	18B
PIR1	0C	PIE1	8C	EEDATA	10C	EECON1	18C
PIR2	0D	PIE2	8D	EEADR	10D	EECON2	18D
TMR1L	0E	PCON	8E	EEDATH	10E		18E
TMR1H	0F		8F	EEADRH	10F		18F
T1CON	10		90		110		190
TMR2	11	SSPCON2	91		111		191
T2CON	12	PR2	92		112		192
SSPBUF	13	SSPADD	93		113		193
SSPCON	14	SSPSTAT	94		114		194
CCPR1L	15		95		115		195
CCPR1H	16		96	General Purpose Register (RAM) 16 Bytes	116	General Purpose Register (RAM) 16 Bytes	196
CCP1CON	17		97		117		197
RCSTA	18	TXSTA	98		118		198
TXREG	19	SPBRG	99		119		199
RCREG	1A		9A		11A		19A
CCPR2L	1B		9B		11B		19B
CCPR2H	1C	CMCON	9C		11C		19C
CCP2CON	1D	CVRCON	9D		11D		19D
ADRESH	1E	ADRESL	9E		11E		19E
ADCON0	1F	ADCON1	9F		11F		19F
General-Purpose Register (RAM) 98 Bytes	20	General Purpose Register (RAM) 80 Bytes	A0	General Purpose Register (RAM) 80 Bytes	120	General Purpose Register (RAM) 80 Bytes	1A0
	7F		EF		16F		1EF
		Accesses 70-7F	F0	Accesses 70-7F	170	Accesses 70-7F	1F0
			FF		17F		1FF

* *Not a physical register*
(1) Not implemented in '876
(2) Reserved

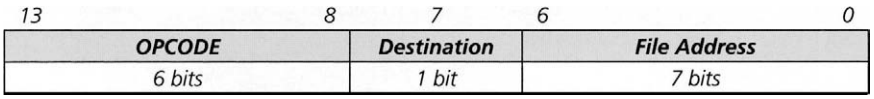
This is an important table; so take some time to carefully look it over. You should see several things:

- Certain files are repeated in multiple banks. This permits us to access those files regardless of the current bank settings.
- Some General-Purpose Registers are marked "accesses." This represents multiple addressed shared memory; the same 16 bytes of memory are addressed at $70...$7D, $F0...$FF, $170...$17F and $1F0...$1FF. This permits quick access to certain memory locations regardless of the bank settings.
- It's full of strange names, like **INTCON**, **CCP1CON** and **SSPSTAT**. This book is not a comprehensive tutorial on assembler programming, and we don't have time to discuss these Special Purpose Registers

except when we find it necessary to use one. And, we'll leave those discussions to Chapters 14 and 15, and those other chapters were we incorporate assembler into our MBasic programs. Of course, detailed information is available on each Special Purpose Register in Microchip's data sheets.

Banking

Let's look at banks and banking. To address memory locations from 0...$1FF (0...511 decimal) requires 9 address bits. Opcodes that involve addressing files must have the file address embedded in the opcode. These opcodes have the following bit structure:



For example, remember our **movwf A** opcode that translated into a machine instruction **00000011001000**? It turns out that the opcode for **movwf** is **0000001**, representing **000000** for the **mov** operation and **1** for the destination code.Since mid-range PICs have only a 14-bit instruction word, this leaves only 7 bits to hold **A**'s address, **1001000**. We've assigned A to represent storage address $48, and seven bits are adequate, since all higher bits are zeros. But, suppose **A**'s address is $196—a perfectly legitimate user file location, but one requiring all nine address bits.

We have a problem. The cleanest fix would have been for Microchip to use a 16-bit op code with 9 bits of address space, but that would have destroyed backwards compatibility with older devices as well as requiring Microchip to extensively rework its silicon layout, an expensive task. We are therefore forced to deal, for historical and compatibility reasons, with the mid-range PIC being able to address only file addresses in the range 0...127 directly in the opcode. Microchip's solution was to break the 9-bit file address into a 7-bit opcode *relative address* portion and a 2-bit *bank address* portion.

The bank address bits are held in the Special Function Register bits **RB1** and **RP0**. (These bits are bits 6 and 5 of the **Status** register.) If we wish to address memory location $196, residing in Bank 3, therefore, we would first set **RB1** and **RB0** to **%1**. The rightmost seven bits of $196, or $16 represent the relative address. (Don't worry if this sounds complicated—the assembler hides much of this complexity from you.)

To simplify setting and clearing the bank bits, we may use an assembler *macro*, **BankSel**, invoked with the following syntax:

```
Banksel FileAddress
```

The term **FileAddress** is the file name. (Remember, the name of a file is defined as a number representing its address, so the assembler simply uses the address value.) A *macro* can be thought of as a "user defined" opcode. Instead of a real opcode that generates a specific machine instruction, the assembler expands a macro into one or more "real" opcodes. The macro's advantages are space savings and automating a repetitive task. In this case, the macro **BankSel** expands into instructions that set or clear **RB1** and **RB0** corresponding to the **FileAddress** value.

Suppose our variable name is **A** and its at address $196. (In assembler, the equate operation, **Equ**, is the equivalent to MBasic's constant assignment operator, **Con**.) How would we use **BankSel**?

```
A       Equ     0x196

BankSel A
movlw           .123
movwf           A
```

OK you're thinking I understand that the **BankSel** macro extracts the two leftmost bits from the value of **A**, but how does **movwf** get the rightmost seven bits without any special instruction? The answer is that since

there are only seven bits of opcode space available for the address, only the rightmost seven bits of **A** are loaded into the opcode, without additional commands. (This simplistic construction may generate assembler warnings, so we'll learn a better way later in this chapter.)

Program Memory

In Chapter 1, we learned that the 16F876/877/876A/877A devices have 8K program memory. The *program counter* in these devices is 13 bits wide and 2^{13} is 8192, so it can address all 8K of program memory. The program counter is the part of the PIC's core that keeps track of which program instruction is currently executing and which program instruction should next be executed. If our assembler code uses all 8K sequentially, so that each program instruction address follows the next in strict numerical order, all is well.

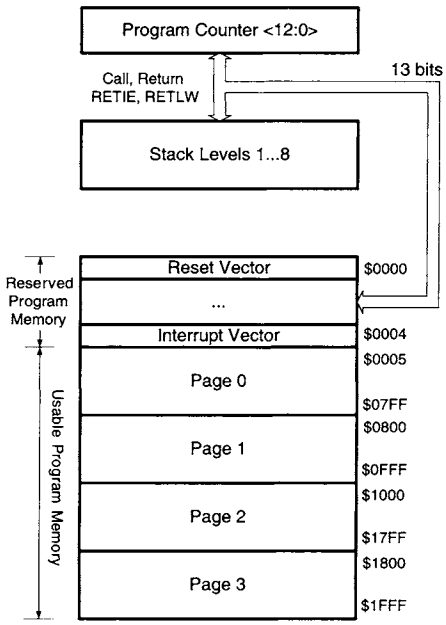


Figure 13-3: Conceptual view—program memory in 16F876/877 devices.

However, when writing a program, it's common to have execution to jump from one location to another. You might write some code in a subroutine that you then *call* (the assembler equivalent of **GoSub**) from multiple locations within your main program. Or, program execution may need to jump around a data table stored in program memory. And, you may wish to repeat some code in the assembler equivalent of a **For...Next** or a **While...WEND** loop. This implies that our opcodes must include instructions to jump execution from the current program counter location to a different location.

Do you see the problem? Our opcode structure must accommodate a series of jump instructions that say, in essence, stop executing the current code and transfer execution to a new program memory address. But our opcode only has 14-bits of space and a full jump address alone is 13 bits. This only leaves 1 bit for all jump opcode instructions, which is unsatisfactory to say the least. So, the engineers at Microchip decided to segment the program memory, and split the program counter into two registers. Figure 13-3 illustrates the program memory structure. Instead of calling them banks, the program memory segments are called *pages*. The 16F876/877 devices have four program memory pages, each 2K long.

There are two jump opcodes in mid-range PICs, the **call** and **goto** instructions. Both reserve 11 bits for the jump address:

13	*11*	*10*	*0*
OPCODE		***Execution Transfer Address***	
3 bits		*11 bits*	

Where then are the other two bits of the 13-bit jump address? Or, more pertinently, since our data structure only deals with eight bits, how do we access a 13-bit program control register? The answer is a bit complicated. The 13-bit program control register is split into two parts: the lower 8 bits are held in the **PCL** register, which may be read or written to, just like any other special purpose register. The upper 5 bits of the program control register are not directly accessible and may be reached only through the **PCLATH** register. We'll end our discussion of the **PCL** and **PCLATH** registers now, and return to them in the context of specific examples in Chapter 14.

The remaining parts of Figure 13-3 are the *stack* memory and the *reset* and *interrupt vectors*. The stack is dedicated memory where the PIC stores return addresses for interrupt and call operations. (This stack is internal to the PIC and is not the same as the general-purpose register file memory that MBasic requires for its stack.) This is transparent to the programmer and for the applications discussed in this book need not further concern you. The reset vector is the location where the program counter begins executing code after a reset, while the interrupt vector is the location where execution jumps to in the event of an interrupt. Normally the programmer will insert a **GoTo** opcode in those locations, pointing to the real reset code, or the real interrupt handler code. Having the reset and interrupt start point hardwired to $0000 and $0004, respectively, permits the programmer to be assured of known code execution point in the event of a reset or interrupt event. In the case of assembler interrupt handlers, Chapter 15 will show us that MBasic takes care of much of this work for us.

Instruction Set Terminology

In reading the opcode summary, we'll make extensive use of the following field descriptors.

Field	Descriptor Meaning
f	*Register file address ($00...$7F)*
W	*Working Register (accumulator)*
b	*Bit address within an 8-bit file register*
k	*Literal field, constant data or label*
x	*Don't care location (0 or 1). The assembler generates code with 0.*
d	*Destination selection:* *d = 0 store result in W* *d = 1 store result in register f* *Default is d = 1*
PC	*Program Counter*
TO	*Time-out Bit*
PD	*Power-down Bit*

One Special Purpose Register that we will deal with frequently when reviewing the opcodes is the **Status** register:

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
IRP	*RP1*	*RP0*	$\overline{TO}$	$\overline{PD}$	*Z*	*DC*	*C*

IRP—Register bank select bit for indirect addressing; covered in Chapter 14.

RP1 and RP0—As we previously saw, the **RP1** and **RP0** bits are the 9th and 8th register file address bits, respectively.

11 = Bank 3
10 = Bank 2
01 = Bank 1
00 = Bank 0

$\overline{TO}$—Time-out bit (the overscore indicates inverted status). We will not further discuss the $\overline{\text{TO}}$ bit.

$\overline{PD}$—Power-down bit (the overscore indicates inverted status). We will not further discuss the $\overline{\text{PD}}$ bit.

Z—Zero bit:

1 = the result of the arithmetic or logic operation is zero
0 = the result of the arithmetic or logic operation is nonzero

DC—Digit carry/Borrow bit (for **ADDWF**, **ADDLW**, **SUBLW** and **SUBWF** instructions). For borrow, the bit sense is reversed:

1 = A carry-out from the 4th low order bit of the result occurred
0 = No carry-out from the 4th low-order bit of the result

C—Carry/Borrow bit (for **ADDWF**, **ADDLW**, **SUBLW** and **SUBWF** instructions):

1 = A carry-out from the Most Significant bit of the result occurred
0 = No carry-out from the Most Significant bit of the result occurred.

For **Borrow**, the bit sense is reversed. A subtraction is executed by adding the two's complement of the second operand. For rotate (**RRF**, **RRL**) instructions, this bit is loaded with the high (or low) order bit of the source register.

OpCodes

The remainder of this chapter is a summary of the 35 16F87x/87xA opcodes. Microchip's Midrange Reference Manual provides more information and should be consulted where necessary.

When reading the summary, remember that a "literal" is Microchip's name for a constant. A literal may be a number entered directly, such as .123 or as a named constant, linked to a numerical value with either MBasic's **Con** operation, or an **Equ** assignment inside an assembler module. The notation **Name <N>** indicates the Nth bit of file register **Name**. N may also be a name, such as **Status <C>** which means the bit named **C** of the file register named **Status**.

In providing examples, we will use Microchip's hexadecimal notation, **0xNN**, instead of Basic Micros's **$NN** identifier.

The field "cycles" indicates how many clock cycles ($F_{OSC}/4$) is required to complete the particular operation.

Standard Notes:

Three notes apply to several opcodes:

1. When an I/O register is modified as a function of itself; for example, **MOVF PORTB,1**, the value used will be that value present on the pins themselves. For example, if the data latch is **%1** for a pin configured as input and is driven low by an external device, the data will be written back with a **%0**.
2. If this instruction is executed on the **TMR0** register (and where applicable, d = **1**), the prescaler will be cleared if assigned to the **Timer0** module.
3. If Program Counter (PC) is modified, or a conditional test is true, the instruction requires two cycles. The second cycle is executed as a **NOP**.

OpCode	Operands	Description	Cycles	Affected	Notes
ADDLW	k	Add Literal and W	1	C, DC, Z	

Addlw adds a literal 'k' to the value currently in **W**. The result remains in **W**. The literal 'k' must be in the range 0...255. If the literal exceeds 255, only the lowest 8 bits will be used.

Example 1:

```
Addlw   .123    ;adds decimal 123 to the current contents of W
```

Remember that **W** is only 8-bits wide and that it is possible the result will overflow **W**. In the event of an overflow, **Status <C>** will be set. If the sum is zero (such as **W** holds 128, to which 128 is added) **Status <Z>** will be set. In this latter example, both **Z** and **C** will be set.

Example 2:

Assume **A** is defined as a byte variable and its address is 0x30. The value stored at address 0x30 is .128 (0x80). Assume **W**'s initial value is 0x10.

```
Addlw   A
```

The result is 0x48, not 0x90. This is because we are adding the numerical value of **A**, which is its *address*, not the *value* of the contents at address **A**. Confusing **A**'s address with the value of the contents of the file at **A**'s address is a common error.

OpCode	Operands	Description	Cycles	Affected	Notes
ADDWF	f, d	Add W and f	1	C,DC,Z	1,2

Addwf adds the value currently in **W** to the value held in a file 'f'. The result is either placed in **W** (destination = **w**) or in the file 'f' (destination = **f**).

Example 1:

```
Addwf   A,f
```

Before the opcode is executed, assume the value at address **A** is .123 and that **W** holds .10. After execution, the value at address A will be .133 and **W**'s value remains at .10.

Example 2:

```
Addwf   A,w
```

Before the opcode is executed, assume the value at address **A** is .123 and that **W** holds .10. After execution, the value at address **A** remains as .123 and **W** will hold .133.

Since both **W** and register files are 8-bits wide, overflow is possible. Overflow will set **Status <C>**, and a zero result will set **Status <Z>**.

OpCode	Operands	Description	Cycles	Affected	Notes
ANDLW	k	AND Literal with W	1	Z	

Andlw performs a bit-by-bit logical AND of the value in **W** and the literal 'k.' The result is retained in the **W** register.

The AND function truth table is:

Input A	Input B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

The logical AND function is often used to "mask off" bits, as in the following example.

Example 1:

Assume **W** holds .123 before the instruction.

```
                ;Before Instruction W: 01111011
Andlw   31      ;                   31: 00011111
                                        --------
                              After W   00011011 = 27
```

We'll see in Chapter 14 how the AND function may be used to calculate the division remainder. It may also be used to test multiple bits to see if all are zero by using a mask literal with **%1** values in the places where the bits to be tested reside. If all bits coinciding with the **%1** mask bits are 0, the result will is 0 and **Status <C>** will be set.

OpCode	Operands	Description	Cycles	Affected	Notes
ANDWF	f, d	AND W with f	1	Z	1,2

Andwf performs a bit-by-bit logical AND of the value in **W** and the value held in a file 'f.' The result is either placed in **W** (destination = **w**) or in the file 'f' (destination = **f**).

The AND function truth table is:

Input A	Input B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

The logical AND function is often used to "mask off" bits, as in the following example.

Example 1:

Assume **W** holds .123 before the instruction and assume file **A** holds .31.

```
                    ;Before Instruction W: 01111011 = .123
Andwf   A,w         ;                   31: 00011111 = .31
                                            --------
                                After W     00011011 = .27
```

The value stored at file address **A** remains .31.

Example 2:

Assume **W** holds .123 before the instruction and assume file **A** holds 31.

```
                    ;Before Instruction W: 01111011 = .123
Andwf   A,f         ;                   31: 00011111 = .31
                                            --------
                                After f     00011011 = .27
```

The value in **W** remains .123.

OpCode	Operands	Description	Cycles	Affected	Notes
BCF	f, b	Bit Clear f	1		1,2

BCF clears (makes it a zero) bit 'b' of a file 'f'. Bit 'b' may be identified with a number (0...7) or a named constant in the range 0...7.

Example 1:

Assume PortB is set to be an output and we wish to set pin **B0** to low.

```
Bcf     PortB,0
```

OpCode	Operands	Description	Cycles	Affected	Notes
BSF	f, b	Bit Set f	1		1,2

BSF sets (makes it a one) bit' b' of a file 'f'. Bit 'b' may be identified with a number (0...7) or a named constant in the range 0...7.

Example 1:

Assume PortB is set to be an output and we wish to set pin **B0** to high.

```
Bsf     PortB,0
```

OpCode	Operands	Description	Cycles	Affected	Notes
BTFSC	f, b	Bit Test f, Skip if Clear	1 (2)		3

Btfsc checks the status of bit 'b' in file 'f'. If bit 'b' is clear (equal to 0), the next opcode is skipped. If bit 'b' is set (equal to 1) the next opcode is executed.

Btfsc is a program flow control opcodes and is analogous to MBasic's **If...Then** test.

Example 1:

Suppose PortB is set as input and has a switch connected to pin **B0**, with a pull-up resistor. Depending upon whether the switch is set (switch open and **B0=%1**) or clear (switch closed and **B0=%0**) we wish to execute different code.

```
...
Btfsc   PortB,0
        GoTo SwitchOpenRoutine          ;SwitchOpenRoutine executed if no skip
...                                     ;code from here down is executed otherwise
```

The **Btfsc** opcode tests **PortB**'s bit **0** (pin **B0**) value. If the value is clear (**B0=0**), the next instruction (**GoTo SwitchOpenRoutine**) is skipped and code execution resumes with the statement immediately following the skipped instruction. If **PortB**, bit **0** is set (**B0=1**) the next instruction is not skipped so the **GoTo SwitchOpenRoutine** statement executes and program execution branches to the code beginning with the label **SwitchOpenRoutine**.

A common use for **btfsc** is to test the result of an arithmetic or logical operation and branch program flow based upon whether **Status <C>** or **Status <Z>** flags are set or clear.

OpCode	Operands	Description	Cycles	Affected	Notes
BTFSS	f, b	Bit Test f, Skip if Set	1 (2)		3

Btfss checks the status of bit 'b' in file' f'. If bit 'b' is set (equal to 1), the next opcode is skipped. If bit 'b' is clear (equal to 0) the next opcode is executed. **Btfss** is the mirror image of **btfsc**.

Btfsc is a program flow control opcodes and is analogous to MBasic's **If...Then** test.

Example 1:

Suppose PortB is set as input and has a switch connected to pin **B0**, with a pull-up resistor. Depending upon whether the switch is set (switch open and **B0=%1**) or clear (switch closed and **B0=%0**) we wish to execute different code.

```
...
Btfss   PortB,0
        GoTo SwitchClosedRoutine        ;SwitchClosedRoutine executed if no skip
...                                     ;code from here down is executed otherwise
```

The **Btfss** opcode tests **PortB**'s bit **0** (pin **B0**) value. If the value is set (**B0=1**), the next instruction (**GoTo SwitchClosedRoutine**) is skipped and code execution resumes with the statement immediately following the skipped instruction. If **PortB**, bit **0** is clear (**B0=0**) the next instruction is *not* skipped so the **GoTo**

SwitchClosedRoutine statement executes and program execution branches to the code beginning with the label **SwitchClosedRoutine**.

A common use for **btfss** is to test the result of an arithmetic or logical operation and branch program flow based upon whether **Status <C>** or **Status <Z>** flags are set or clear.

OpCode	Operands	Description	Cycles	Affected	Notes
CALL	k	Call Subroutine	2		

Call transfers program execution to a subroutine, similar to MBasic's **GoSub** function. Upon return from the subroutine, program execution resumes with the opcode immediately following the **Call** operator.

Example 1:

```
            …
            Call SubOne
            …

SubOne
            …       ;subroutine code goes here
            Return
```

The operand for **Call** is a literal value, in the range 0…2047. However, in most instances, you will wish to use a label, such as **SubOne** in the example, and allow the assembler to resolve the label into a numerical argument for the **Call** opcode. As we learned earlier, the range of the call is one program page and if the jump distance is greater, the two extra page bits must be generated. This is an advanced topic and will not be further considered in this chapter.

OpCode	Operands	Description	Cycles	Affected	Notes
CLRF	f	Clear f	1	Z	2

Clrf clears, or sets to 0 all bits in file 'f'. Since the result is zero, **Status <Z>** is set by this operation.

Example 1:

Assume file **A** has been defined and that it holds the value .123.

```
                ;Before instruction A= .123 01111011
Clrf    A       ;After Clrf A       A= 0    00000000
```

OpCode	Operands	Description	Cycles	Affected	Notes
CLRW	–	Clear W	1	Z	

Clrw clears, or sets all bits to 0 in register **W**. Since the result is zero, **Status <Z>** is set by this operation.

Example 1:

Assume **W** holds the value .123.

```
                ;Before instruction W= .123 01111011
Clrw            ;After Clrw A       W= 0    00000000
```

OpCode	Operands	Description	Cycles	Affected	Notes
CLRWDT	–	Clear Watchdog Timer	1	$\overline{TO}$,$\overline{PD}$	

ClrWdt clears the watchdog timer. It also sets the $\overline{\text{TO}}$ and $\overline{\text{PD}}$ bits.

OpCode	Operands	Description	Cycles	Affected	Notes
COMF	f, d	Complement f	1	Z	1,2

Comf performs a one's complement on the contents of file 'f.' The result is either placed in **W** (destination = **w**) or in the file 'f' (destination = **f**). The one's complement operation replaces all 0's by one's and vice versa.

Example 1:

Assume file **A** has been defined and that it holds the value .123.

```
                    ;Before instruction A= .123 01111011
        Comf   A,f  ;After Comf A       A= .132 10000100
```

In 8-bit arithmetic used in **W** and **A**, the sum of **A** and its one's complement is 255. Note this is not the same as for two's complement, where the sum of **A** and its two's complement is 256.

OpCode	Operands	Description	Cycles	Affected	Notes
DECF	f, d	Decrement f	1	Z	1,2

Decf decrements by 1 the value in file 'f'. If, after decrementing, the result is 0, **Status <Z>** is set. The result is either placed in **W** (destination = **w**) or in the file 'f' (destination = **f**). **Decf** is the assembler analog of **A=A-1**.

Example 1:

Assume file **A** has been defined and that it holds the value .123.

```
                    ;Before instruction A= .123 01111011
        Decf   A,f  ;After Decf A       A= .122 01111010
```

The result of decrementing 0 is, of course .255.

Decf is useful in control loops, as it can be combined with a **btfss** or **btfsc** operator to provide the assembler analog of MBasic's **For...Next** loop.

OpCode	Operands	Description	Cycles	Affected	Notes
DECFSZ	f, d	Decrement f, Skip if 0	1(2)		1,2,3

Decfsz combines the decrement operation with a zero result test; the value in file 'f' is decremented by 1 and if the result is 0, the next instruction is skipped. The result is either placed in **W** (destination = **w**) or in the file 'f' (destination = **f**).

Example 1:

Assume file **A** has been defined and that it holds the value .123.

```
Loop    ...             ;this code is executed 123 times
        Decfsz  A,f
        GoTo    Loop
...
```

When program execution hits the **Decfsz** instruction, the then current value of **A** is decremented by 1 and the result tested. If the result is nonzero, the next instruction is executed and program flow jumps back to the label **Loop**. If the result is zero, the **GoTo Loop** statement is skipped over and program execution goes to the first opcode following the **GoTo**.

Example 1 is the assembler analog of:

```
A = 123
Repeat
      …
   A=A-1
Until A=0
```

Note that the assembler code in Example 1 is executed 123 times, not 124 times because the test for zero is made at the bottom of the loop, *after* decrementing **A**.

OpCode	Operands	Description	Cycles	Affected	Notes
GOTO	k	Go to Address	2		

GoTo transfers program execution to a code sequence. Its effect is similar to MBasic's **GoTo** statement.

Example 1:

Assume file **A** has been defined and that it holds the value .123.

```
Loop    …               ;this code is executed 123 times
        Decfsz  A,f
        GoTo    Loop
```

When A is nonzero, the **GoTo** statement is executed and program flow jumps to the address identified by the label **Loop**.

The operand for **GoTo** is a literal value, in the range 0…2047. However, in most instances, you will wish to use a label, such as **Loop** in the example, and allow the assembler to resolve the label into a numerical argument for the **GoTo** opcode. As we learned earlier, the range of the jump is one program page and if the jump distance is greater, the two extra page bits must be generated. This is an advanced topic and will not be further considered in this chapter.

OpCode	Operands	Description	Cycles	Affected	Notes
INCF	f, d	Increment f	1	Z	1,2

Incf increments by 1 the value in file 'f.' If, after incrementing, the result is 0, **Status <Z>** is set. The result is either placed in **W** (destination = **w**) or in the file 'f' (destination = **f**). **Incf** is the assembler analog of **A=A+1**.

Example 1:

Assume file **A** has been defined and that it holds the value .123.

```
                ;Before instruction A= .123 01111011
Incf    A,f     ;After Decf A       A= .124 01111100
```

The result of incrementing .255 is, of course .0 In this case, **Status <Z>** will be set.

Incf is useful in control loops, as it can be combined with a **btfss** or **btfsc** operator to provide the assembler analog of MBasic's **For…Next** loop or other control structures.

OpCode	Operands	Description	Cycles	Affected	Notes
INCFSZ	f, d	Increment f, Skip if 0	1(2)		1,2,3

Incfsz combines the increment operation with a zero result test; the value in file 'f' is incremented by 1 and if the result is 0, the next instruction is skipped. The result is either placed in **W** (destination = **w**) or in the file 'f' (destination = **f**).

Example 1:

Assume file **A** has been defined and that it holds the value .123.

```
Loop    …               ;this code is executed 133 times not 123 times
        Incfsz  A,f
        GoTo    Loop
        …
```

When program execution hits the **Incfsz** instruction, the then current value of **A** is incremented by 1 and the result tested. If the result is nonzero, the next instruction is executed and program flow jumps back to the label **Loop**. If the result is zero, the **GoTo Loop** statement is skipped over and program execution goes to the first opcode following the **GoTo**.

Example 1 is the assembler analog of:

```
A = 123
Repeat
      …
   A=A+1
Until A=0
```

A becomes 0, of course, due to rollover. Rollover occurs when **A** should be 256, but, since **A** only holds 8-bits it instead is zero. (256 is **%1 00000000** where the first **%1** is the 9th bit.) The number of times the loop code will execute is thus 256-123, or 133 times.

OpCode	Operands	Description	Cycles	Affected	Notes
IORLW	k	Inclusive OR Literal with W	1	Z	

Iorlw performs a bit-by-bit logical OR of the value in **W** and the literal 'k.' The result is retained in the **W** register.

The OR function truth table is:

Input A	Input B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

Example 1:

Assume **W** holds .123 before the instruction.

```
                ;Before Instruction W: 01111011 = .123
Iorlw   31      ;                   31: 00011111 = .31
                                        --------
                              After     W: 01111111 = 127
```

OpCode	Operands	Description	Cycles	Affected	Notes
IORWF	f, d	Inclusive OR W with f	1	Z	1,2

Iorwf performs a bit-by-bit logical OR of the value in **W** and the value held in a file 'f.' The result is either placed in **W** (destination = **w**) or in the file 'f' (destination = **f**).

The OR function truth table is:

Input A	Input B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

Example 1:

Assume **W** holds .123 before the instruction and assume file **A** holds .31.

```
                    ;Before Instruction W: 01111011 = .123
Andwf   A,w         ;                   31: 00011111 = .31
                                            --------
                                After   W: 01111111 = .127
```

The value stored at file address **A** remains .31.

Example 2:

Assume **W** holds .123 before the instruction and assume file **A** holds 31.

```
                    ;Before Instruction W: 01111011
Andwf   A,f         ;                   31: 00011111
                                            --------
                                After   f: 01111111 = 127
```

The value in **W** remains .123.

OpCode	Operands	Description	Cycles	Affected	Notes
MOVF	f, d	Move f	1	Z	1,2

Movf copies the value in file 'f' to either **W** or back to the same file 'f.' The result is either placed in **W** (destination = **w**) or in the source file 'f' (destination = **f**).

It may seem pointless to copy a file back to itself. However, copying the file back to itself sets the **Status <Z>** flag if the file value is 0, so it permits us to test the file 'f' for zero value. **Movf** is most often used to load **W** with the value of file 'f.'

Example 1:

Assume file **A** has been defined and that it holds the value .123.

```
Movf    A,w
```

Register **W** will contain .123 after the operation is executed.

Remember—**movf** does not allow you to copy the contents of one file to another file. The copy is either to **W** or back to the source file.

OpCode	Operands	Description	Cycles	Affected	Notes
MOVLW	k	Move Literal to W	1		

Movlw loads register **W** with the value of the literal 'k' where k is in the range 0...255.

Example 1:

```
Movlw   .123
```

After this operation is executed **W** contains the value .123.

OpCode	Operands	Description	Cycles	Affected	Notes
MOVWF	*f*	*Move W to f*	*1*		

Movwf copies the contents of register W to file 'f.' **Movwf** is how results in **W** are placed in a file 'f.'

Example 1:

Assume file **A** has been defined and that it holds an unknown value

```
Movlw   .123
Movwf   A
```

First, we load register **W** with the literal .123. Next, the **movwf** operation copies **W**'s value to file **A**. After these two operations complete, **A** holds the value .123.

OpCode	Operands	Description	Cycles	Affected	Notes
NOP	–	*No Operation*	*1*		

NOP is the "no operation" code. It consumes one cycle and is used to generate brief time delays such as to balance execution times where different code segments must execute in the identical time.

Example 1:

```
NOP
```

All register and file values are unchanged after the **NOP**.

OpCode	Operands	Description	Cycles	Affected	Notes
RETFIE	–	*Return from Interrupt*	*2*		

Retfie is used to terminate an interrupt handler.

Using **Retfie** is advanced topic and this opcode will not be further discussed. Our use of MBasic's assembler interrupt support will not require us to use the **Retfie** operation.

OpCode	Operands	Description	Cycles	Affected	Notes
RETLW	*k*	*Return with Literal in W*	*2*		

Retlw returns from a **GoTo** with a literal 'k' loaded in the **W** register.

Retlw is used most often for a data or look-up table, to return, and allows us to implement the assembler equivalent of MBasic's **ByteTable**, whereby we may access X(n), the value of the n^{th} data entry.

Example 1:

Assume file **A** is declared and holds a value between 0…7.

```
movf    A,w
call    GetHalfPattern
…
GetHalfPattern
        addwf   PCL,f
        retlw   0x08
        retlw   0x0C
        retlw   0x04
        retlw   0x06
        retlw   0x02
        retlw   0x03
        retlw   0x01
        retlw   0x09
```

W is loaded with the index into the table. After the **Call** operation, **W** holds the indexed value. For example, if **A** holds .3, after the **Call** operation, **W** holds 0x6.

OpCode	Operands	Description	Cycles	Affected	Notes
RETURN	–	Return from Subroutine	2		

Return transfers program execution back to the **Call** statement at the end of a subroutine, similar to MBasic's **Return** operation. Upon return from the subroutine, program execution resumes with the opcode immediately following the **Call** operator.

Example 1:

```
        ...
        Call SubOne
        ...

SubOne
        ...       ;subroutine code goes here
        Return
```

OpCode	Operands	Description	Cycles	Affected	Notes
RLF	f, d	Rotate Left f through Carry	1	C	1,2

RLF rotates the value held in file 'f' one bit to the left, through the carry flag. The result is either placed in **W** (destination = **w**) or in the source file 'f' (destination = **f**). This process re-circulates the bits.

Example 1:

Assume file **A** is declared and holds the value .179.

```
RLF A,f
RLF A,f
```

Status	Register A								
<C>	B7	B6	B5	B4	B3	B2	B1	B0	Comments
0	1	0	1	1	0	0	1	1	Before rotate command, A = 179
1	0	1	1	0	0	1	1	0	First shift to the left. A= 102
0	1	1	0	0	1	1	0	1	Second shift to left; note the 1 shifted through carry has now reappeared at B7. A= 205

If you wish a straight shift, not re-circulation, it is necessary to clear **Status <C>** after each **RLF**.

OpCode	Operands	Description	Cycles	Affected	Notes
RRF	f, d	Rotate Right f through Carry	1	C	1,2

RRF rotates the value held in file 'f' one bit to the right, through the carry flag. The result is either placed in **W** (destination = **w**) or in the source file 'f' (destination = **f**). This process re-circulates the bits.

Example 1:

Assume file **A** is declared and holds the value .123.

```
RRF A,f
RRF A,f
```

Status	Register A								
<C>	B7	B6	B5	B4	B3	B2	B1	B0	Comments
0	0	0	1	1	0	0	1	1	Before rotate command, A = 123
1	0	0	0	1	1	0	0	1	First shift to the right. A= 25
1	1	0	0	0	1	1	0	0	Second shift to right; note the 1 shifted through carry has now reappeared at B7. A= 140

If you wish a straight shift, not re-circulation, it is necessary to clear **Status <C>** after each **RRF**.

OpCode	Operands	Description	Cycles	Affected	Notes
SLEEP	–	Go into Standby mode	1	TO,PD	

Sleep places the PIC into standby mode. This is an advanced topic and will not be further discussed.

OpCode	Operands	Description	Cycles	Affected	Notes
SUBLW	k	Subtract W from Literal	1	C,DC,Z	

Sublw subtracts the **W** register from the literal 'k' using two's complement arithmetic. Note the order of subtraction is **W = k-W**.

Example 1:

Suppose file A is declared and holds the value .47

```
movf A&0x7F,w          ;w = .47
sublw   .100           ;w = (.100 - .47) = .53
```

Example 2:

Suppose file A is declared and holds the value .123

```
movf A&0x7F,w          ;w = .123
sublw   .100           ;w = (.100 - .123) = .233
```

Example 2 has the subtrahend larger than the minuend. The result may be calculated using the two's complement offset, .256:

Result = .256 + .100 - .123 = .233. Alternatively, the .256 may be viewed as a carry to the 9th bit.

OpCode	Operands	Description	Cycles	Affected	Notes
SUBWF	f, d	Subtract W from f	1	C,DC,Z	1,2

Subwf subtracts the value in register **W** from the value in file 'f' using two's complement arithmetic. The result is either placed in **W** (destination = **w**) or in the source file 'f' (destination = **f**). Note the order of subtraction is **Result = f-W**.

Example 1:

Suppose file **A** is declared and holds the value .100

```
                       ; A = .100
movlw .47              ; W = .47
subwf A,f              ; A = (.100-.47) = .53
```

Example 2:

Suppose file **A** is declared and holds the value .100

```
                          ; A = .100
movlw .123                ; W = .123
subwf A&0x7F,f ; A = (.100 - .123) = .233
```

Example 2 has the subtrahend larger than the minuend. The result may be calculated using the two's complement offset, .256:

Result = .256 + .100 – .123 = .233. Alternatively, the .256 may be viewed as a carry to the 9th bit.

OpCode	Operands	Description	Cycles	Affected	Notes
SWAPF	f, d	Swap nibbles in f	1		1,2

Swapf reverses the high and low nibble in file 'f.' The result is either placed in **W** (destination = **w**) or in the source file 'f' (destination = **f**).

Example 1:

Assume file **A** has been defined and that it holds the value .123.

```
                ;Before Instruction A: 01111011 = .123
Swapf   A,f     ;After Swapf        A: 10110111 = .183
```

OpCode	Operands	Description	Cycles	Affected	Notes
XORLW	k	Exclusive OR Literal with W	1	Z	

Xorlw performs a bit-by-bit logical XOR of the value in **W** with the literal 'k.' The result is retained in the **W** register.

Input A	Input B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Example 1:

Assume **W** holds .123 before the instruction.

```
                ;Before Instruction W: 01111011 = .123
Xorlw   31      ;                   31: 00011111 = .31
                                        --------
                              After   W: 01100100 = .100
```

OpCode	Operands	Description	Cycles	Affected	Notes
XORWF	f, d	Exclusive OR W with f	1	Z	1,2

Xorwf performs a bit-by-bit logical XOR of the value in **W** with the literal 'k.' The result is retained in the **W** register.

Input A	Input B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Example 1:

Assume the file **A** is declared and holds the value .123 and that **W** holds .31 before the instruction.

```
                    ;Before Instruction A: 01111011  = .123
Xorwf   A,w         ;                   W: 00011111  = .31
                                           --------
                              After     W: 01100100  = .100
```

References

[13-1] A complete data sheet for most PICs comprises two elements; (a) a detailed "family" reference manual and (b) the particular device datasheet. MBasic supports only PICs from Microchip's "midrange" family and the associated PICmicro™ Mid-Range MCU Family Reference Manual may be downloaded at http://www.microchip.com/download/lit/suppdoc/refernce/midrange/33023.pdf. This is a 688-page document, in almost mind numbing detail, but nonetheless is an essential reference to a complete understanding of PICs. For individual PIC family member datasheets, the easiest source is to go to http://www.microchip.com/1010/pline/picmicro/index.htm and select either the PIC12 or PIC16 group and from that link then select the individual PIC device.

[13-2] Benson, David, *Easy PIC'n A Beginners Guide to Using PIC Microcontrollers version 3.1*, Square 1 Electronics, Kelseyville, CA (1999).

[13-3] Benson, David, *PIC'n Up the Pace PIC Microcontroller Applications Guide version 1.1*, Square 1 Electronics, Kelseyville, CA (1999).

[13-4] Predko, Myke, *Programming and Customizing PICmicro Microcontrollers, Second Ed.*, McGraw Hill, New York (2002).

[13-5] Predko, Myke, *PICMicro Microcontroller Pocket Reference*, McGraw Hill, New York (2001).