APPENDIX A

PIC18 INSTRUCTIONS: FORMAT AND DESCRIPTION

OVERVIEW

In the first section of this appendix, we describe the instruction format of the PIC18. Special emphasis is placed on the instructions using both WREG and file registers. This section includes a list of machine cycles (clock counts) for each of the PIC18 instructions.

In the second section of this appendix, we describe each instruction of the PIC18. In many cases, a simple programming example is given to clarify the instruction.

This Appendix deals mainly with PIC18 instructions. In Section A.1, we describe the instruction formats and categories. In Section A.2, we describe each instruction of PIC18 with some examples.

SECTION A.1: PIC18 INSTRUCTION FORMATS AND CATEGORIES

As shown in Figure A-1, the PIC18 instructions fall into five categories:

1. Bit-oriented instructions
2. Intructions using a literal value
3. Byte-oriented instructions
4. Table read and write instructions
5. Control instructions using branch and call

In this section, we describe the format and syntax with special emphasis placed on byte-oriented instructions. For some of the instructions, the reader needs to review the concepts of access bank and bank registers in Chapter 6 (Section 6.3).

Bit-oriented instructions

The bit-oriented instructions perform operations on a specific bit of a file register. After the operation, the result is placed back in the same file register. For example, the "BCF f,b,a" instruction clears a specific bit of fileReg. See Table A-1. In these types of instructions, the b is the specific bit of the fileReg, which can be 0 to 7, representing the D0 to D7 bits of the register. The fileReg location can be in the bank register called access bank (if a = 0) or a location within other bank registers (if a = 1). Notice that if a = 0, the assembler assumes the access bank automatically.

Table A-1: Bit-Oriented Instructions (from Microchip datasheet)

Mnemonic, Operands		Description	Cycles
BIT-ORIENTED FILE REGISTER OPERATIONS			
BCF	f, b, a	Bit Clear f	1
BSF	f, b, a	Bit Set f	1
BTFSC	f, b, a	Bit Test f, Skip if Clear	1 (2 or 3)
BTFSS	f, b, a	Bit Test f, Skip if Set	1 (2 or 3)
BTG	f, d, a	Bit Toggle f	1

Look at the examples that follow for clarification of bit-oriented instructions:

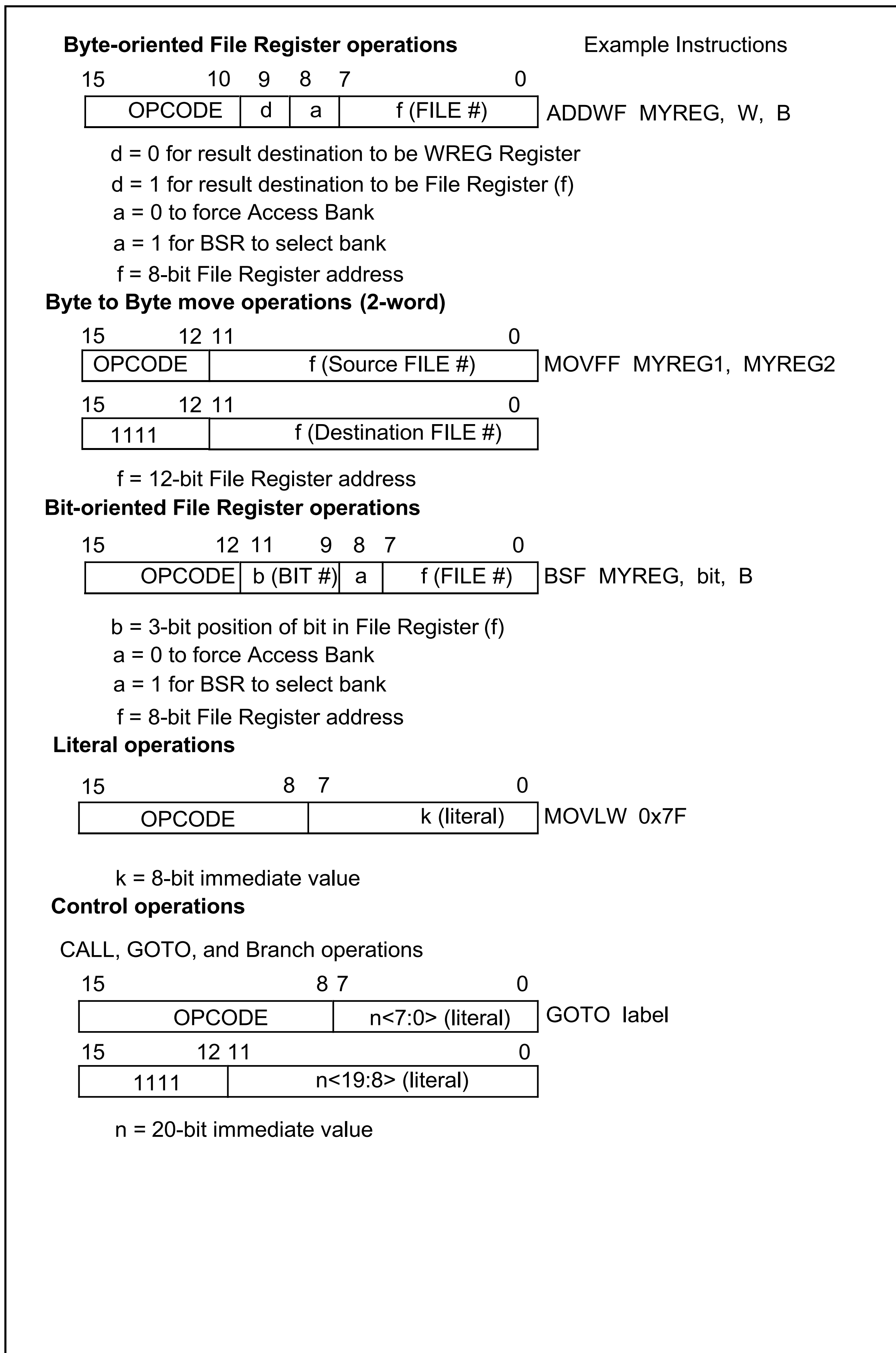


Figure A-1. General Formatting of PIC18 Instructions (From MicroChip)

```
BCF  PORTB,5        ;clear bit D5 of PORTB
BCF  TRISB,4        ;clear bit D4 of TRISC reg
BTG  PORTC,7        ;toggle bit D7 of PORTC
BTG  PORTD,0        ;toggle bit D0 of PORTD
BSF  STATUS,C       ;set carry flag to one
```

The following example uses the fileReg in the access bank:

```
MyReg   SET  0x30   ;set aside loc 30H for MyReg
MOVLW 0x0           ;WREG = 0
MOVWF MyReg         ;MyReg = 0
BTG MYReg,7         ;toggle bit D7 of MyReg
BTG MYReg,5         ;toggle bit D5 of MyReg
```

The following example uses the fileReg in the access bank:

```
MyReg   SET   0x50 ;set aside loc. 50H for MyReg
MOVLW  0x0         :WREG = 0
MOVWF  MyReg       ;MyReg = 0
BTG   MYReg,2      ;toggle bit D2 of MyReg
BTG   MYReg,4      ;toggle bit D4 of MyReg
```

As we discuss in Chapter 6, when using a bank other than the access bank, we must load the BSR (bank select register) with the desired bank number, which can go from 1 to F (in hex), depending on the family member. We do that by using the MOVLB instruction. Look at the following examples.

The example below uses a location in Bank 2 (RAM locations 200–2FFH).

```
YReg  SET 0x30      ;set aside loc 30H for YReg
MOVLB 0x2      ;use Bank 2 (address loc 230H)
MOVLW 0x0           :WREG = 0
MOVWF YReg          ;YReg = 0
BTG  YReg,7,1  ;toggle bit D7 of YReg in bank 2
BTG  YReg,5,1  ;toggle bit D5 of YReg in bank 2
```

The example below uses a location in Bank 4 (RAM locations 400–4FFH).

```
ZReg  SET 0x10      ;set aside loc 10H for ZReg
MOVLB 0x4      ;use Bank 4 (address loc 410H)
MOVWL 0x0           ;WREG = 0
MOVWF ZReg          ;ZReg = 0
BSF ZReg,6,1 ;set HIGH bit D6 of ZReg in bank 4
BSF ZReg,1,1 ;set HIGH bit D1 of ZReg in bank 4
```

Notice that all the bit-oriented instructions start with letter B (bit). The branch instructions also start with letter B, like “BZ target” for branch if zero, but they are not bit-oriented.

Table A-2: Literal Instructions (from Microchip datasheet)

Mnemonic, Operands		Description	Cycles
LITERAL OPERATIONS			
ADDLW	k	Add literal and WREG	1
ANDLW	k	AND literal with WREG	1
IORLW	k	Inclusive OR literal with WREG	1
LFSR	f, k	Move literal (12-bit) 2nd word to FSRx 1st word	2
MOVLB	k	Move literal to BSR <3:0>	1
MOVLW	k	Move literal to WREG	1
MULLW	k	Multiply literal with WREG	1
RETLW	k	Return with literal in WREG	2
SUBLW	k	Subtract WREG from literal	1
XORLW	k	Exclusive OR literal with WREG	1

Instructions using literal values

In this type of instruction, an operation is performed on the WREG register and a fixed value called k. See Table A-2. Because WREG is only 8-bit, the k value cannot be greater than 8-bit. Therefore, the k value is between 0–255 (00–FF in hex). After the operation, the result is placed back in WREG. Look at the following examples for clarification:

```
MOVLW     0x45 ;WREG = 45H
ADDLW     0x24 ;WREG = 45H + 24H = 69H

MOVLW     0x35 ;WREG = 35H
ANDLW     0x0F ;WREG = 35H ANDed with 0FH = 05H

MOVLW     0x55 ;WREG = 55H
XORLW     0xAA ;WREG = 55H EX-ORed with AAH = FFH
```

Byte-oriented instructions

There are two groups of instructions in this category. In the first group, the operation is performed on the file register and the result is placed back in the file register. The instruction “CLRF f,a” is an example in this group. See Table A-3. In the second group, the operation involves both fileReg and WREG. As a result, we have the options of placing the result in fileReg or in WREG. As an example in this group, examine the “ADDWF f,d,a” instruction. The destination for the result can be WREG (if d = 0) or file register (if d = 1). For the fileReg location, it can be in the access bank (if a = 0) or in other bank registers (if a = 1). Also notice that if a = 0, the assembler assumes that automatically.

Table A-3: Byte-Oriented Instructions (from Microchip datasheet)

Mnemonic, Operands		Description	Cycles
BYTE-ORIENTED FILE REGISTER OPERATIONS			
ADDWF	f, d, a	Add WREG and f	1
ADDWFC	f, d, a	Add WREG and Carry bit to f	1
ANDWF	f, d, a	Add WREG with f	1
CLRF	f, a,	Clear f	1
COMF	f, d, a	Complement f	1
CPFSEQ	f, a,	Compare f with WREG, skip =	1
CPFSGT	f, a,	Compare f with WREG, skip >	1
CPFSLT	f, a,	Compare f with WREG, skip <	1
DECF	f, d, a	Decrement f	1
DECFSZ	f, d, a	Decrement f, Skip if 0	1
DCFSNZ	f, d, a	Decrement f, Skip if Not 0	1
INCF	f, d, a	Increment f	1
INCFSZ	f, d, a	Increment f, Skip if 0	1
INFSNZ	f, d, a	Increment f, Skip if Not 0	1
IORWF	f, d, a	Inclusive OR WREG with f	1
MOVF	f, d, a	Move f	1
MOVFF	f_s, f_d	Move f_s(source) to 1st word f_d(destination) 2nd word	2
MOVWF	f, a	Move WREG to f	1
MULWF	f, a	Multiply WREG with f	1
NEGF	f, a	Negate f	1
RLCF	f, d, a	Rotate Left f through Carry	1
RLNCF	f, d, a	Rotate Left f (No Carry)	1
RRCF	f, d, a	Rotate Right f through Carry	1
RRNCF	f, d, a	Rotate Right f (No Carry)	1
SETF	f, a,	Set f	1
SUBFWB	f, d, a	Subtract f from WREG with borrow	1
SUBWF	f, d, a	Subtract WREG from f	1
SUBWFB	f, d, a	Subtract WREG from f with borrow	1
SWAPF	f, d, a	Swap nibbles in f	1
TSTFSZ	f, a	Test f, Skip if 0	1
XORWF	f, d, a	Exclusive OR WREG with f	1

Look at the following examples.

When d = 0 and a = 0:

```
MyReg  SET  0x20        ;loc 20H for MyReg
MOVLW 0x45              ;WREG = 45H
MOVWF MyReg             ;MyReg = 45H
MOVLW 0x23              ;WREG = 23H
ADDWF MyReg        ;WREG = 68H (45H + 23H = 68H)
```

In the above example, the last instruction could have been coded as "ADDWF MyReg,0,0".

When d = 1 and a = 0:

```
MyReg SET  0x20         ;loc 20H for MyReg
MOVLW 0x45              ;WREG = 45H
MOVWF MyReg             ;MyReg = 45H
MOVLW 0x23              ;WREG = 23H
ADDWF MyReg,F      ;MyReg = 68H (45H + 23H = 68H)
```

In the above example, the last instruction could have been coded as "ADDWF MyReg,F,0" or "ADDWF MyReg,1,0". As far as the MPLAB is concerned, they mean the same thing. Notice that the use of letter F in "ADDWF MyReg,F" is being used in place of 1.

To use banks other than the access bank, we must load the BSR register first. The following example uses a location in Bank 2 (RAM location 200–2FFH).

When d = 0 and a = 1:

```
MyReg SET 0x30 ;set aside location 30H for MyReg
MOVLB0x2       ;use Bank 2 (address loc 230H)
MOVLW 0x45              ;WREG = 45H
MOVWF MyReg,1           ;MyReg = 45H (loc 230H)
MOVLW 0x23              ;WREG = 23H
ADDWF MyReg,1  ;WREG = 68H (add loc 230H to W)
```

When d = 1 and a = 1:

```
MyReg  SET  0x20        ;loc 20H for MyReg
MOVLB 0x4               ;use bank 4
MOVLW 0x45              ;WREG = 45H
MOVWF MyReg             ;MyReg = 45H (loc 420H)
MOVLW 0x23              ;WREG = 23H
ADDWF MyReg,F,1         ;MyReg = 68H (loc 420)
```

Register-indirect addressing mode uses FSRx as a pointer to RAM location. We have three registers, FSR0, FSR1, and FSR2, that can be used for pointers.

Examples:

```
ADDWF POSTINC0 ;add to W data pointed to by FSR0,
               ;also increment FSR0

ADDWF POSTINC1 ;add to W data pointed to by FSR1
               ;also increment FSR1
```

See Example 6-6 in Chapter 6.

Table processing instructions

The table processing instructions allow us to read fixed data located in the program ROM of the PIC18. See Table A-4. They also allow us to write into the program ROM if it is Flash memory. Chapter 14 discusses the TBLRD and TBLWRT instructions in detail. It also shows how to use table read and write to access the EEPROM.

Table A-4: Table Processing Instructions (from Microchip datasheet)

Mnemonic, Operands	Description	Cycles
DATA ←→ PROGRAM MEMORY OPERATIONS		
TBLRD*	Table Read	2
TBLRD*+	Table Read with post-increment	2
TBLRD*-	Table Read with post-decrement	2
TBLRD+*	Table Read with pre-increment	2
TBLWT*	Table Write	2
TBLWT*+	Table Write with post-increment	2
TBLWT*-	Table Write with post-decrement	2
TBLWT+*	Table Write with pre-increment	2

Control instructions

The control instructions such as branch and call deal mainly with flow control. See Table A-5. We must pay special attention to the target address of the control instructions. The target address for some of the branch instructions such as BZ (branch if zero) cannot be farther than 128 bytes away from the current instruction. The CALL instruction allows us to call a subroutine located anywhere in the 2M ROM space of the PIC18. See the individual instructions in the next section for further discussion on this issue.

Table A-5: Control Instructions (from Microchip datasheet)

Mnemonic, Operands		Description	Cycles
CONTROL OPERATIONS			
BC	n	Branch if Carry	1
BN	n	Branch if Negative	1
BNC	n	Branch if Not Carry	1
BNN	n	Branch if Not Negative	1
BNOV	n	Branch if Not Overflow	1
BNZ	n	Branch if Not Zero	1
BOV	n	Branch if Overflow	1
BRA	n	Branch Unconditionally	2
BZ	n	Branch if Zero	1
CALL	n, s	Call subroutine 1st word 2nd word	2
CLRWDT	—	Clear Watchdog Timer	1
DAW	—	Decimal Adjust WREG	1
GOTO	n	Go to address 1st word 2nd word	2
NOP	—	No Operation	1
NOP	—	No Operation	1
POP	—	Pop top of return stack (TOS)	1
PUSH	—	Push top of return stack (TOS)	1
RCALL	n	Relative Call	2
RESET		Software device RESET	1
RETFIE	s	Return from interrupt enable	2
RETLW	k	Return with literal in WREG	2
RETURN	s	Return from Subroutine	2
SLEEP	—	Go into standby mode	1

SECTION A.2: THE PIC18 INSTRUCTION SET

In this section we provide a brief description of each instruction with some examples.

ADDLW K **Add Literal to WREG**

Function: ADD literal value of k to WREG
Syntax: ADDLW k

This adds the literal value of k to the WREG register, and places the result back into WREG. Because register WREG is one byte in size, the operand k must also be one byte.

The ADD instruction is used for both signed and unsigned numbers. Each one is discussed separately. See Chapter 5 for discussion of signed numbers.

Unsigned addition

In the addition of unsigned numbers, the status of C, DC, Z, N, and OV may change. The most important of these flags is C. It becomes 1 when there is a carry from D7 out in 8-bit (D0–D7) operations.

Example:

```
MOVLW 0x45          ;WREG = 45H
ADDLW 0x4F     ;WREG = 94H (45H + 4FH = 94H)
                    ;C = 0
```

Example:

```
MOVLW  0xFE         ;WREG = FEH
ADDLW  0x75         ;WREG = FE + 75 = 73H
                    ;C = 1
```

Example:

```
MOVLW  0x25         ;WREG = 25H
ADDLW  0x42    ;WREG = 67H (25H + 42H = 67H)
                    ;C = 0
```

Notice that in all the above examples we ignored the status of the OV flag. Although ADD instructions do affect OV, it is in the context of signed numbers that the OV flag has any significance. This is discussed next.

Signed addition and negative numbers

In the addition of signed numbers, special attention should be given to the overflow flag (OV) because this indicates if there is an error in the result of the addition. There are two rules for setting OV in signed number operation. The overflow flag is set to 1:

1. If there is a carry from D6 to D7 and no carry from D7 out.
2. If there is a carry from D7 out and no carry from D6 to D7.
   Notice that if there is a carry both from D7 out and from D6 to D7, OV = 0.

Example:

```
        MOVLW +D'8'    ;W = 0000 1000
        ADDLW +D'4'    ;W = 0000 1100 OV = 0,
                       ;C = 0, N = 0
```

Notice that N = D7 = 0 because the result is positive, and OV = 0 because there is neither a carry from D6 to D7 nor any carry beyond D7. Because OV = 0, the result is correct [(+8) + (+4) = (+12)].

Example:

```
        MOVLW +D'66'   ;W = 0100 0010
        ADDLW +D'69'   ;W = 1000 0101 = -121
        ADDWF          ;W = 1000 0111 = -121
             ;(INCORRECT) C = 0, N = D7 = 1, OV = 1
```

In the above example, the correct result is +135 [(+66) + (+69) = (+135)], but the result was –121. OV = 1 is an indication of this error. Notice that N = 1 because the result is negative; OV = 1 because there is a carry from D6 to D7 and C = 0.

Example:

```
        MOVLW -D'12'   ;W = 1111 0100
        ADDLW +D'18'   ;W = W + (+0001 0010)
                       ;W = 0000 0110 (+6) correct
                       ;N = 0, OV = 0, and C = 1
```

Notice above that the result is correct (OV = 0), because there is a carry from D6 to D7 and a carry from D7 out.

Example:

```
        MOVLW -D'30'   ;W = 1110 0010
        ADDLW +D'14'   ;W = W + 0000 1110
                       ;W = 1111 0000 (-16, CORRECT)
                       ;N = D7 = 1, OV = 0, C = 0
```

OV = 0 because there is no carry from D7 out nor any carry from D6 to D7.

Example:

```
        MOVLW -D'126'  ;W = 1000 0010
        ADDLW -D'127'  ;W = W + 1000 0001
                  ;W = 0000 0011 (+3, INCORRECT)
                       ;D7 = N = 0, OV = 1
```

C = 1 because there is a carry from D7 out but no carry from D6 to D7.

From the above discussion we conclude that while Carry is important in any addition, OV is extremely important in signed number addition because it is used to indicate whether or not the result is valid. As we will see in instruction "DAW", the DC flag is used in the addition of BCD numbers.

ADDWF **Add WREG and f**

Function: ADD WREG and fileReg
Syntax: ADDWF f,d,a

This adds the fileReg value to the WREG register, and places the result in WREG (if d = 0) or fileReg (if d = 1).

The ADDWF instruction is used for both signed and unsigned numbers. (See ADDLW instruction.)

Example:

```
MyReg  SET  0x20    ;loc 20H for MyReg
MOVLW 0x45          ;WREG = 45H
MOVWF MyReg         ;MyReg = 45H
MOVLW 0x4F          ;WREG = 4FH
ADDWF MyReg    ;WREG = 94H (45H + 4FH = 94H)
                    ;C = 0
```

We can place the result in fileReg, as shown in the following example:

```
MyReg  SET  0x20    ;loc 20H for MyReg
MOVLW 0x45          ;WREG = 45H
MOVWF MyReg         ;MyReg = 45H
MOVLW 0x4F          ;WREG = 4FH
ADDWF MyReg,F       ;MyReg = 94H
               ;(45H + 4FH = 94H), C = 0
```

For cases of a = 0 and a = 1, see Section A.1 in this chapter.

ADDWFC **Add WREG and Carry flag to fileReg**

Function: ADD WREG and Carry bit to fileReg
Syntax: ADDWFC f,d,a

This will add WREG and the C flag to fileReg (Destination = WREG + fileReg + C). If C = 1 prior to this instruction, 1 is also added to destination. If C = 0 prior to the instruction, source is added to destination plus 0. This instruction is used in multibyte additions. In the addition of 25F2H to 3189H, for example, we use the ADDWFC instruction as shown below.

Example when d = 0:

Assume we have the following data in RAM locations 0x10 and 0x11

0x10 = (F2)
0x11 = (25)

```
Reg_L     SET  0x10 ;loc 0x10 for Reg_L
Reg_H     SET  0x11 ;loc 0x11 for Reg_H
BCF  STATUS,C       ;make carry = 0
MOVLW 89H           ;WREG = 89H
ADDWFC Reg_L,1 ;Reg_L = 89H + F2H + 0 = 7BH
```

```
                    ;and C = 1
MOVLW 0x31          ;WREG = 31H
ADDWFC Reg_2,1      ;Reg_H = 31H + 25H + 1 = 57H
```

Therefore the result is:

```
  25F2H
 +3189H
  577BH
```

ANDLW **AND Literal byte with WREG**

Function: Logical AND literal value k with WREG

Syntax: ANDLW k

This performs a logical AND on the WREG and the Literal byte operand, bit by bit, storing the result in the WREG.

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

Example:

```
MOVLW  0x39   ;W = 39H
ANDLW  0x09   ;W = 39H ANDed with 09

39H  0011 1001
09H  0000 1001
09H  0000 1001
```

Example:

```
MOVLW 32H  ;W = 32H       32H  0011 0010
ANDLW 50H  ;AND W with    50H  0101 0000
           ;(W = 10H)     10H  0001 0000
```

ANDWF **AND WREG with fileReg**

Function: Logical AND for byte variables

Syntax: ANDWF f,d,a

This performs a logical AND on the fileReg value and the WREG register, bit by bit, and places the result in WREG (if d = 0) or fileReg (if d = 1).

Example:

```
MyReg  SET 0x40;set MyReg loc at 0x40
MOVLW  0x39     ;W = 39H
MOVWF  MyReg    ;MyReg = 39H
MOVLW  0x09
ANDWF  MyReg    ;39H ANDed with 09 (W = 09)

39H  0011 1001
09H  0000 1001
09H  0000 1001
```

Example:

```
MyReg  SET 0x40;set MyReg loc at 0x40
MOVLW 0x32     ;W = 32H
MOVWF MyReg    ;MyReg = 32H
MOVLW 0x0F     ;WREG = 0FH
ANDLW MyReg    ;32H ANDed with 0FH (W = 02)

32H  0011 0010
0FH  0000 1111
02H  0000 0010
```

We can place the result in fileReg as shown in the examples below:

```
MyReg  SET 0x40;set MyReg loc at 0x40
MOVLW 0x32     ;W = 32H
MOVWF MyReg    ;MyReg = 32H
MOVLW 0x50     ;WREG = 50H
ANDLW MyReg,F  ;MyReg = 09, WREG = 50H
```

The instructions below clear (mask) certain bits of the output ports, assuming the ports are configured as output ports:

```
MOVLW 0xFE
ANDWF PORTB,F  ;mask PORTB.0 (D0 of Port B)
MOVLW 0x7F
ANDWF PORTC,F  ;mask PORTC.7 (D7 of Port C)
MOVLW 0xF7
ANDWF PORTD,F  ;mask PORTD.3 (D3 of Port D)
```

Branch Condition

Function: Conditional Branch (jump)

In this type of Branch (jump), control is transferred to a target address if certain conditions are met. The following is list of branch instructions dealing with the flags:

BC	Branch if carry	jump if C = 1
BNC	Branch if no carry	jump if C = 0
BZ	Branch if zero	jump if Z = 1
BNZ	Branch if no zero	jump if Z = 0
BN	Branch if negative	jump if N = 1
BNN	Branch if no negative	jump if N = 0
BOV	Branch if overflow	jump if OV = 1
BNOV	Branch if no overflow	jump if OV = 0

Notice that all "Branch condition" instructions are short jumps, meaning that the target address cannot be more than –128 bytes backward or +127 bytes forward of the PC of the instruction following the jump. In other words, the target address cannot be more than –128 to +127 bytes away from the current PC. What

happens if a programmer needs to use a “Branch condition” to go to a target address beyond the –128 to +127 range? The solution is to use the “Branch condition” along with the unconditional GOTO instruction, as shown below.

```
          ORG   0x100
          MOVLW 0x87      ;WREG = 87H
          ADDLW 0x95      ;C = 1 after addition
          BNC   NEXT      ;branch if C = 0
          GOTO  OVER      ;target more than 128 bytes away
NEXT:     ...
          ...
          ...
          ORG 0x5000
OVER:     MOVWF PORTD
```

BC **Branch if C = 1**

Function: Branch if Carry flag bit = 1
Syntax: BC target_address

This instruction branches if C = 1.

Example:

```
     MOLW 0x0       ;WREG = 0
BACK ADDLW 0x1      ;add 1 to WREG
     BC   EXIT      ;exit if C = 1
     BRA  BACK      ;keep doing it
EXIT .....
     .....
```

Notice that this is a 2-byte instruction; therefore, the target address cannot be more than –128 to +127 bytes away from the program counter. See Branch Condition for further discussion on this issue.

BCF **Bit Clear fileReg**

Function: Clear bit of a fileReg
Syntax: BCF f,b,a

This instruction clears a single bit of a given file register. The bit can be the directly addressable bit of a port, register, or RAM location. Here are some examples of its format:

```
BCF  STATUS,C  ;C = 0
BCF  PORTB,5   ;CLEAR PORTB.5 (PORTB.5 = 0)
BCF  PORTC,7   ;CLEAR PORTC.7 (PORTC.7 = 0)
BCF  MyReg,1   ;CLEAR D1 OF File Register MyFile
```

BN **Branch if N = 1**

Function: Jump if Negative flag bit = 1
Syntax: BN target_address

This instruction branches if N = 1. It is used in signed number addition. See ADDLW instruction. Notice that this is a 2-byte instruction; therefore, the target address cannot be more than –128 to +127 bytes away from the program counter. See Branch Condition for further discussion on this issue.

BNC **Branch if no Carry**

Function: Branch if Carry flag is 0
Syntax: BNC target_address

This instruction examines the C flag, and if it is zero it will jump (branch) to the target address.

Example: Find the total sum of the bytes F6H, 98H, and 8AH. Save the carries in register C_Reg.

```
     C_Reg SET 0x20 ;set aside loc 0x20 for carries

          MOVLW 0x0       ;W = 0
          MOVWF C_Reg     ;C_Reg = 0
          ADDLW 0xF6
          BNC   OVER1
          INCF C_Reg,F
OVER1:    ADDLW 0x98
          BNC  OVER2
          INCF  C_Reg,F
OVER2:    ADDWF 0x8A
          BNC  OVER3
          INCF C_Reg
OVER3:
```

Notice that this is a 2-byte instruction; therefore, the target address cannot be more than –128 to +127 bytes away from the program counter. See Branch Condition for further discussion on this.

BNN **Branch if Not Negative**

Function: Branch if Negative flag bit = 0
Syntax: BNN target_address

This instruction branches if N = 0. It is used in signed number addition. See ADDLW instruction. Notice that this is a 2-byte instruction; therefore, the target address cannot be more than –128 to +127 bytes away from the program counter. See Branch Condition for further discussion on this issue.

BNOV **Branch if No Overflow**

Function: Jump if overflow flag bit = 0
Syntax: BNOV target_address

This instruction branches if OV = 0. It is used in signed number addition. See ADDLW instruction. Notice that this is a 2-byte instruction; therefore, the target address cannot be more than −128 to +127 bytes away from the program counter. See Branch Condition for further discussion on this issue.

BNZ **Branch if No Zero**

Function: Jump if Zero flag is 0
Syntax: BNZ target_address

This instruction branches if Z = 0.

Example:

```
     CLRF TRISB     ;PORTB as output
     CLRF PORTB     ;clear PORTB
OVER INCF PORTB,F   ;INC PORTB
     BNZ  OVER      ;do it until it becomes zero
```

Example: Add value 7 to WREG five times.

```
     COUNTER   SET  0x20 ;loc 20H for COUNTER
     MOVLW 0x5      ;WREG = 5
     MOVWF COUNTER  ;COUNTER = 05
     MOVLW 0x0      ;WREG = 0
OVER ADDLW 0x7      ;add 7 to WREG
     DECF  COUNTER,F ;decrement counter
     BNZ  OVER      ;do it until counter is zero
```

Notice that this is a 2-byte instruction; therefore, the target address cannot be more than −128 to +127 bytes away from the program counter. See Branch Condition for further discussion on this issue.

BOV **Branch if Overflow**

Function: Jump if Overflow flag = 1
Syntax: BOV target_address

This instruction jumps if OV = 1. It is used in signed number addition. See ADDLW instruction. Notice that this is a 2-byte instruction; therefore, the target address cannot be more than −128 to +127 bytes away from the program counter. See Branch Condition for further discussion on this issue.

BRA Branch unconditional

Function: Branch unconditionally
Syntax: BRA target_address

BRA stands for "Branch." It transfers program execution to the target address unconditionally. The target address for this instruction must be within 1K of program memory. This is a 2-byte instruction. The first 5 bits is the opcode and the rest is the signed number displacement, which is added to the PC (program counter) of the instruction following the BRA to get the target address. Therefore, in this branch, the target address must be within −1024 to +1023 bytes of the PC (program counter) of the instruction after the BRA because the 11-bit address can take values of +1024 to −1023. This address is often referred to as a *relative address* because the target address is −1024 to +1023 bytes relative to the program counter (PC).

BSF Bit Set fileReg

Function: Set bit
Syntax: BSF f, b, a

This sets HIGH the indicated bit of a file register. The bit can be any directly addressable bit of a port, register, or RAM location.

Examples:

```
BSF  PORTB,3   ;make PORTB.3 = 1
BSF  PORTC,6   ;make PORTC.6 = 1
BSF  MyReg,2   ;make bit D2 of MyReg = 1
BSF  STATUS,C  ;set Carry Flag C = 1
```

BTFSC Bit Test fileReg, Skip if Clear

Function: Skip the next instruction if bit is 0
Syntax: BTFSC f, b,a

This instruction is used to test a given bit and skip the next instruction if the bit is low. The given bit can be any of the bit-addressable bits of RAM, ports, or registers of the PIC18.

Example: Monitor the PORTB.5 bit continuously and, when it becomes low, put 55H in WREG.

```
          BSF TRISB,5    ;make PORTB.5 an input bit
HERE      BTFSC PORTB,5  ;skip if PORTB.5 = 0
          BRA   HERE
          MOVLW 0x55     ;because PORTB.5 = 0,
                         ;put 55H in WREG
```

Example: See if WREG has an even number. If so, make it odd.

```
        BTFSC WREG,0   ;skip if it is odd
        BRA  NEXT
        ADDLW 0x1      ;it is even, make it odd
NEXT:   ...
```

BTFSS **Bit Test fileReg, Skip if Set**

Function: Skip the next instruction if bit is 1
Syntax: BTFSS f, b, a

This instruction is used to test a given bit and skip the next instruction if the bit is HIGH. The given bit can be any of the bit-addressable bits of RAM, ports, or registers of the PIC18.

Example: Monitor the PORTB.5 bit continuously and when it becomes HIGH, put 55H in WREG.

```
        BSF TRISB,5    ;make PORTB.5 an input bit
HERE    BTFSS PORTB,5  ;skip if PORTB.5 = 1
        BRA   HERE
        MOVLW 55H   ;because PORTB.5 = 0 WREG = 55H
```

Example: See if WREG has an odd number. If so, make it even.

```
        BTFSS WREG,0        ;skip if it is even
        BRA  NEXT
        ADDLW 0x01          ;it is even, make it odd
NEXT:   ...
```

BTG **Bit Toggle fileReg**

Function: Toggle (Complement) bit
Syntax: BTG f, b, a

This instruction complements a single bit. The bit can be any bit-addressable location in the PIC18.

Example:

```
        BCF TRISB,0    ;make PORTB.0 an output
AGAIN   BTG PORTB,0    ;complement PORTB.0 bit
        BRA AGAIN      ;continuously forever
```

Example: Toggle PORTB.7 a total of 150 times.

```
    COUNTER   SET  0x20 ;loc 20H for COUNTER
        MOVLW 'D' 150   ;WREG = 150
        MOVWF COUNTER  ;COUNTER = 150
        BCF   TRISB,7  ;make PORTB.7 an output
```

```
OVER BTG   PORTB.7   ;toggle PORTB.7
     DECF COUNTER,F ;decrement and put it in
                    ;COUNTER
     BNZ  OVER      ;do it 150 times
```

BZ **Branch if Zero**

Function: Branch if Z = 1
Syntax: BZ target_address

Example: Keep checking PORTB for value 99H.

```
     SETF TRISB     ;port B as input
BACK MOVFW PORTB    ;get PORTB into WREG
     SUBLW 0x99     ;subtract 99H from it
     BZ   EXIT      ;if 0x99, exit
     BRA  BACK      ;keep checking
     ...
EXIT:     ...
```

Example: Toggle PORTB 150 times.

```
MyReg     SET  0x40 ;loc 40H for MyReg
     SETF TRISB     ;port B as output
     MOVLW D'150'   ;WREG = 150
     MOVWF MyReg
BACK COMF PORTB     ;toggle PORTB
     DECF MyReg,F   ;decrement MyReg
     BZ   EXIT      ;if MyReg = 0, exit
     BRA  BACK      ;keep toggling
     ...
EXIT:     ...
```

Notice that this is a 2-byte instruction; therefore, the target address cannot be more than –128 to +127 bytes away from the program counter. See Branch Condition for further discussion on this.

CALL

Function: Transfers control to a subroutine
Syntax: CALL k,s ;s is used for fast context switching

The Call intruction is a 4-byte instruction. The first 12 bits are used for the opcode and the rest (20 bits) are set aside for the address. A 20-bit address allows us to reach the target address anywhere in the 2M ROM space of the PIC18. If calling a subroutine, the PC register (which has the address of the instruction after the CALL) is pushed onto the stack and the stack pointer (SP) is incremented by 1. Then the program counter is loaded with the new address and control is transferred to the subroutine. At the end of the procedure, when RETURN is executed, PC is popped off the stack, which returns control to the instruction after the CALL.

Notice that CALL is a 4-byte instruction, in which 12 bits are the opcode, and the other 20 bits are the 20-bit address of an even address location. Because

all the PIC18 instructions are 2 bytes in size, the lowest address bit, A0, is automatically set to zero to make sure that the CALL instruction will not land at the middle of the targeted instruction. The 20-bit address of the CALL provides the A20–A1 part of the address and with the A0 = 0, we have the 21-bit address needed to go anywhere in the 2M address space of the PIC18.

We have two options for the "CALL k,s" instruction. They are s = 0, and s = 1. When s = 0, it is simply calling a subroutine. With s = 1, we are calling a subroutine and we are also asking the CPU to save the three major registers of WREG, STATUS, and BSR in internal buffers (shadow registers) for the purpose of context-switching. This fast context-switching can be used only in the main subroutine because the depth of the shadow registers is only one. That means no nested call with the s = 1. Look at the following case:

```
            ORG   0x0
MAIN        .....
            .....
            .....
            CALL  M_SUB,1       ;call and save the registers
            MOVLW 0x55 ;address of this instruction is saved on stack
            ....

;------------------------
            ORG   0x2000
M_SUB       ......
            .....
            CALL  Y_SUB         ;we cannot use CALL Y_SUB,1
            MOVLW 0xAA;address of this instruction is saved on stack
            .....
            .....
            RETURN,1     ;return to caller and restore the registers
                         ;notice the s = 1 for RETURN

;----------------------------------
            ORG 0x3000
Y_SUB
            .....
            .....
            RETURN

;------------------------------
            END
```

As shown in RETURN instruction, we also have two options for the RETURN: s = 0 and s = 1. If we use s = 1 for the CALL, we must also use s = 1 for the RETURN. Notice that "CALL Target" with no number after it is interpreted as s = 0 by the assembler. Likewise, the "RETURN" with no number after it is interpreted as s = 0 by the assembler.

CLRF **Clear fileReg**

Function: Clear
Syntax: CLRF f, a

This instruction clears the entire byte in the fileReg. All bits of the register are cleared to 0.

Example:

```
MyReg      SET  0x20 ;loc 20H for MyReg
     CLRF MyReg     ;clear MyReg
     CLRF TRISB ;clear TRISB (make PORTB output)
     CLRF PORTB     ;clear PORTB
     CLRF TMR01L    ;TMR0L = 0
```

Notice that in this instruction the result can be placed in fileReg only and there is no option for the WREG to be used as the destination.

CLRWDT

Function: Clear Watchdog Timer
Syntax: CLRWDT

This instruction clears the Watchdog Timer.

COMF **Complement the fileReg**

Function: Complement a fileReg
Syntax: COMF f, d, a

This complements the contents of a given fileReg. The result is the 1's complement of the register; that is, 0s become 1s and 1s become 0s. The result can be placed in WREG (if d = 0) or fileReg (if d = 1).

Example:

```
          MOVLW 0x0      ;WREG = 0
          MOVWF TRISB    ;Make PORTB an output port
          MOVLW 0x55     ;WREG = 01010101
          MOVWF PORTB
AGAIN     COMF PORTB,F   ;complement (toggle) PORTB
          CALL DELAY
          BRA  AGAIN ;continuously (notice WREG = 55H)
```

Example:

```
          MyReg  SET 0x40;set MyReg loc at 0x40
          MOVLW  0x39    ;W = 39H
          MOVWF  MyReg   ;MyReg = 39H
          COMPF  MyReg,F ;MyReg = C6H and WREG = 39H
```

Where 39H (0011 1001 bin) becomes C6H (1100 0110).

Example:

```
        MyReg  SET 0x40;set MyReg loc at 0x40
        MOVLW 0x55     ;W = 55H
        MOVWF MyReg    ;MyReg = 55H
        COMPF MyReg,F  ;MyReg AAH, WREG = 55H
```

where 55H (0101 0101) becomes AAH (1010 1010).

Example: Toggle PORTB 150 times.

```
COUNTER   SET  0x40 ;loc 40H for COUNTER
     SETF TRISB     ;port B as output
     MOVLW D'150'   ;WREG = 150
     MOVWF COUNTER  ;COUNTER = 150
     MOVLW 0x55     ;WREG = 55H
     MOVWF PORTB
BACK COMF PORTB,F   ;toggle PORTB
     DECF COUNTER,F ;decrement COUNTER
     BNZ  BACK ;toggle until counter becomes 0
```

We can place the result in WREG as shown in the examples below:

```
MyReg  SET 0x40     ;set MyReg loc at 0x40
     MOVLW  0x39    ;W = 39H
     MOVWF  MyReg   ;MyReg = 39H
     COMPF  MyReg   ;MyReg = 39H and WREG = C6H
```

Example:

```
MyReg  SET    0x40 ;set MyReg loc at 0x40
     MOVLW 0x55     ;W = 55H
     MOVWF MyReg    ;MyReg = 55H
     COMPF MyReg    ;WREG = AA and MyReg 55H SETF
```

CPFSEQ   Compare FileReg with WREG and skip if equal (F = W)

Function: Compare fileReg and WREG and skip if they are equal
Syntax: CPFSEQ f, a

The magnitudes of the fileReg byte and WREG byte are compared. If they are equal, it skips the next instruction.

Example: Keep monitoring PORTB indefinitely for the value of 99H. Get out only when PORTB has the value 99H.

```
     SETF  TRISB    ;PORTB an input port
     MOVLW 0x99     ;WREG = 99h
BACK CPFSEQ PORTB   ;skip if PORTB has 0x99
     BRA BACK       ;keep monitoring
```

Notice that CPFSEQ skips only when fileReg and WREG have equal values.

CPFSGT Compare FileReg with WREG and skip if greater (F > W)

Function: Compare fileReg and WREG and skip if fileReg > WREG.
Syntax: CPFSGT f, a

The magnitudes of the fileReg byte and WREG byte are compared. If fileReg is larger than the WREG, it skips the next instruction.

Example: Keep monitoring PORTB indefinitely for the value of 99H. Get out only when PORTB has a value greater than 99H.

```
          SETF   TRISB    ;PORTB an input port
          MOVLW  0x99     ;WREG = 99H
BACK      CPFSGT PORTB    ;skip if PORTB > 99H
          BRA BACK        ;keep monitoring
```

Notice that CPFSGT skips only if FileReg is greater than WREG.

CPFSLT Compare FileReg with WREG and skip if less than (F < W)

Function: Compare fileReg and WREG and skip if fileReg < WREG.
Syntax: CPFSLT f, a

The magnitudes of the fileReg byte and WREG byte are compared. If fileReg is less than the WREG, it skips the next instruction.

Example: Keep monitoring PORTB indefinitely for the value of 99H. Get out only when PORTB has a value less than 99H.

```
          SETF   TRISB    ;PORTB an input port
          MOVLW  0x99     ;WREG = 99H
BACK:     CPFSEQ PORTB    ;skip if PORTB < 99H
          BRA BACK        ;keep monitoring
```

Notice that CPFSLT skips only if FileReg < WREG.

DAW

Function: Decimal-adjust WREG after addition
Syntax: DAW

This instruction is used after addition of BCD numbers to convert the result back to BCD. The data is adjusted in the following two possible cases:

1. It adds 6 to the lower 4 bits of WREG if it is greater than 9 or if DC = 1.
2. It also adds 6 to the upper 4 bits of WREG if it is greater than 9 or if C = 1.

Example:

```
        MOVLW  0x47       ;WREG = 0100 0111
        ADDLW  0x38       ;WREG = 47H + 38H = 7FH,
                          ;invalid BCD
        DAW        ;WREG = 1000 0101 = 85H, valid BCD

         47H
       + 38H
         7FH   (invalid BCD)
       +  6H   (after DAW)
         85H   (valid BCD)
```

In the above example, because the lower nibble was greater than 9, DAW added 6 to WREG. If the lower nibble is less than 9 but DC = 1, it also adds 6 to the lower nibble. See the following example:

```
        MOVLW  0x29       ;WREG = 0010 1001
        ADDLW  0x18       ;WREG = 0100 0001 INCORRECT
        DAW         ;WREG = 0100 0111 = 47H VALID BCD

         29H
       + 18H
         41H   (incorrect result in BCD)
       +  6H
         47H   correct result in BCD
```

The same thing can happen for the upper nibble. See the following example:

```
        MOVLW  0x52       ;WREG = 0101 0010
        ADDLW  0x91       ;WREG = 1110 0011 INVALID BCD
        DAW               ;WREG = 0100 0011 AND C = 1

         52H
       + 91H
         E3H        (invalid BCD)
       + 6          (after DAW, adding to upper nibble)
        143H        valid BCD
```

Similarly, if the upper nibble is less than 9 and C = 1, it must be corrected. See the following example:

```
        MOVLW  0x94             ;W = 1001 0100
        ADDLW  0x91             ;W = 0010 0101 INCORRECT
        DAW               ;W = 1000 0101, VALID BCD
                                ;FOR 85, C = 1
```

```
   94H
 + 91H
 1 25H        (incorrect BCD)
 + 6          (after DAW, adding to upper nibble)
 1 8 5
```

It is possible that 6 is added to both the high and low nibbles. See the following example:

```
MOVLW   0x54    ;WREG = 0101 0100
ADDLW   0x87    ;WREG = 1101 1011 INVALID BCD
DAW        ;WREG = 0100 0001, C = 1 (BCD 141)

   54H
 + 87H
   DBH  (invalid result in BCD)
 + 6 6H
1 4 1H       valid BCD
```

DECF **Decrement fileReg**

Function: Decrement fileReg
Syntax: DECF f, d, a

This instruction subtracts 1 from the byte operand in fileReg. The result can be placed in WREG (if d = 0) or fileReg (if d = 1).

Example:

```
MyReg SET 0x40 ;set aside loc 40H for MyReg
MOVLW 0x99     ;WREG = 99H
MOVWF MyReg    ;MyReg = 99H
DECF MyReg,F   ;MyReg = 98H, WREG 99H
DECF MyReg,F   ;MyReg = 97H, WREG 99H
DECF MyReg,F   ;MyReg = 96H, WREG 99H
```

Example: Toggle PORTB 250 times.

```
COUNTER   SET  0x40 ;loc 40H for COUNTER
     SETF TRISB     ;PORTB as output
     MOVLW D'250'   ;WREG = 250
     MOVWF COUNTER  ;COUNTER = 250
     MOVLW 0x55     ;WREG = 55H
     MOVWF PORTB
BACK COMF PORTB,F   ;toggle PORTB
     DECF COUNTER,F ;decrement COUNTER
     BNZ  BACK ;toggle until counter becomes 0
```

We can place the result in WREG as shown in the examples below:

```
MyReg SET 0x40 ;set aside loc for MyReg
MOVLW 0x99     ;WREG = 99H
MOVWF MyReg    ;MyReg = 99H
DECF MyReg     ;WREG = 98H, MyReg = 99H
DECF MyReg     ;WREG = 97H, MyReg = 99H
DECF MyReg     ;WREG = 96H, MyReg = 99H
```

Example:

```
MyReg  SET     0x50 ;set MyReg loc at 0x50
MOVLW  0x39    ;W = 39H
MOVWF  MyReg   ;MyReg = 39H
DECF   MyReg   ;WREG = 38H and MyReg = 39H
DECF   MyReg   ;WREG = 37H and MyReg = 39H
DECF   MyReg   ;WREG = 36H and MyReg = 39H
DECF   MyReg   ;WREG = 35H and MyReg = 39H
```

DECFSZ — Decrement fileReg and Skip if zero

Function: Decrement fileReg and skip if fileReg has zero in it
Syntax: DECFSZ f, d, a

This instruction subtracts 1 from the byte operand of fileReg. If the result is zero, then it skips execution of the next instruction.

Example: Toggle PORTB 250 times.

```
COUNT     SET  0x40 ;loc 40H for COUNT
     CLRF TRISB     ;PORTB an output
     MOVLW D'250'   ;WREG = 250
     MOVWF COUNT    ;COUNT = 250
     MOVLW 0x55     ;WREG = 55H
     MOVWF PORTB
BACK COMF PORTB,F   ;toggle PORTB
     DECFSZ COUNT,F ;decrement COUNT and
                    ;skip if zero
     BRA  BACK ;toggle until counter becomes 0
     ....
```

DECFSNZ — Decrement fileReg and skip if not zero

Function: Decrement fileReg and skip if fileReg has other than zero
Syntax: DECFSNZ f, d, a

This instruction subtracts 1 from the byte operand of fileReg. If the result is not zero, then it skips execution of the next instruction.

Example: Toggle PORTB 250 times continuously.

```
COUNT     SET  0x40 ;loc 40H for COUNT
     CLRF TRISB     ;PORTB an output
OVER MOVLW D'250'   ;WREG = 250
     MOVWF COUNT    ;COUNT = 250
     MOVLW 0x55     ;WREG = 55H
     MOVWF PORTB
BACK COMF PORTB,F   ;toggle PORTB
     DECFSNZ COUNT,F ;decrement COUNT and
                    ;skip if zero
     BRA  OVER      ;start over
     BRA  BACK ;toggle until counter becomes 0
```

GOTO **Unconditional Branch**

Function: Transfers control unconditionally to a new address.
Syntax: GOTO k

In the PIC18 there are two unconditional branches (jumps): GOTO (long jump) and BRA (short jump). Each is described next.

1. GOTO (long jump): This is a 4-byte instruction. The first 12 bits are the opcode, and the next 20 bits are an even address of the target location. Because all the PIC18 instructions are 2 bytes in size, the lowest address bit, A0, is automatically set to zero to make sure that the GOTO instruction will not land at the middle of the targeted instruction. The 20-bit address of the GOTO provides the A20–A1 part of the address and with A0 = 0, we have the 21-bit address needed to go anywhere in the 2M address space of the PIC18.
2. BRA: This is a 2-byte instruction. The first 5 bits are the opcode and the remaining 11 bits are the signed number displacement, which is added to the PC (program counter) of the instruction following the BRA to get the target address. Therefore, for the BRA instruction the target address must be within −1023 to +1024 bytes of the PC of the instruction after the BRA because a 11-bit address can take values of +1023 to −1024.

While GOTO is used to jump to any address location within the 2M code space of the PIC18, BRA is used to jump to a location within the 1K ROM space. The advantage of BRA is the fact that it takes 2 bytes of program ROM, while GOTO takes 4 bytes. BRA is widely used in chips with a small amount of program ROM and a limited number of pins.

Notice that the difference between GOTO and CALL is that the CALL instruction will return and continue execution with the instruction following the CALL, whereas GOTO will not return.

INCF **Increment fileReg**

Function: Increment
Syntax: INCF f, d, a

This instruction adds 1 to the byte operand in fileReg. The result can be placed in WREG (if d = 0) or fileReg (if d = 1).

Example:

```
     MyReg SET 0x40 ;set aside loc 40H for MyReg
     MOVLW 0x99     ;WREG = 99H
     MOVWF MyReg
     INCF MyReg,F   ;MyReg = 9AH, WREG 99H
     INCF MyReg,F   ;MyReg = 9BH, WREG 99H
     DECF MyReg,F   ;MyReg = 9CH, WREG 99H
```

Example: Toggle PORTB 5 times.

```
COUNTER   SET  0x40 ;loc 40H for COUNTER
     SETF TRISB     ;PORTB as output
     MOVLW D'251'   ;WREG = 251
     MOVWF COUNTER  ;COUNTER = 251
     MOVLW 0x55     ;WREG = 55H
     MOVWF PORTB
BACK COMF PORTB,F   ;toggle PORTB
     INCF COUNTER,F ;INC COUNTER
     BNC  BACK ;toggle until counter becomes 0
```

We can place the result in fileReg as shown in the examples below:

```
     MyReg SET 0x40 ;set aside loc for MyReg
     MOVLW 0x99     ;WREG = 99H
     MOVWF MyReg    ;MyReg = 99H
     INCF MyReg     ;WREG = 9AH, MyReg = 99H
     INCF MyReg     ;WREG = 9BH, MyReg = 99H
```

Example:

```
     MyReg  SET     0x40 ;set MyReg loc at 0x40
     MOVLW  0x5     ;W = 05H
     MOVWF  MyReg   ;MyReg = 05H
     INCF   MyReg   ;WREG = 06H and MyReg = 05H
```

INCFSZ **Increment fileReg and skip if zero**

Function: Increment
Syntax: INCFSZ f, d, a

This instruction adds 1 to fileReg and if the result is zero it skips the next instruction.

Example: Toggle PORTB 156 times.

```
COUNTER   SET  0x40 ;loc 40H for COUNTER
     SETF TRISB     ;PORTB as output
     MOVLW D'156'   ;WREG = 156
     MOVWF COUNTER  ;COUNTER = 156
     MOVLW 0x55     ;WREG = 55H
     MOVWF PORTB
BACK COMF PORTB,F   ;toggle PORTB
     INCFSZ COUNTER,F ;INC COUNTER and skip if 0
     BRA  BACK ;toggle until counter becomes 0
     .....
```

INCFSNZ Increment fileReg and skip if not zero

Function: Increment
Syntax: INFSNZ f, d, a

This instruction adds 1 to the register or memory location specified by the operand. If the result is not zero, it skips the next instruction.

Example: Toggle PORTB 156 times continuously.

```
COUNTER   SET  0x40 ;loc 40H for COUNTER
     SETF TRISB     ;PORTB as output
OVER MOVLW D'156'   ;WREG = 156
     MOVWF COUNTER  ;COUNTER = 156
     MOVLW 0x55     ;WREG = 55H
     MOVWF PORTB
BACK COMF PORTB,F   ;toggle PORTB
     INCFSNZ COUNTER,F;INC COUNTER, skip if not 0
     BRA  OVER ;start over
     BRA  BACK ;toggle until counter becomes 0
```

IORLW OR K value with WREG

Function: Logical-OR WREG with value k
Syntax: IORLW k

This performs a logical OR on the WREG register and k value, bit by bit, and stores the result in WREG.

Example:

```
     MOVLW 0x30  ;W = 30H
     IORLW 0x09  ;now W = 39H
```

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

```
39H   0011 0000
09H   0000 1001
39    0011 1001
```

Example:

```
MOVLW  0x32      ;W = 32H
IORLW  0x50      ;(W = 72H)

32H   0011 0010
50H   0101 0000
72H   0111 0010
```

IORWF **OR FileReg with WREG**

Function: Logical-OR fileReg and WREG
Syntax: IORWF f, d, a

This performs a logical OR on the fileReg value and the WREG register, bit by bit, and places the result in WREG (if d = 0) or fileReg (if d = 1).

Example:

```
MyReg  SET 0x40;set MyReg loc at 0x40
MOVLW  0x39    ;WREG = 39H
MOVWF  MyReg   ;MyReg = 39H
MOVLW  0x07
IORWF  MyReg   ;39H ORed with 07 (W = 3F)

39   0011 1001
07   0000 0111
3F   0011 1111
```

Example:

```
MyReg  SET 0x40;set MyReg loc at 0x40
MOVLW  0x5     ;WREG = 05H
MOVWF  MyReg   ;MyReg = 05H
MOVLW  0x30
IORWF  MyReg   ;30H ORed with 05 (W = 35H)

05   0000 0101
30   0011 0000
35   0011 0101
```

We can place the result in fileReg as shown in the examples below:

```
MOVLW 0x30      ;W = 30H
IORWF PORTB,F   ;W and PORTB are ORed and result
                ;goes to PORTB
```

Example:

```
MyReg     SET  0x20
MOVLW 0x54     ;WREG = 54H
MOVWF MyReg
MOVLW 0x67     ;WREG = 67H
IORWF MyReg,F  ;OR WREG and MyReg
;after the operation MyReg = 77H

44H  0101 0100
67H  0110 0111
77H  0111 0111
```

Therefore MyReg will have 77H, WREG = 54H.

LFSR **Load FSR**

Function: Load into FSR registers a 12-bit value of k
Syntax: LFSR f,k ;k is between 000 and FFFH

This loads a 12-bit value into one of the FSR registers of FSR0, FSR1, or FSR2.

```
LFSR 0 , 0x200 ;FSR0 = 200H
LFSR 1 , 0x050 ;FSR1 = 050H
LFSR 2 , 0x160 ;FSR2 = 160H
```

This is widely used in register indirect addressing mode. See Chapter 6.

MOVF (or MOVFW) **Move fileReg to WREG**

Function: Copy byte from fileReg to WREG
Syntax MOVF f, d, a:

This instruction is widely used for moving data from a fileReg to WREG. Look at the following examples:

```
CLRF      TRISC     ;PORTC output
SETF      TRISB     ;PORTB as input
MOVFW     PORTB     ;copy PORTB to WREG
ANDLW     0x0F      ;mask the upper 4 bits
MOVWF     PORTC     ;put it in PORTC
```

Example:

```
CLRF      TRISD     ;PORTD as output
SETF      TRISB     ;PORTB as input
MOVFW     PORTB     ;copy PORTB to WREG
IORW      0x30      ;OR it with 30H
MOVWF     PORTD     ;put it in PORTD
```

This instruction can be used to copy the fileReg to itself in order to get the status of the N and Z flags. Look at the following example.

Example:

```
MyReg SET 0x20 ;set aside loc 0x20 to MyReg
MOVLW 0x54     ;W = 54H
MOVWF MyReg    ;MyReg = 54H
MOVFW MyReg,F ;My Reg = 54, also N = 0 and Z = 0
```

MOVFF **Move FileReg to Filereg**

Function: Copy byte from one fileReg to another fileReg
Syntax: MOVFF fs, fd

This copies a byte from the source location to the destination. The source and destination locations can be any of the file register locations, SFRs, or ports.

```
MOVFF     PORTB,MyReg
MOVFF     PORTC,PORTD
MOVFF     RCREG,PORTC
MOVFF     Reg1,REG2
```

Notice that this a 4-byte instruction because the source and destination address each take 12 bits of the instruction. That means the 24 bits of the instruction are used for the source and destination addresses. The 12-bit address allows data to be moved from any source location to any destination location within the 4K RAM space of the PIC18.

MOVLB **Move Literal 4-bit value to lower 4-bit of the BSR**

Function: Move 4-bit value k to lower 4 bits of the BSR registers
Syntax: MOVLB k ;k is between 0 and 15 (0–F in hex)

We use this instruction to select a register bank other than the access bank. With this instruction we can load into the BSR (bank selector register) a 4-bit value representing one of 16 banks supported by the PIC18. That means the values between 0000 and 1111 (0–F in hex). For examples of the MOVLB instruction, see Chapter 6 and Section A.1 in this chapter.

MOVLW K **Move Literal to WREG**

Function: Move 8-bit value k to WREG
Syntax: MOVLW k ;k is between 0 and 255 (0–FF in hex)

Example:

```
MOVLW     0x55     ;WREG = 55H
MOVLW     0x0      ;clear WREG (WREG = 0)
MOVLW     0xC2     ;WREG = C2H
MOVLW     0x7F     ;WREG = 7FH
```

This instruction, along with the MOVWF, is widely used to load fixed values into any port, SFR, or fileReg location. See the next instruction to see how it is used.

MOVWF **Move WREG to a fileReg**

Function: Copy the WREG contents to a fileReg
Syntax: MOVWF f, a

This copies a byte from WREG to fileReg. This instruction is widely used along with the MOVLW instruction to load any of the fileReg locations, SFRs, or PORTs with a fixed value. See the following examples:

Example: Toggle PORTB.
```
MOVLW     0x55       ;WREG = 55H
MOVWF     PORTB
MOVLW     0xAA       ;WREG = AAH
MOVWF     PORTB
BRA       OVER       ;keep toggling the PORTB
```

Example: Load RAM location 20H with value 50H.
```
MyReg SET 0x20 ;set aside the loc 0x20 for MyReg
MOVLW     0x50
MOVWF     MyReg      ;MyReg = 50H (loc 20H has 50H)
```

Example: Initialize the Timer0 low and high registers.
```
MOVLW     0x05       ;WREG = 05H
MOVWF     TMR0H      ;TMR0H = 0x5
MOVLW     0x30       ;WREG = 30H
MOVWF     TMR0L      ;TMR0L = 0x30
```

MULLW **Multiply Literal with WREG**

Function: Multiply $k \times$ WREG
Syntax: MULLW k

This multiplies an unsigned byte k by an unsigned byte in register WREG and the 16-bit result is placed in registers PRODH and PRODL, where PRODL has the lower byte and PRODH has the higher byte.

Example:
```
     MOVLW 0x5       ;WREG = 5H
     MULLW 0x07      ;PRODL = 35 = 23H, PRODH = 00
```

Example:
```
     MOVLW 0x0A      ;WREG = 10
     MULLW 0x0F      ;PRODL = 10 x 15 = 150 = 96H
                     ;PRODH = 00
```

Example:
```
     MOVLW 0x25
     MULLW 0x78  ;PRODL = 58H, PRODH = 11H
     ;because 25H x 78H = 1158H
```

Example:

```
     MOVLW D'100'   ;WREG = 100
     MULLW D'200'   ;PRODL = 20H, PRODH = 4EH
               ;(100 x 200 = 20,000 = 4E20H)
```

MULWF **Multiply WREG with fileReg**

Function: Multiply WREG × fileReg and place the result in PRODH:PROFDL registers

Syntax: MULWF f, a

This multiplies an unsigned byte in WREG by an unsigned byte in the fileReg register and the result is placed in PRODL and PRODH, where PRODL has the lower byte and PRODH has the higher byte.

Example:

```
MyReg     SET  0x20 ;MyReg has location of 0x20
     MOVLW 0x5
     MOVWF MyReg    ;MyReg has 0x5
     MOVLW 0x7      ;WREG = 0x7
     MULWF MyReg    ;PRODL = 35 = 23H, PRODH = 00
```

Example:

```
     MOVLW 0x0A
     MOVWF MyReg    ;MyReg = 10
     MOVLW 0x0F     ;WREG = 15
     MULFW MyReg  ;PRODL = 150 = 96H, PRODH = 00
```

Example:

```
     MOVLW 0x25
     MOVWF MyReg    ;MyReg = 0x25
     MOVLW 0x78     ;WREG 78H
     MULWF Myreg    ;PRODL = 58H, PRODH = 11H
                    ;(25H x 78H = 1158H)
```

Example:

```
     MOVLW D'100'   ;WREG = 100
     MOVWF MyReg    ;MyReg = 100
     MOVLW D'200'   ;WREG = 200
     MULWF MyReg    ;PRODL = 20H, PRODH = 4EH
               ;(100 x 200 = 20,000 = 4E20H)
```

NEGF **Negate fileReg**

Function: No operation

Syntax: NEGF f, a

This performs 2's complement on the value stored in fileReg and places it back in fileReg.

Example:

```
MyReg     SET 0x30
MOVLW 0x98     ;WREG = 0x98
MOVWF MyReg    ;MyReg = 0x98
NEGF           ;2's complement fileReg

98H  10011000
     01100111       1's complement
   +        1
     01101000       Now FileReg = 68H
```

Example:

```
MyReg     SET 0x10
MOVLW 0x75     ;WREG = 0x75
MOVWF MyReg    ;MyReg = 0x75
NEGF           ;2's complement fileReg

75H  01110101
     10001010       1's complement
   +        1
     10001011       Now FileReg = 7AH
```

Notice that in this instruction we cannot place the result in the WREG register.

NOP No Operation

Function: No operation
Syntax: NOP

This performs no operation and execution continues with the next instruction. It is sometimes used for timing delays to waste clock cyles. This instruction only updates the PC (program counter) to point to the next instruction following NOP. In PIC18, this a 2-byte instruction.

POP POP Top of Stack

Function: Pop from the stack
Syntax: POP

This takes out the top of stack (TOS) pointed to by SP (stack pointer) and discards it. It also decrements SP by 1. After the operation, the top of the stack will be the value pushed onto the stack previously.

PUSH PUSH Top of the Stack

Function: Push the PC onto the stack
Syntax: PUSH

This copies the program counter (PC) onto the stack and increments SP by 1, which means the previous top of the stack is pushed down.

RCALL Relative Call

Function: Transfers control to a subroutine within 1K space
Syntax: RCALL target_address

There are two types of CALLs: RCALL and CALL. In RCALL, the target address is within 1K of the current PC (program counter). To reach the target address in the 2M ROM space of the PIC18, we must use CALL. In calling a subroutine, the PC register (which has the address of the instruction after the RCALL) is pushed onto the stack and the stack pointer (SP) is incremented by 1. Then the program counter is loaded with the new address and control is transferred to the subroutine. At the end of the procedure, when RETURN is executed, PC is popped off the stack, which returns control to the instruction after the RCALL.

Notice that RCALL is a 2-byte instruction, in which 5 bits are used for the opcode and the remaining 11 bits are used for the target subroutine address. An 11-bit address limits the range to –1024 to +1023. See the CALL instruction for discussion of the target address being anywhere in the 2M ROM space of the PIC18. Notice that RCALL is a 2-byte instruction while CALL is a 4-byte instruction. Also notice that the RCALL does not have the option of context saving, as CALL has.

RESET Reset (by software)

Function: Reset by software
Syntax: RESET

This instruction is used to reset the PIC18 by way of software. After execution of this instruction, all the registers and flags are forced to their reset condition. The reset condition is created by activating the hardware pin MCLR. In other words, the RESET instruction is the software version of the MCLR pin.

RETFIE Return from Interrupt Exit

Function: Return from interrupt
Syntax: RETFIE s

This is used at the end of an interrupt service routine (interrupt handler). The top of the stack is popped into the program counter and program execution continues at this new address. After popping the top of the stack into the program counter (PC), the stack pointer (SP) is decremented by 1.

Notice that while the RETURN instruction is used at the end of a subroutine associated with the CALL and RCALL instructions, RETFIE must be used for the interrupt service routines (ISRs).

RETLW **Return with Literal in WREG**

Function: The k value is placed in WREG and the top of the stack is the placed in PC (program counter)
Syntax: RETLW k

After execution of this instruction, the k value is loaded into WREG and the top of the stack is popped into the program counter (PC). After popping the top of the stack into the program counter, the stack pointer (SP) is decremented by 1. This instruction is used for the implementation of a look-up table. See Section 6.3 in Chapter 6.

RETURN **Return**

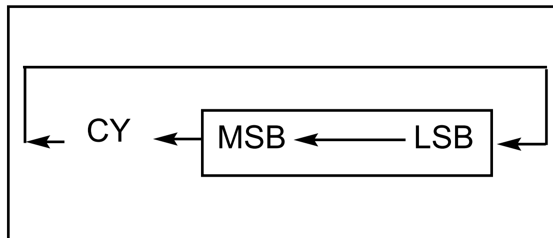
Function: Return from subroutine
Syntax: RETURN s ;where s = 0 or s = 1

This instruction is used to return from a subroutine previously entered by instructions CALL or RCALL. The top of the stack is popped into the program counter (PC) and program execution continues at this new address. After popping the top of the stack into the program counter, the stack pointer (SP) is decremented by 1. For the case of "RETURN s" where s = 1, the RETURN will also restore the context registers. See the CALL instruction for the case of s = 1. Notice that "RETURN 1" cannot be used for subroutines associated with RCALL.

RLCF **Rotate Left Through Carry the fileReg**

Function: Rotate fileReg left through carry
Syntax: RLCF f, d, a

This rotates the bits of a fileReg register left. The bits rotated out of fileReg are rotated into C, and the C bit is rotated into the opposite end of the fileReg register.



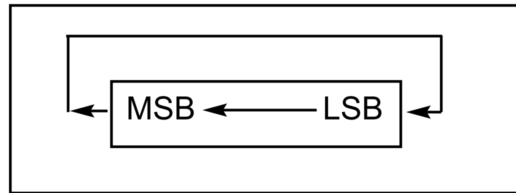
Example:

```
MyReg SET 0x30      ;set aside loc 30H for MyReg
      BCF  STATUS,C ;C = 0
      MOVLW 0x99    ;WREG = 99H
      MOVWF MyReg   ;MyReg = 99H = 10011001
      RLCF MyReg,F  ;now MyReg = 00110010 and
                    ;C = 1
      RLCF MyReg,F  ;now MyReg = 01100101 and
                    ;C = 0
```

RLNCF **Rotate left not through Carry**

Function: Rotate left the fileReg
Syntax: RLNCF f, d, a

This rotates the bits of a fileReg register left. The bits rotated out of fileReg are rotated back into fileReg at the opposite end.



Example:

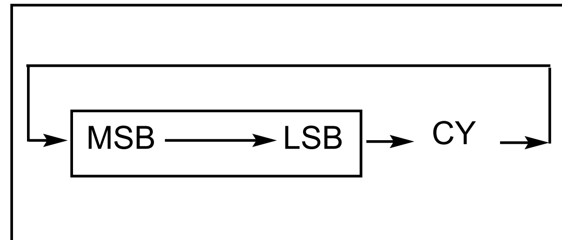
```
MyReg SET 0x20 ;set aside loc 20 for MyReg
MOVLW 0x69     ;WREG = 01101001
MOVWF MyReg    ;MyReg = 69H = 01101001
RLNCF MyReg,F  ;now MyReg = 11010010
RLNCF MyReg,F  ;now MyReg = 10100101
RLNCF MyReg,F  ;now MyReg = 01001011
RLNCF MyReg,F  ;now MyReg = 10010110
```

Notice that after four rotations, the upper and lower nibbles are swapped.

RRCF **Rotate Right through Carry**

Function: Rotate fileReg right through carry
Syntax: RRCF f, d, a

This rotates the bits of a fileReg register right. The bits rotated out of the register are rotated into C, and the C bit is rotated into the opposite end of the register.



Example:

```
MyReg SET 0x20 ;set aside loc 20 for MyReg
      BSF   STATUS,C  ;C = 1
      MOVLW 0x99      ;WREG = 10011001
      MOVWF MyReg     ;MyReg = 99H = 10011001
      RRCF  MyReg,F   ;now MyReg = 11001100, C = 1
      RRCF  MyReg,F   ;now MyReg = 11100110, C = 0
```

RRNCF **Rotate Right not through Carry**

Function: Rotate fileReg right
Syntax: RRNCF f, d, a

This rotates the bits of a fileReg register right. The bits rotated out of the register are rotated back into fileReg at the opposite end.

Example:

```
MyReg SET 0x20 ;set aside loc 20H for MyReg
     MOVLW 0x66     ;WREG = 66H = 01100110
     MOVWF MyReg    ;MyReg = 66H = 01100110
     RRNCF MyReg,F  ;now MyReg = 00110011
     RRNCF MyReg,F  ;now MyReg = 10011001
     RRNCF MyReg,F  ;now MyReg = 11001100
     RRNCF MyReg,F  ;now MyReg = 01100110
```

Example: We can use this instruction to swap the upper and lower nibbles.

```
MyReg SET 0x20 ;set aside loc 20H for MyReg
     MOVLW 0x36     ;WREG = 36H = 00110110
     MOVWF MyReg    ;MyReg = 36H = 00110110
     RRNCF MyReg,F  ;now MyReg = 00011011
     RRNCF MyReg,F  ;now MyReg = 10001101
     RRNCF MyReg,F  ;now MyReg = 11000110
     RRNCF MyReg,F  ;now MyReg = 01100011 = 63H
```

SETF **Set fileReg**

Function: Set
Syntax: SETF f, a

This instruction sets the entire byte in fileReg to HIGH. All bits of the register are set to 1.

Examples:

```
     SETF MyReg     ;MyReg = 11111111
     SETF TRISB ;TRISB = FFH,(makes PORTB input)
     SETF PORTC     ;PORTC = 1111 1111
```

Notice that in this instruction, the result can be placed in fileReg only and there is no option for WREG to be used as the destination for the result.

SLEEP **Enter Sleep mode**

Function: Put the CPU into sleep mode
Syntax: SLEEP

This instruction stops the oscillator and puts the CPU into sleep mode. It also resets the Watchdog Timer (WDT). The WDT is used mainly with the SLEEP instruction. Upon execution of the SLEEP instruction, the entire microcontroller goes into sleep mode by shutting down the main oscillator and by stopping the Program Counter from fetching the next instruction after SLEEP. There are two ways to get out of sleep mode: (a) an external event via hardware interrupt, (b) the internal WDT interrupt. Upon wake-up from a WDT interrupt, the microcontroller resumes operation by executing the next instruction after SLEEP.

Check the Microchip Corp. website for application notes on WDT.

SUBFWB **Subtract fileReg from WREG with borrow**

Function: WREG – fileReg – #borrow ;#borrow is inverted carry
Syntax: SUBFWB f, d, a

This subtracts fileReg and the Carry (borrow) flag from WREG and puts the result in WREG (d = 0) or fileReg (d = 1). The steps for subtraction performed by the internal hardware of the CPU are as follows:

1. Take the 2's complement of the fileReg byte.
2. Add this to register WREG.
3. Add the inverted Carry (borrow) flag to the result.
4. Ignore the Carry.
5. Examine the N (negative) flag for positive or negative result.

Example:

```
    MyReg SET 0x20 ;set aside loc 0x20 for MyReg
    BSF   STATUS,C ;make Carry = 1
    MOVLW 0x45     ;WREG 45H
    MOVWF MyReg    ;MYReg = 45H
    MOVLW 0x23
    SUBWF MyReg    ;WREG = 45H - 23H - 0 = 22H

 45H     0100 0101             0100 0101
-23H     0010 0011 2's comp  + 1101 1101
              Inverted carry +         0
-------                      -----------
+22H                           0010 0010
Because D7 (the N flag) is 0, the result is
positive.
```

This instruction sets the negative flag according to the following:

	N	
WREG > (fileReg + #C)	0	the result is positive
WREG = (fileReg + #C)	0	the result is 0
WREG < (fileReg + #C)	1	the result is negative and in 2's comp

SUBLW **Subtract WREG from Literal value**

Function: Subtract WREG from literal value k (WREG = k – WREG)
Syntax: SUBLW k

This subtracts the WREG value from the literal value k and puts the result in WREG. The steps for subtraction performed by the internal hardware of the CPU are as follows:

1. Take the 2's complement of the WREG value.
2. Add it to literal value k.
3. Ignore the Carry.
4. Examine the N (negative) flag for positive or negative result.

```
MOVLW 0x23      ;WREG 23H
SUBLW 0x45      ;WREG = 45H - 23H = 22H
```

```
 45H      0100 0101             0100 0101
-23H      0010 0011 2's comp   +1101 1101
-------                        ------------------
+22H                            0010 0010
```

Because D7 (the N flag) is 0, the result is positive.

This instruction sets the negative flag according to the following:

	N	
Literal value k > WREG	0	the result is positive
Literal value k = WREG	0	the result is 0
Literal value < WREG	1	the result is negative and in 2's comp

Example:

```
MOVLW 0x98      ;WREG 98H
SUBLW 0x66      ;WREG = 66H - 98H = CEH
```

```
 66H      0110 0110             0110 0110
-98H      1001 1000 2's comp   +0110 1000
 ------                        ----------------
 CEH                             1100 1110
```

Because D7 (the N flag) is 1, the result is negative and in 2's comp.

SUBWF **Subtract WREG from fileReg**

Function: Subtract WREG from fileReg (Dest = fileReg – WREG)
Syntax: SUBWF f, d, a

This subtracts the WREG value from the fileReg value and puts the result in either WREG (d = 0) or fileReg (d = 1). The steps for subtraction performed by the internal hardware of the CPU are as follows:

1. Take the 2's complement of the WREG byte.
2. Add this to the fileReg register.
3. Ignore the carry.
4. Examine the N (negative) flag for positive or negative result.

Example:

```
MyReg SET 0x20 ;set aside loc 0x20 for MyReg
MOVLW 0x45     ;WREG 45H
MOVWF MyReg    ;MYReg = 45H
MOVLW 0x23     ;WREG = 23H
SUBWF MyReg,F  ;MyReg = 45H - 23H = 22H
```

```
 45H      0100 0101            0100 0101
-23H      0010 0011 2's comp  +1101 1101
-------                        ------------------
+22H                           0010 0010
Because D7 (the N flag) is 0, the result is
positive.
```

This instruction sets the negative flag according to the following:

	N	
fileReg > WREG	0	the result is positive
fileReg = WREG	0	the result is 0
fileReg < WREG	1	the result is negative and in 2's comp

SUBWFB **Subtract WREG from fileReg with borrow**

Function: Dest = fileReg – WREG – #borrow ;#borrow is inverted carry
Syntax: SUBWFB f, d, a

This subtracts the WREG value and the inverted borrow (carry) flag from the fileReg value and puts the result in WREG (if d = 0), or fileReg (if d = 1). The steps for subtraction performed by the internal hardware of the CPU are as follows:

1. Take the 2's complement of WREG.
2. Add this to fileReg.
3. Add the inverted Carry flag to the result.
4. Ignore the carry.
5. Examine the N (negative) flag for positive or negative result.

Example:

```
MyReg SET 0x20 ;set aside loc 0x20 for MyReg
BSF     STATUS,C   ;C = 1
MOVLW 0x45     ;WREG 45H
MOVWF MyReg    ;MYReg = 45H
MOVLW 0x23     ;WREG = 23H
SUBWFB MyReg,F ;MyReg = 45H - 23H - 0 = 22H
```

```
 45H      0100 0101            0100 0101
-23H      0010 0011 2's comp  +1101 1101
              Inverted carry +        0
-----                         ------------
+22H                           0010 0010
Because D7 (the N flag) is 0, the result is
positive.
```

This instruction sets the negative flag according to the following:

	N	
fileReg > (WREG + #C)	0	the result is positive
fileReg = (WREG + #C)	0	the result is 0
fileReg < (WREG + #C)	1	the result is negative and in 2's comp

SWAPF Swap Nibbles in fileReg

Function: Swap nibbles within fileReg
Syntax: SAWPF f, d, a

The SWAPF instruction interchanges the lower nibble (D0–D3) with the upper nibble (D4–D7) inside fileReg. The result is placed in WREG (d = 0) or fileReg (d = 1).

Example:

```
MyReg SET 0X20 ;set aside loc 20H for MyReg
MOVLW 0x59H  ;W = 59H (0101 1001 in binary)
MOVWF MyReg    ;MyReg = 59H (0101 1001)
SWAPF MyReg,F  ;MyReg = 95H (1001 0101)
```

TBLRD Table Read

Function: Read a byte from ROM to the TABLAT register
Syntax: TBLRD *
TBLRD *+
TBLRD *-
TBLRD +*

This instruction moves (copies) a byte of data located in program (code) ROM into the TableLatch (TABLAT) register. This allows us to put strings of data, such as look-up table elements, in the code space and read them into the CPU. The address of the desired byte in the program space (on-chip ROM) is held by the TBLPTR register. Table A-6 shows the auto-increment feature of the TBLRD instruction.

Table A-6: PIC18 Table Read Instructions

Instruction	Function	
TBLRD*	Table Read	After read, TBLPTR stays the same
TBLRD*+	Table Read with post-increment	(Read and increment TBLPTR)
TBLRD*-	Table Read with post-decrement	(Read and decrement TBLPTR)
TBLRD+*	Table Read with pre-increment	(increment TBLPTR and read)

Note: A byte of data is read into the TABLAT register from code space pointed to by TBLPTR.

Example: Assume that an ASCII character string is stored in the on-chip ROM program memory starting at address 500H. Write a program to bring each character into the CPU and send it to PORTB.

```
      ORG   0000H           ;burn into ROM starting at 0
      MOVLW LOW(MESSAGE)    ;WREG = 00 low-byte addr.
      MOVWF TBLPTRL         ;look-up table low-byte addr
      MOVLW HIGH(MESSAGE)   ;WREG = 05 = high-byte addr
      MOVWF TBLPTRH         ;look-up table high-byte addr
      CLRF  TBLPTRU         ;clear upper 5 bits

B8    TBLRD*+     ;read the table,then increment TBLPTR
      MOVF  TABLAT,W        ;copy to WREG (Z = 1 if null)
      BZ    EXIT            ;exit if end of string
      MOVWF PORTB           ;copy WREG to PORTB
      BRA   B8
EXIT GOTO EXIT
;---------------------message
      ORG 0x500       ;data burned starting at 0x500
           ORG  0x500
MESSAGE    DB   "The earth is but one country and "
           DB   "mankind its citizens","Baha'u'llah",0
           END
```

In the program above, the TBLPTR holds the address of the desired byte. After the execution of the TBLRD*+ instruction, register TABLAT has the character. Notice that TBLPTR is incremented automatically to point to the next character in the MRESSAGE table.

TBLWT — Table Write

Function: Write to Flash a block of data
Syntax: TBLWT*
TBLWT*+
TBLWT*-
TBLWT+*

This instruction writes a block of data to the program (code) space assuming that the on-chip program ROM is of Flash type. The address of the desired location in Flash ROM is held by the TBLPTR register. The process of writing to Flash ROM using the TBLWT instruction is discussed in Section 14.3 of Chapter 14.

TSTFSZ — Test fileReg, Skip if Zero

Function: Test fileReg for zero value and skip if it is zero
Syntax: TSTFSZ f, a

This instruction tests the entire contents of fileReg for value zero and skips the next instruction if fileReg has zero in it.

Example: Test PORTB for zero continuously.

```
     SETF TRISB  ;make PORTB an input
```

```
     CLRF TRISD  ;make PORTD an output
BACK TSTFSZ PORTB
     BRA    BACK
     MOVFF  PORTB,PORTD
```

Example: Toggle PORTB 250 times.

```
COUNTER   SET  0x40 ;loc 40H for COUNTER
     SETF TRISB     ;PORTB as output
     MOVLW D'250'   ;WREG = 250
     MOVWF COUNTER  ;COUNTER = 250
     MOVLW 0x55     ;WREG = 55H
     MOVWF PORTB
BACK COMF PORTB,F   ;toggle PORTB
     DECF COUNTER,F ;decrement COUNTER
     TSTFSZ COUNTER ;test counter for 0
     BRA  BACK      ;keep doing it
     ......
```

XORLW **Ex-Or Literal with WREG**

Function: Logical exclusive-OR Literal k and WREG
Syntax: XORLW k

This performs a logical exclusive-OR on the Literal value and WREG operands, bit by bit, storing the result in WREG.

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

Example:

```
MOVLW 0x39   ;WREG = 39H
XORLW 0x09   ;WREG = 39H ORed with 09
               ;now, WREG = 30H
39H   0011 1001
09H   0000 1001
30    0011 0000
```

Example:

```
MOVLW 0x32     ;WREG = 32H
XORLW 0x50     ;(now, WREG = 62H)

32H   0011 0010
50H   0101 0000
62H   0110 0010
```

XORWF **Ex-Or WREG with fileReg**

Function: Logical exclusive-OR fileReg and WREG
Syntax: XORWF f, d, a

This performs a logical exclusive-OR on the operands, bit by bit, storing the result in the destination. The destination can be WREG (d = 0), or fileReg (d = 1).

Example:

```
MyReg SET 0x20 ;set aside loc 20h for MyReg
MOVLW 0x39     ;WREG = 39H
MOVWF MyReg    ;MyReg = 39H
MOVLW 0x09     ;WREG = 09H
XORWF MyReg,F   ;MyReg = 39H ORed with 09
               ;MyReg = 30H

39H    0011 1001
09H    0000 1001
30     0011 0000
```

Example:

```
MyReg SET 0x15 ;set aside loc 15 for MyReg
MOVLW 0x32     ;WREG = 32H
MOVWF MyReg    ;MyReg = 32H
MOVLW 0x50     ;WREG = 50H
XORWF MyReg,F   ;now W = 62H

32H    0011 0010
50H    0101 0000
62H    0110 0010.
```

We can place the result in WREG.

Example:

```
MyReg SET 0x15 ;set aside loc 15 for MyReg
MOVLW 0x44     ;WREG = 44H
MOVWF MyReg    ;MyReg = 44H
MOVLW 0x67     ;WREG = 67H
XORWF MyReg    ;now W = 23H, and MyReg = 44H

44H  0100 0100
67H  0110 0111
23H  0010 0011
```

APPENDIX B

BASICS OF WIRE WRAPPING

OVERVIEW

This appendix shows the basics of wire wrapping.

BASICS OF WIRE WRAPPING

Note: For this tutorial appendix, you will need the following:
Wire-wrapping tool (Radio Shack part number 276-1570)
30-gauge (30-AWG) wire for wire wrapping
(Thanks to Shannon Looper and Greg Boyle for their assistance on this section.)

The following describes the basics of wire wrapping:

1. There are several different types of wire-wrap tools available. The best one is available from Radio Shack for less than $10. The part number for the Radio Shack model is 276-1570. This tool combines the wrap and unwrap functions in the same end of the tool and includes a separate stripper. We found this to be much easier to use than the tools that combined all these features on one two-ended shaft. There are also wire-wrap guns, which are, of course, more expensive.
2. Wire-wrapping wire is available prestripped in various lengths or in bulk on a spool. The prestripped wire is usually more expensive and you are restricted to the different wire lengths you can afford to buy. Bulk wire can be cut to any length you wish, which allows each wire to be custom fit.
3. Serveral different types of wire-wrap boards are available. These are usually called perfboards or wire-wrap boards. These types of boards are sold at many electronics stores (such as Radio Shack). The best type of board has plating around the holes on the bottom of the board. These boards are better because the sockets and pins can be soldered to the board, which makes the circuit more mechanically stable.
4. Choose a board that is large enough to accommodate all the parts in your design with room to spare so that the wiring does not become too cluttered. If you wish to expand your project in the future, you should be sure to include enough room on the original board for the complete circuit. Also, if possible, the layout of the IC on the board needs to be such that signals go from left to right just like the schematics.
5. To make the wiring easier and to keep pressure off the pins, install one standoff on each corner of the board. You may also wish to put standoffs on the top of the board to add stability when the board is on its back.
6. For power hook-up, use some type of standard binding post. Solder a few single wire-wrap pins to each power post to make circuit connections (to at least one pin for each IC in the circuit).
7. To further reduce problems with power, each IC must have its own connection to the main power of the board. If your perfboard does not have built-in power buses, run a separate power and ground wire from each IC to the main power. In other words, DO NOT daisy chain (chip-to-chip connection is called daisy chain) power connections, as each connection down the line will have more wire and more resistance to get power through. See Figure B-1. However, daisy chaining is acceptable for other connections such as data, address, and control buses.
8. You must use wire-wrap sockets. These sockets have long square pins whose edges will cut into the wire as it is wrapped around the pin.

9. Wire wrapping will not work on round legs. If you need to wrap to components, such as capacitors, that have round legs, you must also solder these connections. The best way to connect single components is to install individual wire-wrap pins into the board and then solder the components to the pins. An alternate method is to use an empty IC socket to hold small components such as resistors and wrap them to the socket.
10. The wire should be stripped about 1 inch. This will allow 7 to 10 turns for each connection. The first turn or turn-and-a-half should be insulated. This prevents stripped wire from coming in contact with other pins. This can be accomplished by inserting the wire as far as it will go into the tool before making the connection.
11. Try to keep wire lengths to a minimum. This prevents the circuit from looking like a bird nest. Be neat and use color coding as much as possible. Use only red wires for V_{CC} and black wires for ground connections. Also use different colors for data, address, and control signal connections. These suggestions will make troubleshooting much easier.
12. It is standard practice to connect all power lines first and check them for continuity. This will eliminate trouble later on.
13. It's also a good idea to mark the pin orientation on the bottom of the board. Plastic templates are available with pin numbers preprinted on them specifically for this purpose, or you can make your own from paper. Forgetting to reverse pin order when looking at the bottom of the board is a very common mistake when wire wrapping circuits.
14. To prevent damage to your circuit, place a diode (such as IN5338) in reverse bias across the power supply. If the power gets hooked up backwards, the diode will be forward biased and will act as a short, keeping the reversed voltage from your circuit.
15. In digital circuits, there can be a problem with current demand on the power supply. To filter the noise on the power supply, a 100 μF electrolytic capacitor and a 0.1 μF monolithic capacitor are connected from V_{CC} to ground, in parallel with each other, at the entry point of the power supply to the board. These two together will filter both the high- and the low-frequency noises. Instead of using two capacitors in parallel, you can use a single 20–100 μF tantalum capacitor. Remember that the long lead is the positive one.
16. To filter the transient current, use a 0.1 μF monolithic capacitor for each IC. Place the 0.1 μF monolithic capacitor between V_{CC} and ground of each IC. Make sure the leads are as short as possible.

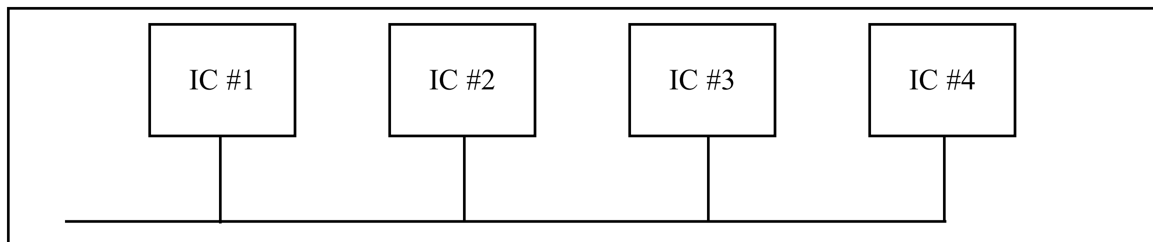


Figure B-1. Daisy Chain Connection (not recommended for power lines)

APPENDIX C

IC TECHNOLOGY AND SYSTEM DESIGN ISSUES

OVERVIEW

This appendix provides an overview of IC technology and PIC18 interfacing. In addition, we look at the microcontroller-based system as a whole and examine some general issues in system design.

First, in Section C.1, we provide an overview of IC technology. Then, in Section C.2, the internal details of PIC18 I/O ports and interfacing are discussed. Section C.3 examines system design issues.

C.1: OVERVIEW OF IC TECHNOLOGY

In this section we examine IC technology and discuss some major developments in advanced logic families. Because this is an overview, it is assumed that the reader is familiar with logic families on the level presented in basic digital electronics books.

Transistors

The transistor was invented in 1947 by three scientists at Bell Laboratory. In the 1950s, transistors replaced vacuum tubes in many electronics systems, including computers. It was not until 1959 that the first integrated circuit was successfully fabricated and tested by Jack Kilby of Texas Instruments. Prior to the invention of the IC, the use of transistors, along with other discrete components such as capacitors and resistors, was common in computer design. Early transistors were made of germanium, which was later abandoned in favor of silicon. This was because the slightest rise in temperature resulted in massive current flows in germanium-based transistors. In semiconductor terms, it is because the band gap of germanium is much smaller than that of silicon, resulting in a massive flow of electrons from the valence band to the conduction band when the temperature rises even slightly. By the late 1960s and early 1970s, the use of the silicon-based IC was widespread in mainframes and minicomputers. Transistors and ICs at first were based on P-type materials. Later on, because the speed of electrons is much higher (about two-and-a-half times) than the speed of holes, N-type devices replaced P-type devices. By the mid-1970s, NPN and NMOS transistors had replaced the slower PNP and PMOS transistors in every sector of the electronics industry, including in the design of microprocessors and computers. Since the early 1980s, CMOS (complementary MOS) has become the dominant technology of IC design. Next we provide an overview of differences between MOS and bipolar transistors. See Figure C-1.

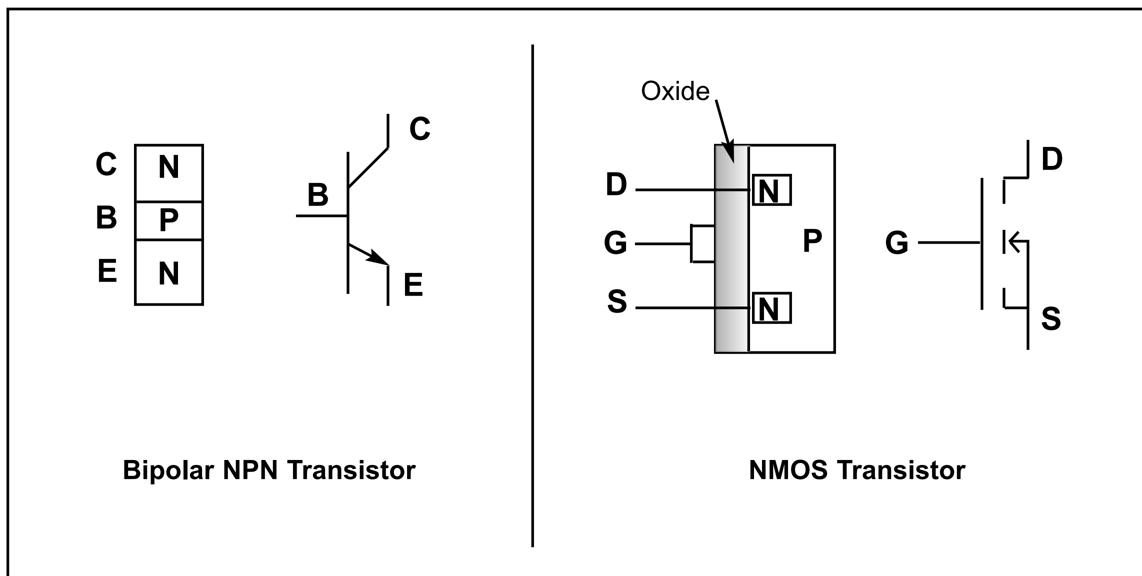


Figure C-1. Bipolar vs. MOS Transistors

MOS vs. bipolar transistors

There are two types of transistors: bipolar and MOS (metal-oxide semiconductor). Both have three leads. In bipolar transistors, the three leads are referred to as the *emitter*, *base*, and *collector*, while in MOS transistors they are named *source*, *gate*, and *drain*. In bipolar transistors, the carrier flows from the emitter to the collector, and the base is used as a flow controller. In MOS transistors, the carrier flows from the source to the drain, and the gate is used as a flow controller. In NPN-type bipolar transistors, the electron carrier leaving the emitter must overcome two voltage barriers before it reaches the collector (see Figure C-1). One is the N-P junction of the emitter-base and the other is the P-N junction of the base-collector. The voltage barrier of the base-collector is the most difficult one for the electrons to overcome (because it is reverse-biased) and it causes the most power dissipation. This led to the design of the unipolar type transistor called MOS. In N-channel MOS transistors, the electrons leave the source and reach the drain without going through any voltage barrier. The absence of any voltage barrier in the path of the carrier is one reason why MOS dissipates much less power than bipolar transistors. The low power dissipation of MOS allows millions of transistors to fit on a single IC chip. In today's technology, putting 10 million transistors into an IC is common, and it is all because of MOS technology. Without the MOS transistor, the advent of desktop personal computers would not have been possible, at least not so soon. The bipolar transistors in both the mainframes and minicomputers of the 1960s and 1970s were bulky and required expensive cooling systems and large rooms. MOS transistors do have one major drawback: They are slower than bipolar transistors. This is due partly to the gate capacitance of the MOS transistor. For a MOS to be turned on, the input capacitor of the gate takes time to charge up to the turn-on (threshold) voltage, leading to a longer propagation delay.

Overview of logic families

Logic families are judged according to (1) speed, (2) power dissipation, (3) noise immunity, (4) input/output interface compatibility, and (5) cost. Desirable qualities are high speed, low power dissipation, and high noise immunity (because it prevents the occurrence of false logic signals during switching transition). In interfacing logic families, the more inputs that can be driven by a single output, the better. This means that high-driving-capability outputs are desired. This, plus the fact that the input and output voltage levels of MOS and bipolar transistors are not compatible mean that one must be concerned with the ability of one logic family to drive the other one. In terms of the cost of a given logic family, it is high during the early years of its introduction but it declines as production and use rise.

The case of inverters

As an example of logic gates, we look at a simple inverter. In a one-transistor inverter, the transistor plays the role of a switch, and R is the pull-up resistor. See Figure C-2. For this inverter to work most effectively in digital circuits, however, the R value must be high when the transistor is "on" to limit the current flow from V_{CC} to ground in order to have low power dissipation (P = VI, where V

= 5 V). In other words, the lower the I, the lower the power dissipation. On the other hand, when the transistor is "off", R must be a small value to limit the voltage drop across R, thereby making sure that V_{OUT} is close to V_{CC}. This is a contradictory demand on R. This is one reason that logic gate designers use active components (transistors) instead of passive components (resistors) to implement the pull-up resistor R.

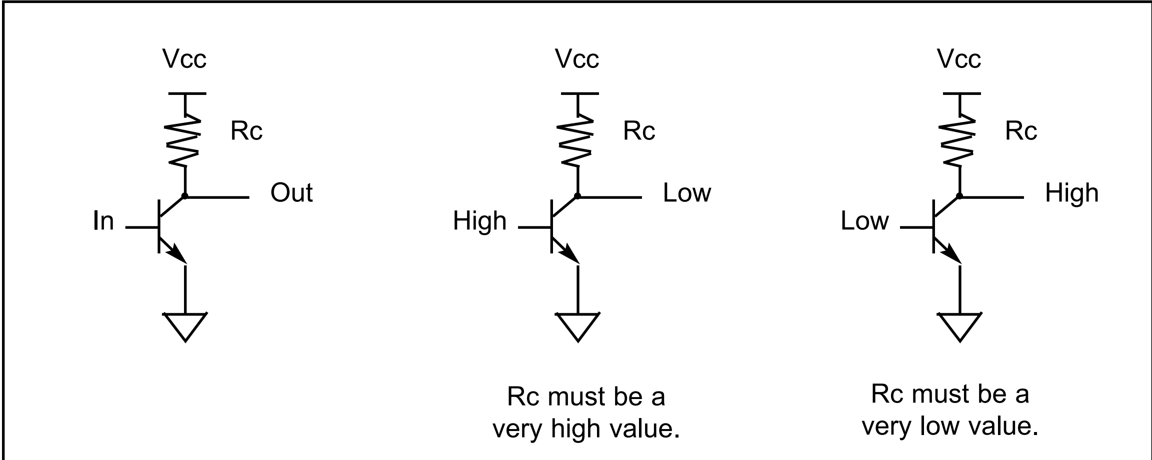


Figure C-2. One-Transistor Inverter with Pull-up Resistor

The case of a TTL inverter with totem-pole output is shown in Figure C-3. In Figure C-3, Q3 plays the role of a pull-up resistor.

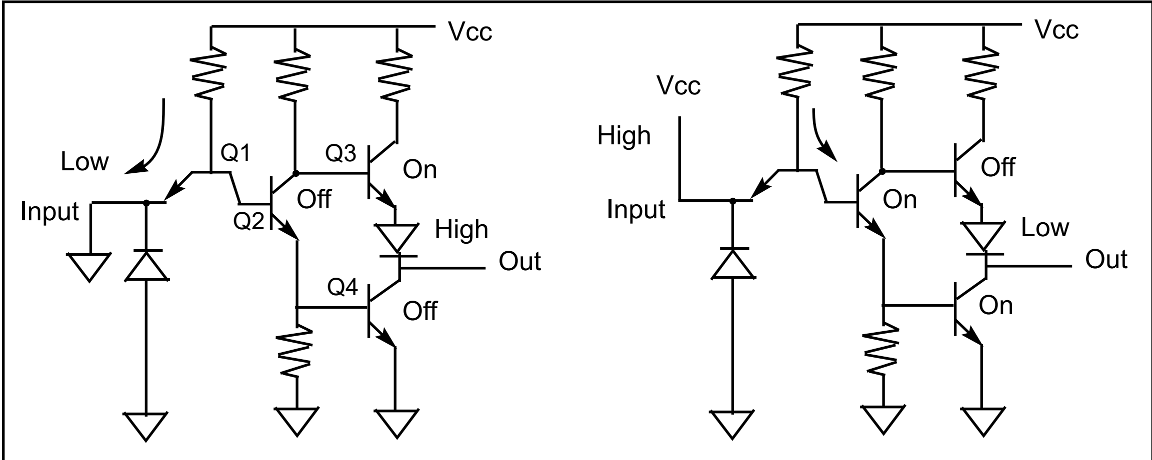


Figure C-3. TTL Inverter with Totem-Pole Output

CMOS inverter

In the case of CMOS-based logic gates, PMOS and NMOS are used to construct a CMOS (complementary MOS) inverter as shown in Figure C-4. In CMOS inverters, when the PMOS transistor is off, it provides a very high impedance path, making leakage current almost zero (about 10 nA); when the PMOS is on, it provides a low resistance on the path of V_{DD} to load. Because the speed of the hole is slower than that of the electron, the PMOS transistor is wider to compensate for this disparity; therefore, PMOS transistors take more space than NMOS transistors in the CMOS gates. At the end of this section we will see an open-collector gate in which the pull-up resistor is provided externally, thereby allowing system designers to choose the value of the pull-up resistor.

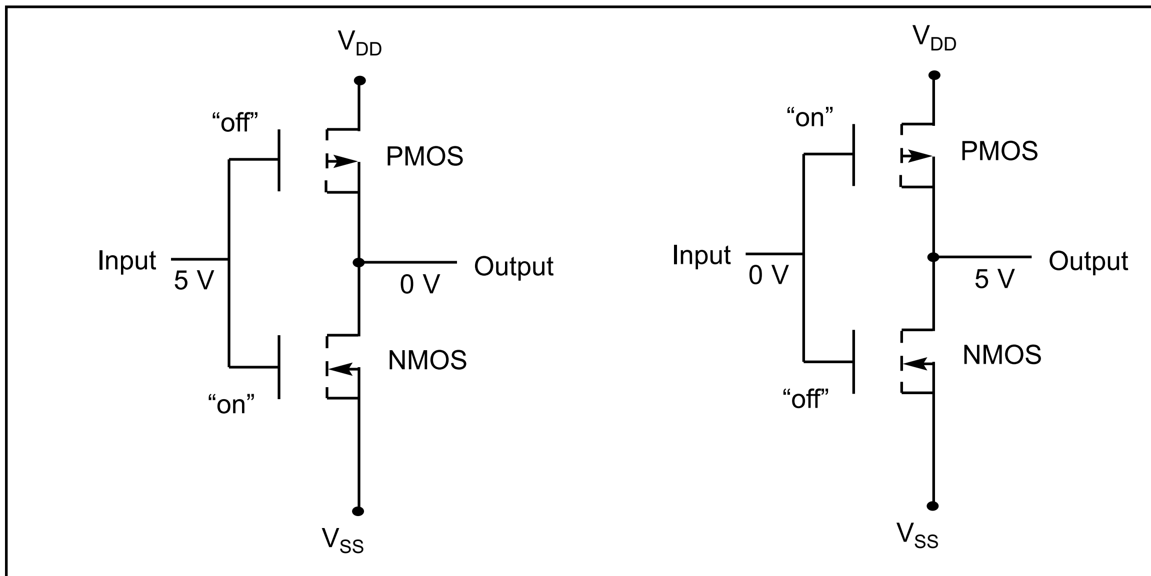


Figure C-4. CMOS Inverter

Input/output characteristics of some logic families

In 1968 the first logic family made of bipolar transistors was marketed. It was commonly referred to as the standard TTL (transistor-transistor logic) family. The first MOS-based logic family, the CD4000/74C series, was marketed in 1970. The addition of the Schottky diode to the base-collector of bipolar transistors in the early 1970s gave rise to the S family. The Schottky diode shortens the propagation delay of the TTL family by preventing the collector from going into what is called deep saturation. Table C-1 lists major characteristics of some logic families. In Table C-1, note that as the CMOS circuit's operating frequency rises, the power dissipation also increases. This is not the case for bipolar-based TTL.

Table C-1: Characteristics of Some Logic Families

Characteristic	STD TTL	LSTTL	ALSTTL	HCMOS
V_{CC}	5 V	5 V	5 V	5 V
V_{IH}	2.0 V	2.0 V	2.0 V	3.15 V
V_{IL}	0.8 V	0.8 V	0.8 V	1.1 V
V_{OH}	2.4 V	2.7 V	2.7 V	3.7 V
V_{OL}	0.4 V	0.5 V	0.4 V	0.4 V
I_{IL}	−1.6 mA	−0.36 mA	−0.2 mA	−1 μA
I_{IH}	40 μA	20 μA	20 μA	1 μA
I_{OL}	16 mA	8 mA	4 mA	4 mA
I_{OH}	−400 μA	−400 μA	−400 μA	4 mA
Propagation delay	10 ns	9.5 ns	4 ns	9 ns
Static power dissipation (f = 0)	10 mW	2 mW	1 mW	0.0025 nW
Dynamic power dissipation at f = 100 kHz	10 mW	2 mW	1 mW	0.17 mW

History of logic families

Early logic families and microprocessors required both positive and negative power voltages. In the mid-1970s, 5 V V_{CC} became standard. In the late 1970s, advances in IC technology allowed combining the speed and drive of the S family with the lower power of LS to form a new logic family called FAST (Fairchild Advanced Schottky TTL). In 1985, AC/ACT (Advanced CMOS Technology), a much higher speed version of HCMOS, was introduced. With the introduction of FCT (Fast CMOS Technology) in 1986, the speed gap between CMOS and TTL at last was closed. Because FCT is the CMOS version of FAST, it has the low power consumption of CMOS but the speed is comparable with TTL. Table C-2 provides an overview of logic families up to FCT.

Table C-2: Logic Family Overview

Product	Year Introduced	Speed (ns)	Static Supply Current (mA)	High/Low Family Drive (mA)
Std TTL	1968	40	30	−2/32
CD4K/74C	1970	70	0.3	−0.48/6.4
LS/S	1971	18	54	−15/24
HC/HCT	1977	25	0.08	−6/−6
FAST	1978	6.5	90	−15/64
AS	1980	6.2	90	−15/64
ALS	1980	10	27	−15/64
AC/ACT	1985	10	0.08	−24/24
FCT	1986	6.5	1.5	−15/64

Reprinted by permission of Electronic Design Magazine, c. 1991.

Recent advances in logic families

As the speed of high-performance microprocessors reached 25 MHz, it shortened the CPU's cycle time, leaving less time for the path delay. Designers normally allocate no more than 25% of a CPU's cycle time budget to path delay. Following this rule means that there must be a corresponding decline in the propagation delay of logic families used in the address and data path as the system frequency is increased. In recent years, many semiconductor manufacturers have responded to this need by providing logic families that have high speed, low noise, and high drive I/O. Table C-3 provides the characteristics of high-performance logic families introduced in recent years. ACQ/ACTQ are the second-generation advanced CMOS (ACMOS) with much lower noise. While ACQ has the CMOS input level, ACTQ is equipped with TTL-level input. The FCTx and FCTx-T are second-generation FCT with much higher speed. The “x” in the FCTx and FCTx-T refers to various speed grades, such as A, B, and C, where A means low speed and C means high speed. For designers who are well versed in using the FAST logic family, FASTr is an ideal choice because it is faster than FAST, has higher driving capability (I_{OL}, I_{OH}), and produces much lower noise than FAST. At the time of this writing, next to ECL and gallium arsenide logic gates, FASTr is the fastest logic family in the market (with the 5 V V_{CC}), but the power consumption is high relative to other logic families, as shown in Table C-3. The combining of

high-speed bipolar TTL and the low power consumption of CMOS has given birth to what is called BICMOS. Although BICMOS seems to be the future trend in IC design, at this time it is expensive due to extra steps required in BICMOS IC fabrication, but in some cases there is no other choice. (For example, Intel's Pentium microprocessor, a BICMOS product, had to use high-speed bipolar transistors to speed up some of the internal functions.) Table C-3 provides advanced logic characteristics. The "x" is for different speeds designated as A, B, and C. A is the slowest one while C is the fastest one. The above data is for the 74244 buffer.

Table C-3: Advanced Logic General Characteristics

Family	Year	Number Suppliers	Tech Base	I/O Level	Speed (ns)	Static Current	I_{OH}/I_{OL}
ACQ	1989	2	CMOS	CMOS/CMOS	6.0	80 µA	−24/24 mA
ACTQ	1989	2	CMOS	TTL/CMOS	7.5	80 µA	−24/24 mA
FCTx	1987	3	CMOS	TTL/CMOS	4.1–4.8	1.5 mA	−15/64 mA
FCTxT	1990	2	CMOS	TTL/TTL	4.1–4.8	1.5 mA	−15/64 mA
FASTr	1990	1	Bipolar	TTL/TTL	3.9	50 mA	−15/64 mA
BCT	1987	2	BICMOS	TTL/TTL	5.5	10 mA	−15/64 mA

Reprinted by permission of Electronic Design Magazine, c. 1991.

Since the late 70s, the use of a +5 V power supply has become standard in all microprocessors and microcontrollers. To reduce power consumption, 3.3 V V_{CC} is being embraced by many designers. The lowering of V_{CC} to 3.3 V has two major advantages: (1) it lowers the power consumption, prolonging the life of the battery in systems using a battery, and (2) it allows a further reduction of line size (design rule) to submicron dimensions. This reduction results in putting more transistors in a given die size. As fabrication processes improve, the decline in the line size is reaching submicron level and transistor densities are approaching 1 billion transistors.

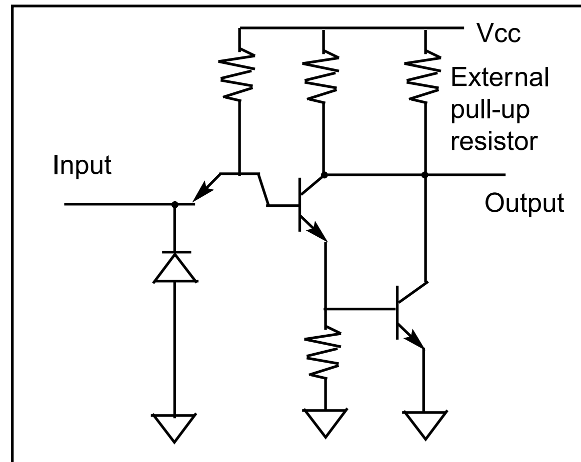


Figure C-5. Open Collector

Open-collector and open-drain gates

To allow multiple outputs to be connected together, we use open-collector logic gates. In such cases, an external resistor will serve as load. This is shown in Figures C-5 and C-6.

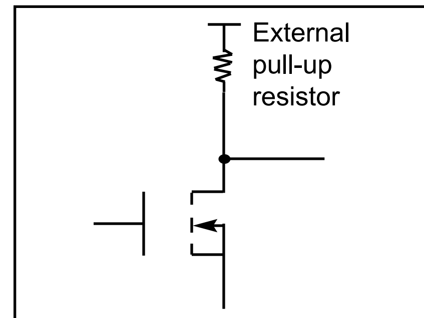


Figure C-6. Open Drain

SECTION C.2: PIC18 I/O PORT STRUCTURE AND INTERFACING

In interfacing the PIC18 microcontroller with other IC chips or devices, fan-out is the most important issue. To understand the PIC18 fan-out we must first understand the port structure of the PIC18. This section provides a detailed discussion of the PIC18 port structure and its fan-out. It is very critical that we understand the I/O port structure of the PIC18 lest we damage it while trying to interface it with an external device.

IC fan-out

When connecting IC chips together, we need to find out how many input pins can be driven by a single output pin. This is a very important issue and involves the discussion of what is called IC fan-out. The IC fan-out must be addressed for both logic "0" and logic "1" outputs. See Example C-1. Fan-out for logic LOW and fan-out for logic HIGH are defined as follows:

$$\text{fan-out (of LOW)} = \frac{I_{OL}}{I_{IL}} \qquad \text{fan-out (of HIGH)} = \frac{I_{OH}}{I_{IH}}$$

Of the above two values, the lower number is used to ensure the proper noise margin. Figure C-7 shows the sinking and sourcing of current when ICs are connected together.

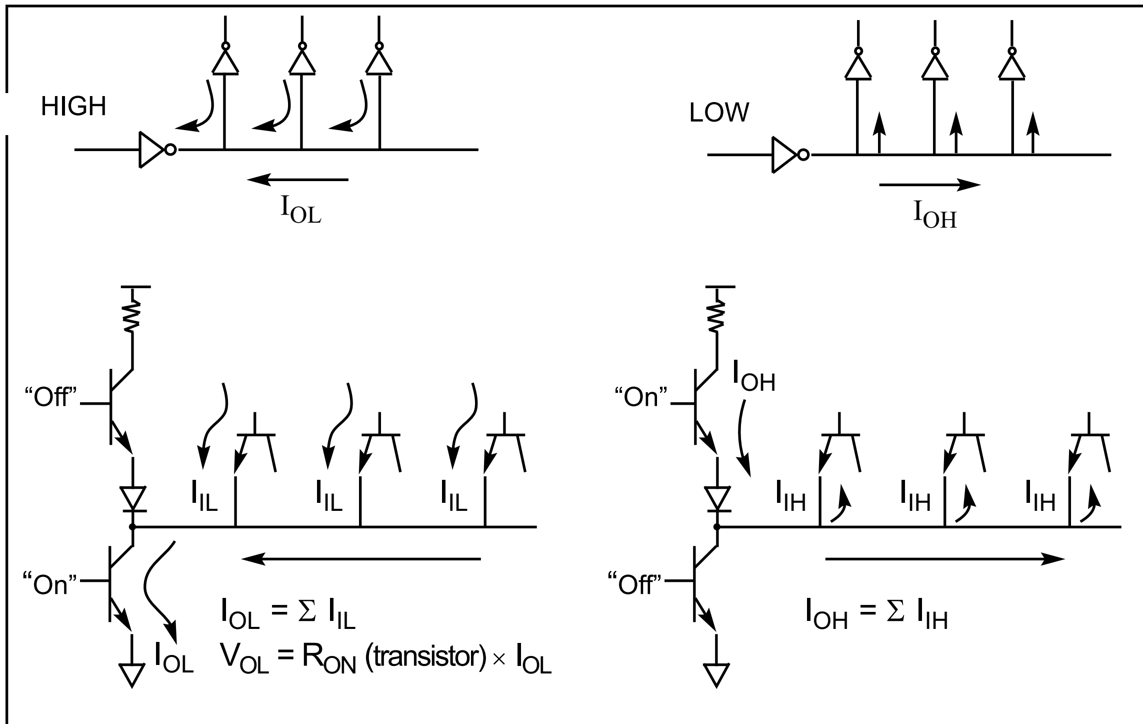


Figure C-7. Current Sinking and Sourcing in TTL

Notice that in Figure C-7, as the number of input pins connected to a single output increases, I_{OL} rises, which causes V_{OL} to rise. If this continues, the rise of V_{OL} makes the noise margin smaller, and this results in the occurrence of false logic due to the slightest noise.

Example C-1

Find how many unit loads (UL) can be driven by the output of the LS logic family.

Solution:

The unit load is defined as I_{IL} = 1.6 mA and I_{IH} = 40 μA. Table C-1 shows I_{OH} = 400 μA and I_{OL} = 8 mA for the LS family. Therefore, we have

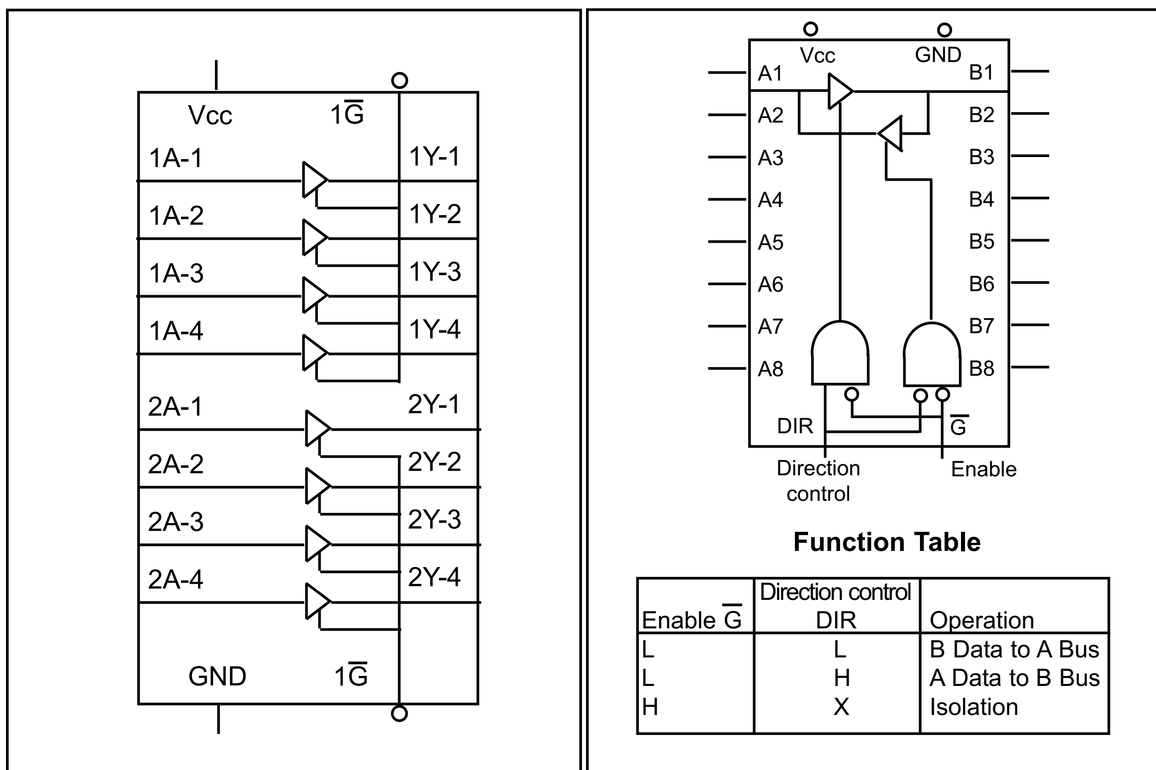
$$\text{fan-out (LOW)} = \frac{I_{OL}}{I_{IL}} = \frac{8\ \text{mA}}{1.6\ \text{mA}} = 5$$

$$\text{fan-out (HIGH)} = \frac{I_{OH}}{I_{IH}} = \frac{400\ \mu\text{A}}{40\ \mu\text{A}} = 10$$

This means that the fan-out is 5. In other words, the LS output must not be connected to more than 5 inputs with unit load characteristics.

74LS244 and 74LS245 buffers/drivers

In cases where the receiver current requirements exceed the driver's capability, we must use buffers/drivers such as the 74LS245 and 74LS244. Figure C-8 shows the internal gates for the 74LS244 and 74LS245. The 74LS245 is used for bidirectional data buses, and the 74LS244 is used for unidirectional address buses.



Function Table

Enable $\overline{G}$	Direction control DIR	Operation
L	L	B Data to A Bus
L	H	A Data to B Bus
H	X	Isolation

Figure C-8 (a). 74LS244 Octal Buffer

(Reprinted by permission of Texas Instruments, Copyright Texas Instruments, 1988)

Figure C-8 (b). 74LS245 Bidirectional Buffer

(Reprinted by permission of Texas Instruments, Copyright Texas Instruments, 1988)

Tri-state buffer

Notice that the 74LS244 is simply 8 tri-state buffers in a single chip. As shown in Figure C-9 a tri-state buffer has a single input, a single output, and the enable control input. By activating the enable, data at the input is transferred to the output. The enable can be an active-LOW or an active-HIGH. Notice that the enable input for the 74LS244 is an active-LOW whereas the enable input pin for Figure C-9 is active-HIGH.

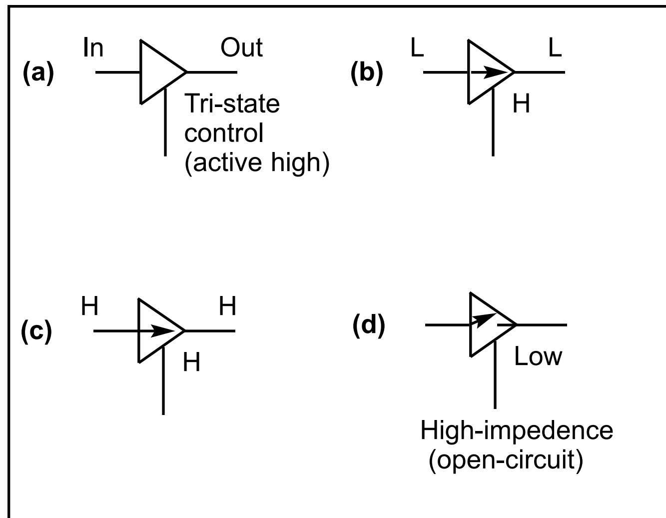


Figure C-9. Tri-State Buffer

74LS245 and 74LS244 fan-out

It must be noted that the output of the 74LS245 and 74LS244 can sink and source a much larger amount of current than that of other LS gates. See Table C-4. That is the reason we use these buffers for driver when a signal is travelling a long distance through a cable or it has to drive many inputs.

Table C-4: Electrical Specifications for Buffers/Drivers

	I_{OH} (mA)	I_{OL} (mA)
74LS244	3	12
74LS245	3	12

After this background on the fan-out, next we discuss the structure of PIC18 ports.

PIC18 port structure and operation

Because all the ports of the PIC18 are bidirectional they all have the following four components in their structure:

1. Data latch
2. Output driver
3. Input buffer
4. TRIS latch

Figure C-10 shows the structure of a port and its four components. Notice that in Figure C-10, the PIC18 ports have both the latch and buffer. Now the question is, in reading the port, are we reading the status of the input pin or are we read-

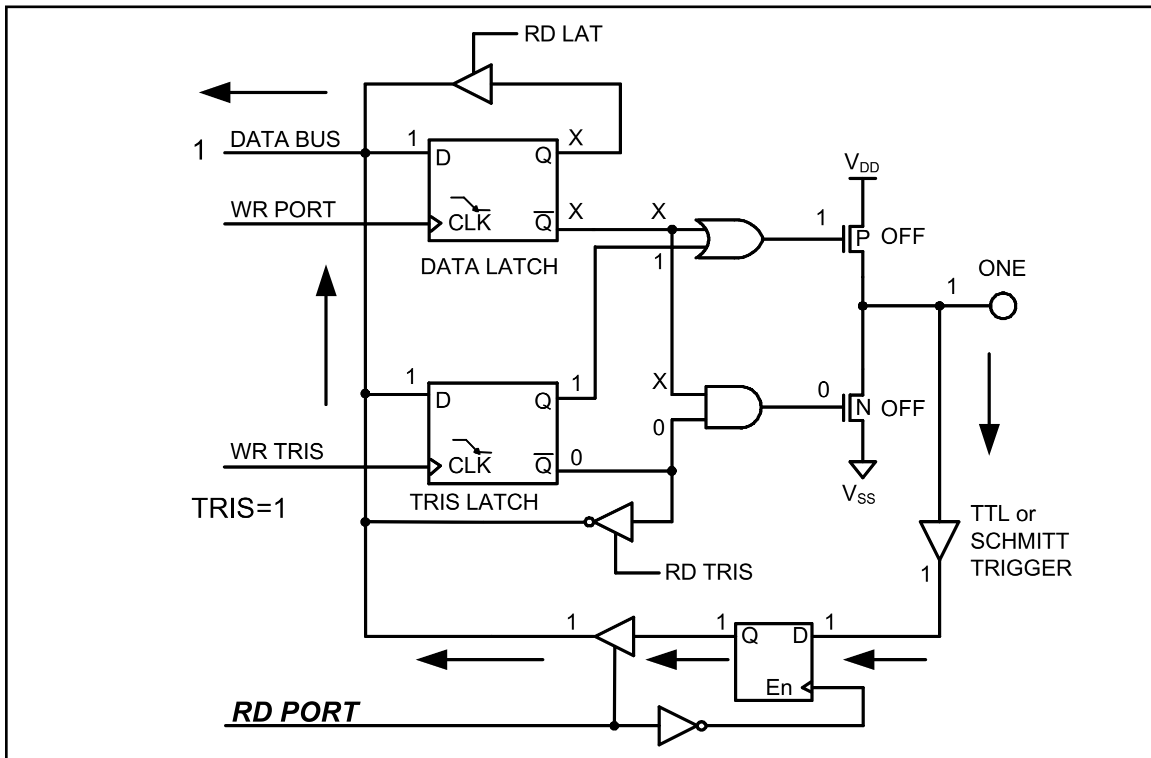


Figure C-10. Inputting (Reading) 1 from a Pin in the PIC18

ing the status of the latch? That is an extremely important question and its answer depends on which instruction we are using. Therefore, when reading the ports there are two possibilities: (1) reading the input pin, or (2) reading the latch. The above distinction is very important and must be understood lest you damage the PIC18 port. Each is described next.

Reading the pin when TRIS = 1 (Input)

As we stated in Chapter 4, to make any bits of any port of the PIC18 an input port, we first must write a 1 (logic HIGH) to the TRIS bit. Look at the following sequence of events to see why:

1. As can be seen from Figure C-10, a 1 written to the TRIS latch has "HIGH" on its Q. Therefore, Q = 1 and $\overline{Q}$ = 0. Because Q = 1, it turns off the P transistor.
2. Because $\overline{Q}$ = 0 and is connected to the gate of the N transistor, the N transistor is off.
3. When both transistors are off, they block any path to the ground or VCC for any signal connected to the input pin, and the input signal is directed to the buffer.
4. When reading the input port in instructions such as "`MOVFW PORTB`" we are really reading the data present at the pin. In other words, it is bringing into the CPU the status of the external pin. This instruction activates the read pin of buffer and lets data at the pins flow into the CPU's internal bus. Figures C-10 and C-11 show HIGH and LOW signals at the input, respectively.

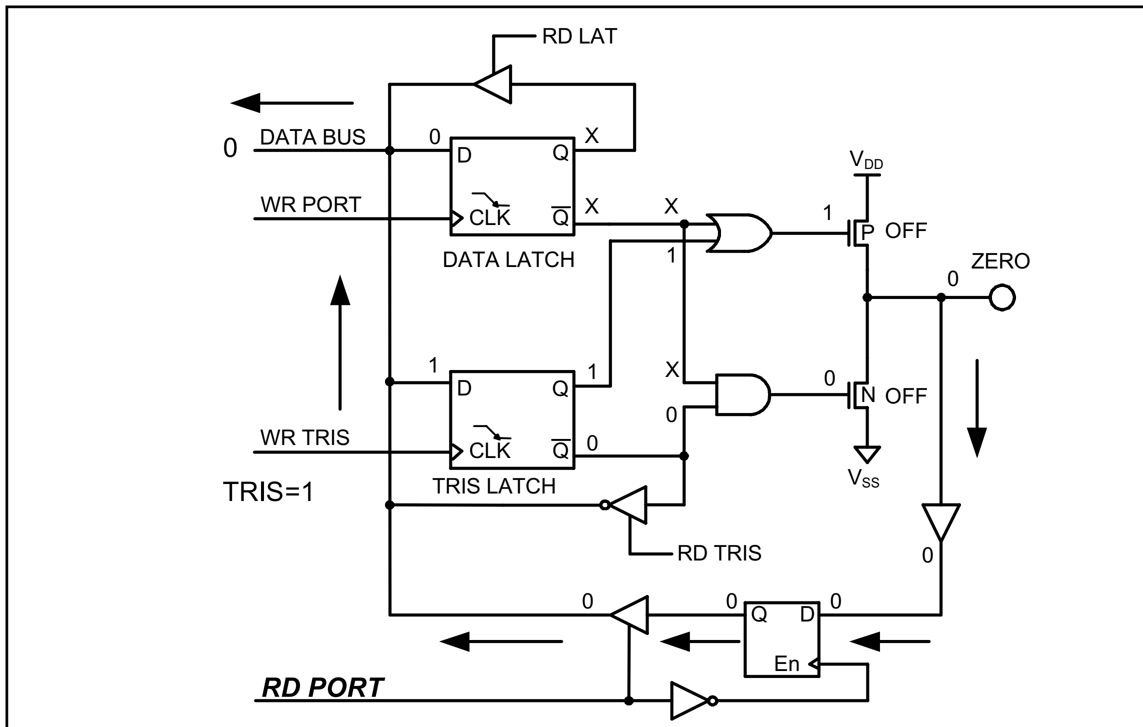


Figure C-11. Inputting (Reading) 0 from a Pin in the PIC18

Writing to pin when TRIS = 0 (Output)

The above discussion showed why we must write a "HIGH" to a port's TRIS bits in order to make it an input port. What happens if we write a "0" to TRIS that was configured as an input port? From Figure C-12 we see that when TRIS = 0, if we write a 0 to the Data latch, then Q = 0 and $\overline{Q}$ = 1. As a result of $\overline{Q}$ = 1, the N transistor is "on" and the P transistor is "off." If N is "on," it provides the path to ground for the input pin. Therefore, any attempt to read the input pin will always get the "LOW" ground signal. Figure C-13 shows what happens when we write "HIGH" to output port (Data latch) when TRIS = 0. Writing 1 to the Data latch makes $\overline{Q}$ = 0. As a result of that, the P transistor is "on" and the N transistor is "off," which allows a 1 to be provided to the output pin. Therefore, any attempt to read the input pin will always get the "HIGH" signal.

Avoid damaging the port

The following methods can be used as precautions to prevent damage to the PIC18 ports:

1. Have a 10k ohms resistor on the V_{CC} path to limit current flow.
2. Connect any input switch to a 74LS244 tri-state buffer before it is fed to the PIC18 pin.

The above points are extremely important and must be emphasized because many people damage their ports and afterwards wonder how it happened. We must also use the right instruction when we want to read the status of an input pin. Table C-5 shows the list of instructions in which reading the port reads the status of the input pin.

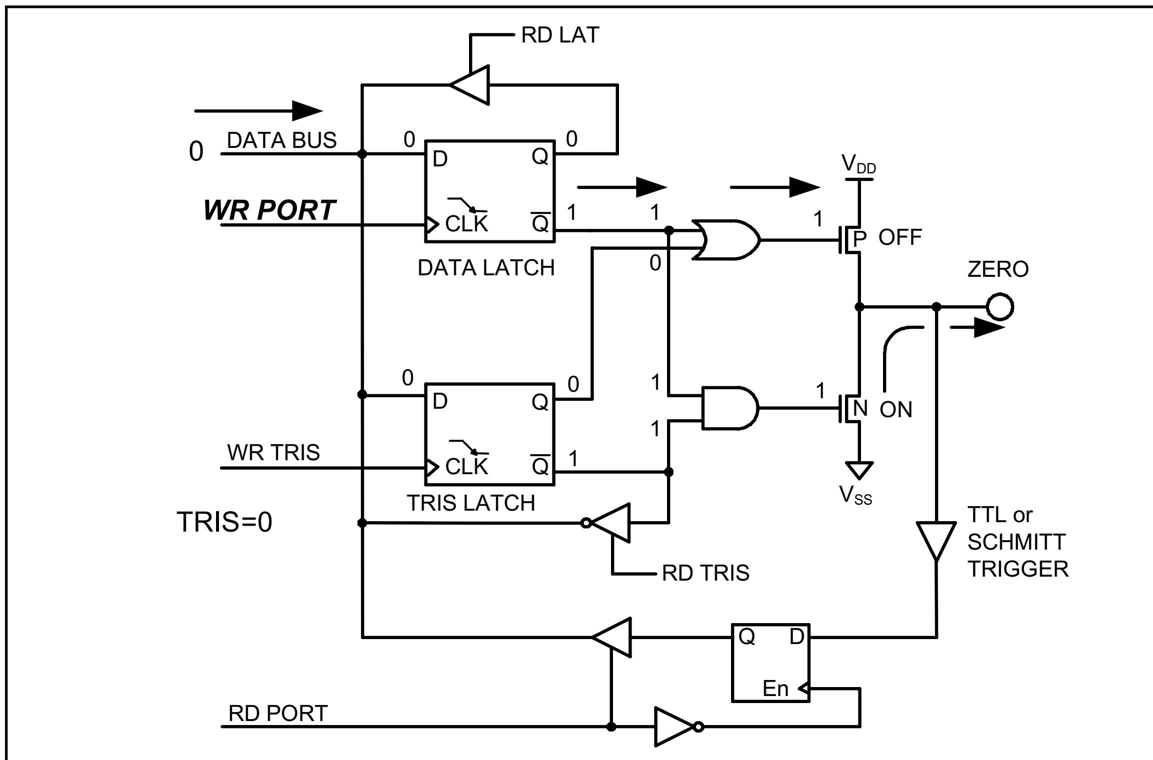


Figure C-12. Outputting (Writing) 0 to a Pin in the PIC18

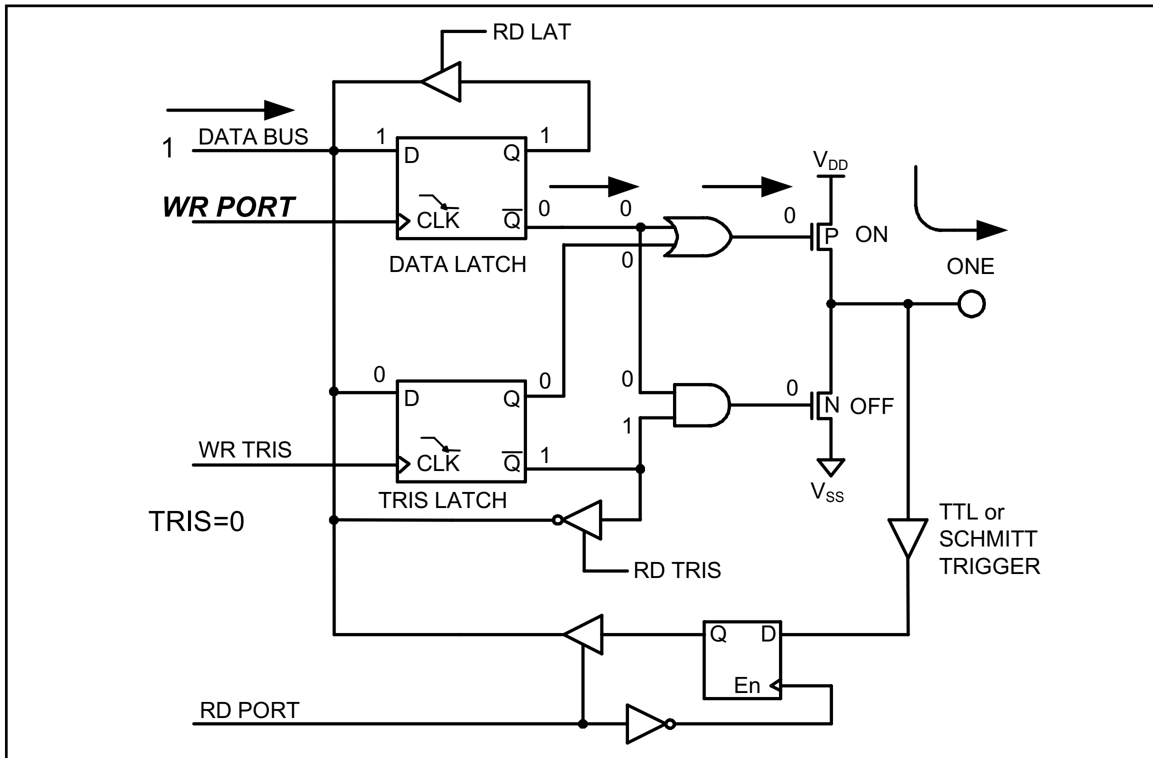


Figure C-13. Outputting (Writing) 1 to a Pin in the PIC18

Table C-5: Some of the Instructions Reading the Status of Input Port

Mnemonics	Examples
`MOVFW PORTx`	`MOVFW PORTB`
`TSTFSZ f`	`TSTFSZ PORTC`
`BTFSS f,b`	`BTFSS PORTD,0`
`BTFSC f,b`	`BTFSC PORTB,7`
`CPFSEQ f`	`CPFSEQ PORTB`

PIC18 port fan-out

Now that we are familiar with the port structure of the PIC18, we need to examine the fan-out for the PIC18 microconctroller. While the early chips were based on NMOS IC technology, today's PIC18 microcontrollers are all based on CMOS technology. Note, however, that while the core of the PIC18 microcontroller is CMOS, the circuitry driving its pins is all TTL compatible. That is, the PIC18 is a CMOS-based product with TTL-compatible pins. All the ports of the PIC18 have the same I/O structure, and therefore the same fan-out. Table C-6 provides the I/O characteristics of PIC18F458 ports.

Table C-6: PIC18 Fan-out for PORTS

Pin	Fan-out
IOL	8.5 mA
IOH	–3 mA
IIL	1 μA
IIH	1 μA

Note: Negative current is defined as current sourced by the pin.

74LS244 driving an output pin

In some cases, when an PIC18 port is driving multiple inputs, or driving a single input via a long wire or cable (e.g., printer cable), we can use the 74LS244 as a driver. When driving an off-board circuit, placing the 74LS244 buffer between your PIC18 and the circuit is essential because the PIC18 lacks sufficient current. See Figure C-14.

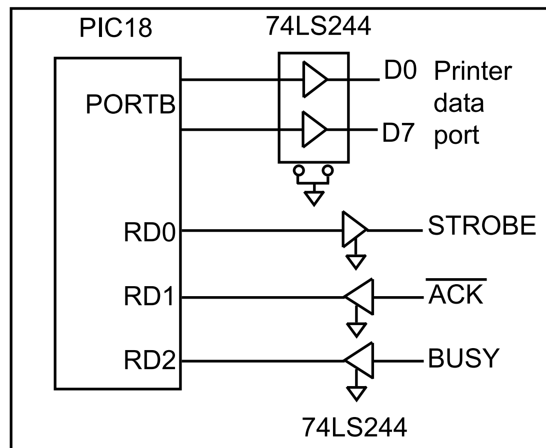


Figure C-14. PIC18 Connection to Printer Signals

SECTION C.3: SYSTEM DESIGN ISSUES

In addition to fan-out, the other issues related to system design are power dissipation, ground bounce, V_{CC} bounce, crosstalk, and transmission lines. In this section we provide an overview of these topics.

Power dissipation considerations

Power dissipation of a system is a major concern of system designers, especially for laptop and hand-held systems in which batteries provide the power. Power dissipation is a function of frequency and voltage as shown below:

$$Q = CV$$

$$\frac{Q}{T} = \frac{CV}{T}$$

$$\text{since} \quad F = \frac{1}{T} \quad \text{and} \quad I = \frac{Q}{T}$$

$$I = CVF$$

$$\text{now} \quad P = VI = CV^2F$$

In the above equations, the effects of frequency and V_{CC} voltage should be noted. While the power dissipation goes up linearly with frequency, the impact of the power supply voltage is much more pronounced (squared). See Example C-2.

Example C-2

Compare the power consumption of two microcontroller-based systems. One uses 5 V and the other uses 3 V for V_{CC}.

Solution:
Because P = VI, by substituting I = V/R we have $P = V^2/R$. Assuming that R = 1, we have $P = 5^2 = 25$ W and $P = 3^2 = 9$ W. This results in using 16 W less power, which means power saving of 64%. ($16/25 \times 100$) for systems using 3 V for power source.

Dynamic and static currents

Two major types of currents flow through an IC: dynamic and static. A dynamic current is I = CVF. It is a function of the frequency under which the component is working. This means that as the frequency goes up, the dynamic current and power dissipation go up. The static current, also called DC, is the current consumption of the component when it is inactive (not selected). The dynamic current dissipation is much higher than the static current consumption. To reduce power consumption, many microcontrollers, including the PIC18, have power-saving modes. In the PIC18, the power saving mode is called *sleep mode.* We describe the sleep mode next.

Sleep mode

In sleep mode the on-chip oscillator is frozen, which cuts off frequency to the CPU and peripheral functions, such as serial ports, interrupts, and timers. Notice that while this mode brings power consumption down to an absolute minimum, the contents of RAM and the SFR registers are saved and remain unchanged.

Ground bounce

One of the major issues that designers of high-frequency systems must grapple with is ground bounce. Before we define ground bounce, we will discuss lead inductance of IC pins. There is a certain amount of capacitance, resistance, and inductance associated with each pin of the IC. The size of these elements varies depending on many factors such as length, area, and so on.

The inductance of the pins is commonly referred to as *self-inductance* because there is also what is called *mutual inductance*, as we will show below. Of the three components of capacitor, resistor, and inductor, the property of self-inductance is the one that causes the most problems in high-frequency system design because it can result in ground bounce. Ground bounce occurs when a massive amount of current flows through the ground pin caused by many outputs changing from HIGH to LOW all at the same time. See Figure C-15(a). The voltage is related to the inductance of the ground lead as follows:

$$V = L \frac{di}{dt}$$

As we increase the system frequency, the rate of dynamic current, di/dt, is also increased, resulting in an increase in the inductance voltage L (di/dt) of the ground pin. Because the LOW state (ground) has a small noise margin, any extra voltage due to the inductance can cause a false signal. To reduce the effect of ground bounce, the following steps must be taken where possible:

1. The V_{CC} and ground pins of the chip must be located in the middle rather than at opposite ends of the IC chip (the 14-pin TTL logic IC uses pins 14 and 7 for ground and V_{CC}). This is exactly what we see in high-performance logic gates such as Texas Instruments' advanced logic AC11000 and ACT11000 families. For example, the ACT11013 is a 14-pin DIP chip in which pin numbers 4 and 11 are used for the ground and V_{CC}, instead of 7 and 14 as in the traditional TTL family. We can also use the SOIC packages instead of DIP.
2. Another solution is to use as many pins for ground and V_{CC} as possible to reduce the lead length. This is exactly why all high-performance microprocessors and logic families use many pins for V_{CC} and ground instead of the traditional single pin for V_{CC} and single pin for GND. For example, in the case of Intel's Pentium processor there are over 50 pins for ground, and another 50 pins for V_{CC}.

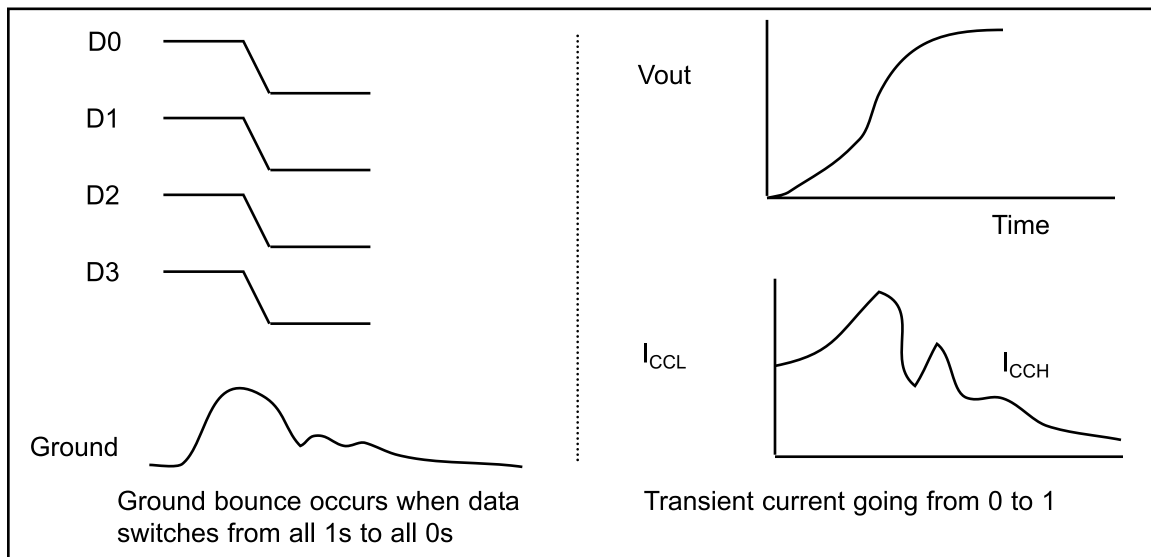


Figure C-15. (a) Ground Bounce (b) Transient Current

The above discussion of ground bounce is also applicable to V_{CC} when a large number of outputs changes from the LOW to the HIGH state; this is referred to as *V_{CC} bounce*. However, the effect of V_{CC} bounce is not as severe as ground bounce because the HIGH ("1") state has a wider noise margin than the LOW ("0") state.

Filtering the transient currents using decoupling capacitors

In the TTL family, the change of the output from LOW to HIGH can cause what is called *transient current*. In a totem-pole output in which the output is LOW, Q4 is on and saturated, whereas Q3 is off. By changing the output from the LOW to the HIGH state, Q3 turns on and Q4 turns off. This means that there is a time when both transistors are on and drawing current from V_{CC}. The amount of current depends on the R_{ON} values of the two transistors, which in turn depend on the internal parameters of the transistors. The net effect of this, however, is a large amount of current in the form of a spike for the output current, as shown in Figure C-15(b). To filter the transient current, a 0.01 μF or 0.1 μF ceramic disk capacitor can be placed between the V_{CC} and ground for each TTL IC. The lead for this capacitor, however, should be as small as possible because a long lead results in a large self-inductance, and that results in a spike on the V_{CC} line [$V = L\ (di/dt)$]. This spike is called V_{CC} bounce. The ceramic capacitor for each IC is referred to as a *decoupling capacitor*. There is also a bulk decoupling capacitor, as described next.

Bulk decoupling capacitor

If many IC chips change state at the same time, the combined currents drawn from the board's V_{CC} power supply can be massive and may cause a fluctuation of V_{CC} on the board where all the ICs are mounted. To eliminate this, a relatively large decoupling tantalum capacitor is placed between the V_{CC} and ground lines. The size and location of this tantalum capacitor varies depending on the number of ICs on the board and the amount of current drawn by each IC, but it is

common to have a single 22 μF to 47 μF capacitor for each of the 16 devices, placed between the V_{CC} and ground lines.

Crosstalk

Crosstalk is due to mutual inductance. See Figure C-16. Previously, we discussed self-inductance, which is inherent in a piece of conductor. *Mutual inductance* is caused by two electric lines running parallel to each other. The mutual inductance is a function of l, the length of two conductors running in parallel, d, the distance between them, and the medium material placed between them. The effect of crosstalk can be reduced by increasing the distance between the parallel or adjacent lines (in printed circuit boards, they will be traces). In many cases, such as printer and disk drive cables, there is a dedicated ground for each signal. Placing ground lines (traces) between signal lines reduces the effect of crosstalk. This method is used even in some ACT logic families where a V_{CC} and a GND pin are next to each other. Crosstalk is also called *EMI* (electromagnetic interference). This is in contrast to *ESI* (electrostatic interference), which is caused by capacitive coupling between two adjacent conductors.

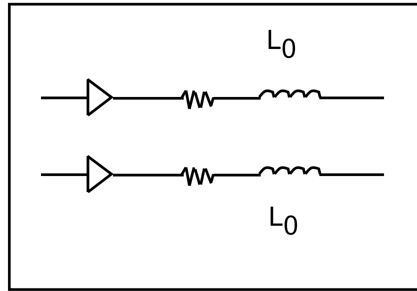


Figure C-16. Crosstalk (EMI)

Transmission line ringing

The square wave used in digital circuits is in reality made of a single fundamental pulse and many harmonics of various amplitudes. When this signal travels on the line, not all the harmonics respond in the same way to the capacitance, inductance, and resistance of the line. This causes what is called *ringing*, which depends on the thickness and the length of the line driver, among other factors. To reduce the effect of ringing, the line drivers are terminated by putting a resistor at the end of the line. See Figure C-17. There are three major methods of line driver termination: parallel, serial, and Thevenin.

In serial termination, resistors of 30–50 ohms are used to terminate the line. The parallel and Thevenin methods are used in cases where there is a need to match the impedance of the line with the load impedance. This requires a detailed analysis of the signal traces and load impedance, which is beyond the scope of this book. In high-frequency systems, wire traces on the printed circuit board (PCB) behave like transmission lines, causing ringing. The severity of this ringing depends on the speed and the logic family used. Table C-7 provides the length of the traces, beyond which the traces must be looked at as transmission lines.

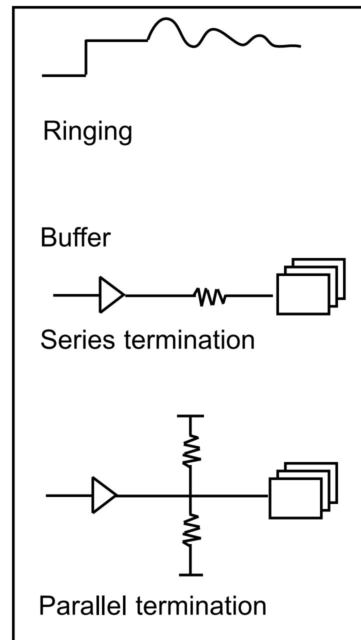


Figure C-17. Reducing Transmission Line Ringing

Table C-7: Line Length Beyond Which Traces Behave Like Transmission Lines

Logic Family	Line Length (in.)
LS	25
S, AS	11
F, ACT	8
AS, ECL	6
FCT, FCTA	5

(Reprinted by permission of Integrated Device Technology, copyright IDT 1991)

APPENDIX D

FLOWCHARTS AND PSEUDOCODE

OVERVIEW

This appendix provides an introduction to writing flowcharts and pseudocode.

Flowcharts

If you have taken any previous programming courses, you are probably familiar with flowcharting. Flowcharts use graphic symbols to represent different types of program operations. These symbols are connected together into a flowchart to show the flow of execution of a program. Figure D-1 shows some of the more commonly used symbols. Flowchart templates are available to help you draw the symbols quickly and neatly.

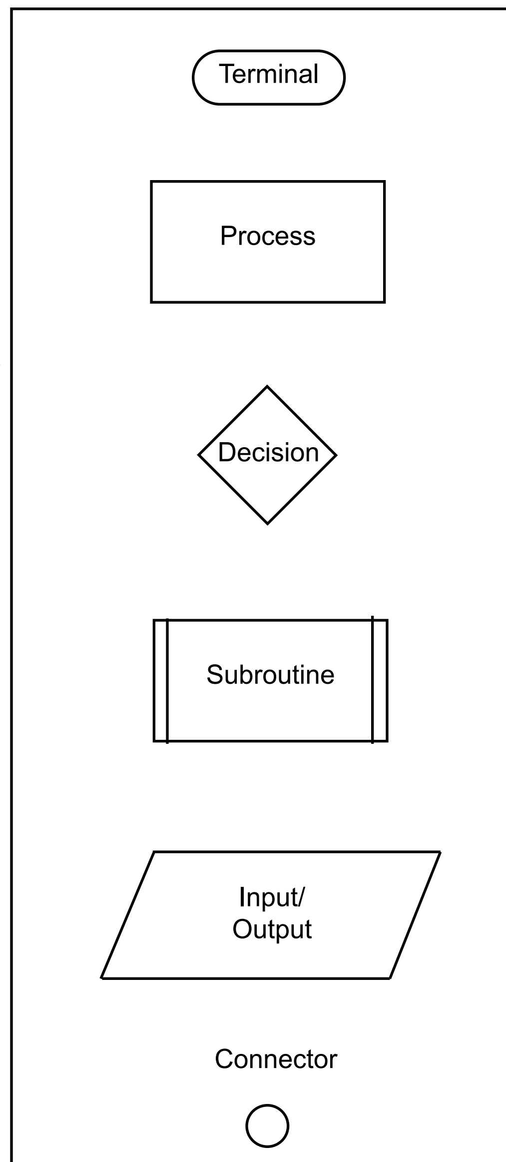


Figure D-1. Commonly Used Flowchart Symbols

Pseudocode

Flowcharting has been standard practice in industry for decades. However, some find limitations in using flowcharts, such as the fact that you can't write much in the little boxes, and it is hard to get the "big picture" of what the program does without getting bogged down in the details. An alternative to using flowcharts is pseudocode, which involves writing brief descriptions of the flow of the code. Figures D-2 through D-6 show flowcharts and pseudocode for commonly used control structures.

Structured programming uses

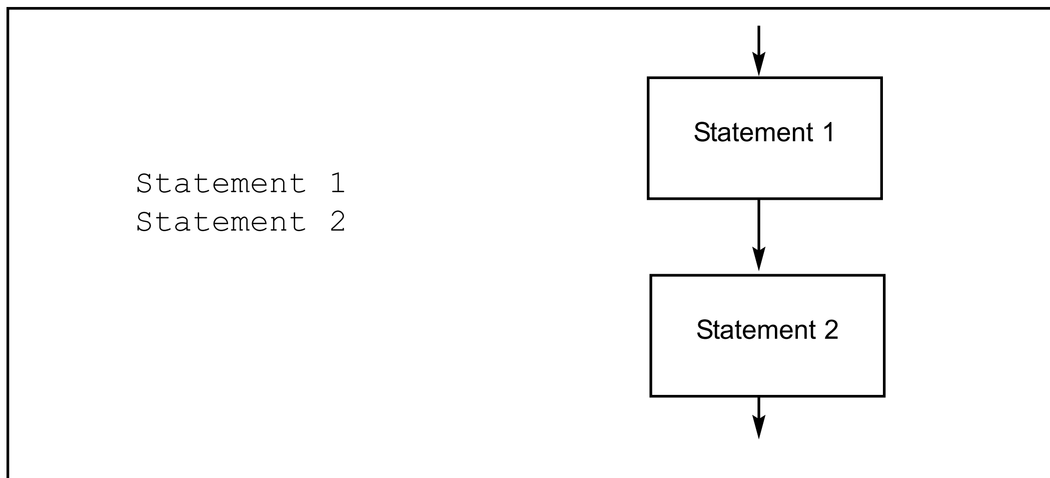


Figure D-2. SEQUENCE Pseudocode versus Flowchart

three basic types of program control structures: sequence, control, and iteration. Sequence is simply executing instructions one after another. Figure D-2 shows how sequence can be represented in pseudocode and flowcharts.

Figures D-3 and D-4 show two control programming structures: IF-THEN-ELSE and IF-THEN in both pseudocode and flowcharts.

Note in Figures D-2 through D-6 that “statement” can indicate one statement or a group of statements.

Figures D-5 and D-6 show two iteration control structures: REPEAT UNTIL and WHILE DO. Both structures execute a statement or group of statements repeatedly. The difference between them is that the REPEAT UNTIL structure always executes the statement(s) at least once, and checks the condition after each iteration, whereas the WHILE DO may not execute the statement(s) at all because the condition is checked at the beginning of each iteration.

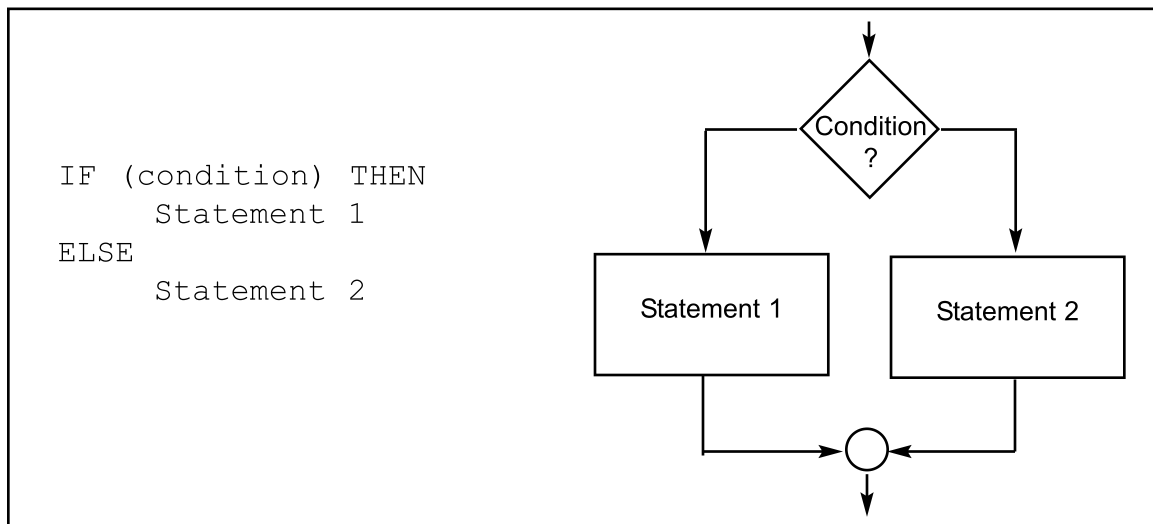


Figure D-3. IF THEN ELSE Pseudocode versus Flowchart

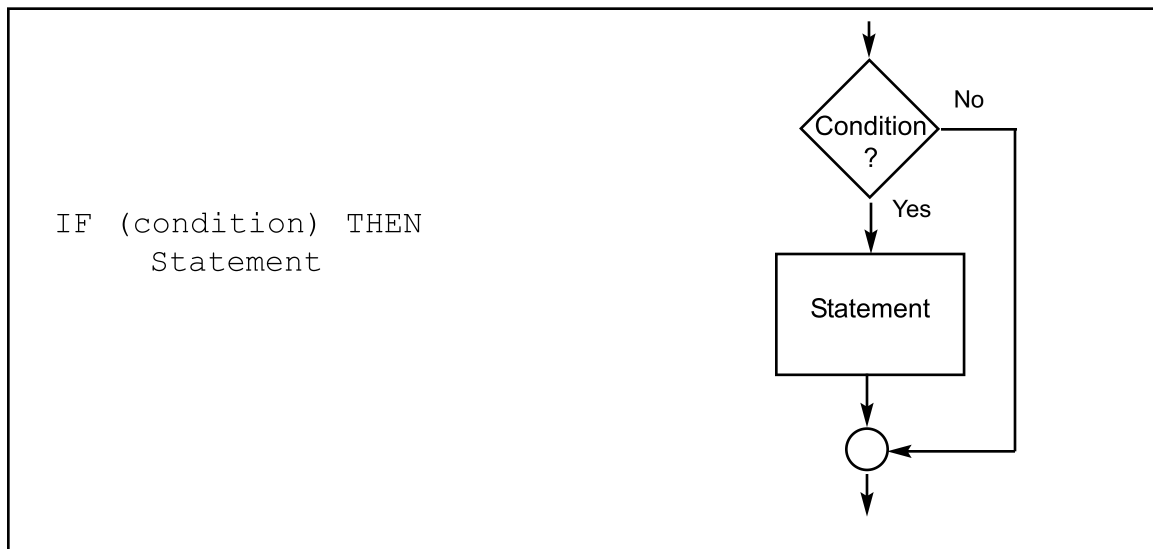


Figure D-4. IF THEN Pseudocode versus Flowchart

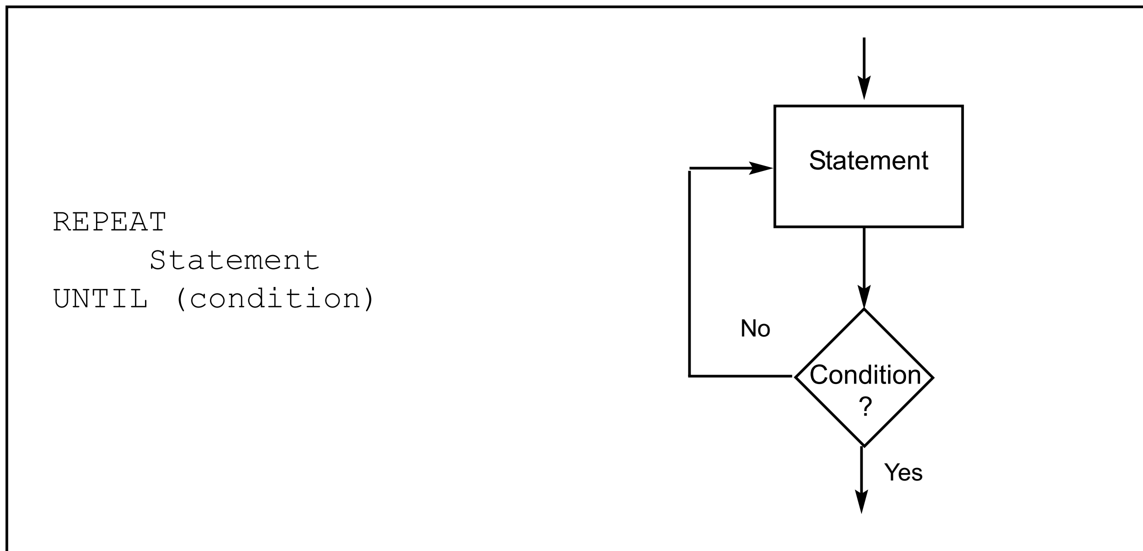


Figure D-5. REPEAT UNTIL Pseudocode versus Flowchart

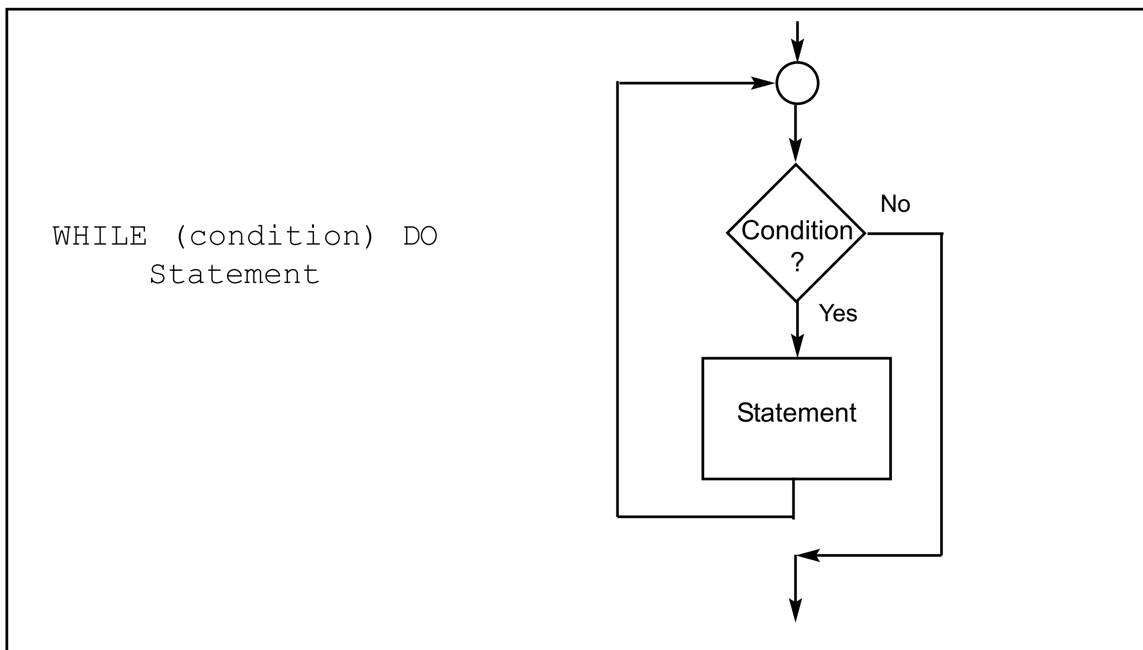


Figure D-6. WHILE DO Pseudocode versus Flowchart

Program D-1 finds the sum of a series of bytes. Compare the flowchart versus the pseudocode for Program D-1 (shown in Figure D-7). In this example, more program details are given than one usually finds. For example, this shows steps for initializing and decrementing counters. Another programmer may not include these steps in the flowchart or pseudocode. It is important to remember that the purpose of flowcharts or pseudocode is to show the flow of the program and what the program does, not the specific Assembly language instructions that accomplish the program's objectives. Notice also that the pseudocode gives the same information in a much more compact form than does the flowchart. It is important to note that sometimes pseudocode is written in layers, so that the outer level or layer shows the flow of the program and subsequent levels show more details of how the program accomplishes its assigned tasks.

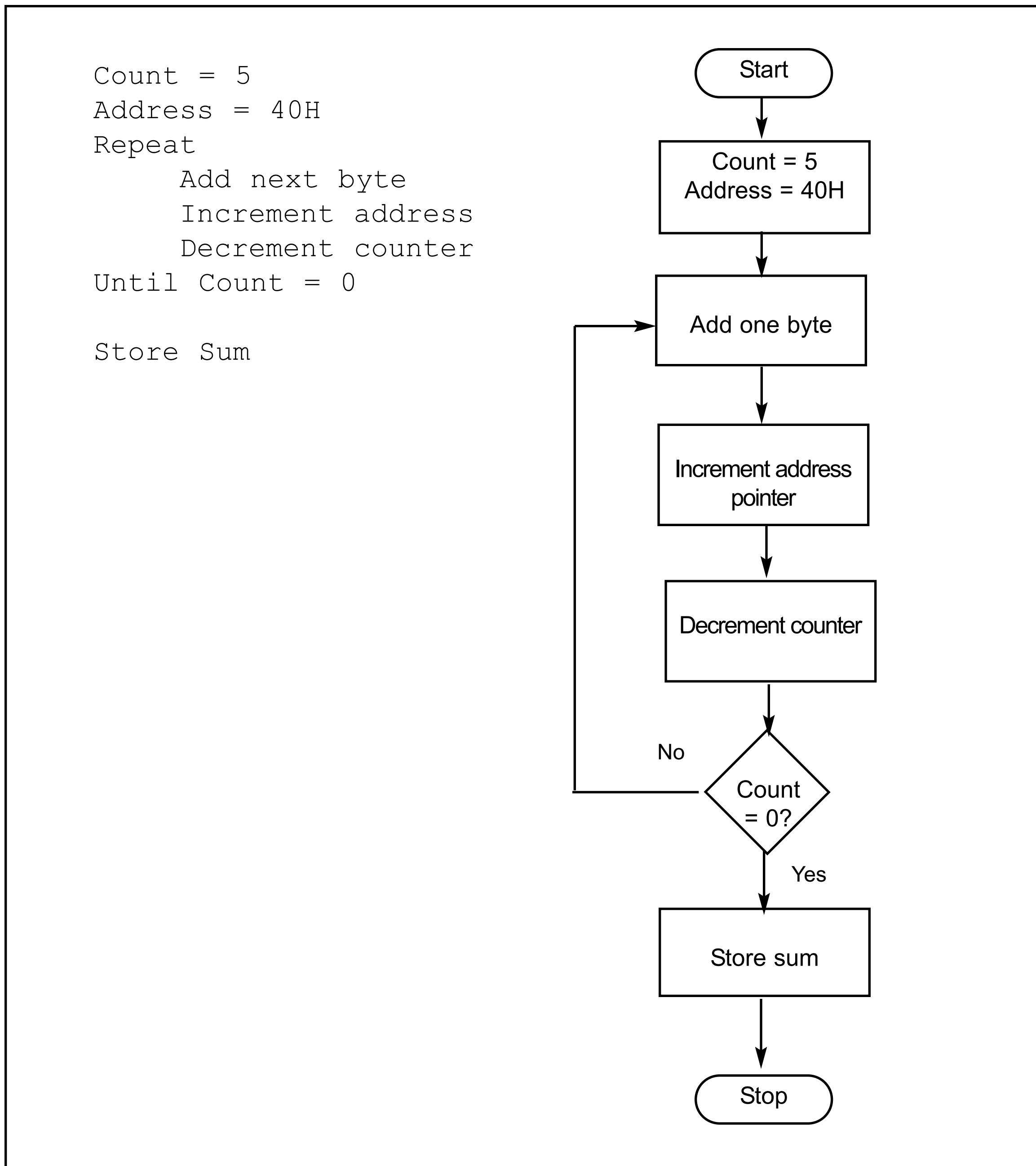


Figure D-7. Pseudocode versus Flowchart for Program D-1

```
COUNTVAL    EQU 5              ;COUNT = 5
COUNTREG    SET 0x20           ;set aside location 20H for counter
SUM         SET 0x30           ;set aside location 30H for sum
      MOVLW  COUNTVAL          ;WREG = 5
      MOVWF  COUNTREG          ;load the counter
      LFSR   0,0x40            ;load pointer. FSR0 = 40H, RAM address
      CLRF   WREG              ;clear WREG
B5    ADDWF  POSTINC0, W       ;add RAM to WREG and increment FSR0
      DECF   COUNTREG,F        ;decrement counter
      BNZ    B5                ;loop until counter = zero
      MOVWF  SUM               ;store WREG in SUM
```

Program D-1

APPENDIX E.1

PIC18 PRIMER FOR x86 PROGRAMMERS

	x86	PIC18
8-bit registers:	AL, AH, BL, BH, CL, CH, DL, DH	WREG and up to 256 RAM locations in Access Bank
16-bit (data pointer):	BX, SI, DI	TBLPTR
Program Counter:	IP (16-bit)	PC (21-bit)
Input:	`MOV DX,port addr` `IN AL,DX`	`MOVFW PORTx ;(x = A,B,..G)`
Output:	`MOV DX,port addr` `OUT DX,AL`	`MOVWF PORTx ;(x = A,B,..G)`
Loop:	`DEC CL` `JNZ TARGET`	`DECF MyReg,F` `BNZ TARGET`
Stack pointer:	SP (16-bit)	SP (21-bit)
	As we PUSH data onto the stack, it decrements the SP.	Push increments the SP. (Used exclusively for saving PC)
	As we POP data from the stack, it increments the SP.	Pop decrements the SP. (Used exclusively for retrieving PC)
Data movement:		
From the code segment:	`MOV AL,CS:[SI]`	`TBLRD`
From the data segment:	`MOV AL,[SI]`	`MOVFW FSRx`
From RAM:	`MOV AL,[SI]` (Use SI, DI, or BX only.)	`MOVFW FSRx`
To RAM:	`MOV [SI],AL`	`MOVWF FSRx`

APPENDIX E.2

PIC18 PRIMER FOR 8051 PROGRAMMERS

	8051	PIC18
8-bit registers:	A, B, R0, R1, ...R7	WREG and up to 256 RAM locations in Access Bank
16-bit (data pointer):	DPTR	TBLPTR
Program Counter:	PC (16-bit)	PC (21-bit)
Input:	`MOV A,Pn ; (n=0 - 3)`	`MOVFW PORTx ;(x = A,B,..G)`
Output:	`MOV Pn,A ; (n=0 - 3)`	`MOVWF PORTx ;(x = A,B,..G)`
Loop:	`DJNZ R3, TARGET` `(Using R0-R7)`	`DECF MyReg,F` `BNZ TARGET`
Stack pointer:	SP (8-bit)	SP (21-bit)
	As we PUSH data onto the stack, it increments the SP.	Push increments the SP. (Used exclusively for saving PC)
	As we POP data from the stack, it decrements the SP.	Pop decrements the SP. (Used exclusively for retrieving PC)
Data movement:		
From the code segment:	`MOVC A,@A+PC`	`TBLRD`
From the data segment:	`MOVX A,@DPTR`	`MOVFW FSRx`
From RAM:	`MOV A, @R0` (Use R0 or R1 only)	`MOVFW FSRx`
To RAM:	`MOV @R0,A` (Use R0 or R1 only)	`MOVWF FSRx`

APPENDIX F

ASCII CODES

Ctrl	Dec	Hex	Ch	Code
^@	0	00		NUL
^A	1	01	☺	SOH
^B	2	02	☻	STX
^C	3	03	♥	ETX
^D	4	04	♦	EOT
^E	5	05	♣	ENQ
^F	6	06	♠	ACK
^G	7	07	•	BEL
^H	8	08	◘	BS
^I	9	09	○	HT
^J	10	0A	◙	LF
^K	11	0B	♂	VT
^L	12	0C	♀	FF
^M	13	0D	♪	CR
^N	14	0E	♫	SO
^O	15	0F	☼	SI
^P	16	10	►	DLE
^Q	17	11	◄	DC1
^R	18	12	↕	DC2
^S	19	13	‼	DC3
^T	20	14	¶	DC4
^U	21	15	§	NAK
^V	22	16	▬	SYN
^W	23	17	↨	ETB
^X	24	18	↑	CAN
^Y	25	19	↓	EM
^Z	26	1A	→	SUB
^[	27	1B	←	ESC
^\	28	1C	∟	FS
^]	29	1D	↔	GS
^^	30	1E	▲	RS
^_	31	1F	▼	US

Dec	Hex	Ch
32	20	
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?

Dec	Hex	Ch
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	_

Dec	Hex	Ch
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	¦
125	7D	}
126	7E	~
127	7F	⌂

Dec	Hex	Ch	Dec	Hex	Ch	Dec	Hex	Ch	Dec	Hex	Ch
128	80	Ç	160	A0	á	192	C0	└	224	E0	α
129	81	ü	161	A1	í	193	C1	┴	225	E1	β
130	82	é	162	A2	ó	194	C2	┬	226	E2	Γ
131	83	â	163	A3	ú	195	C3	├	227	E3	π
132	84	ä	164	A4	ñ	196	C4	─	228	E4	Σ
133	85	à	165	A5	Ñ	197	C5	┼	229	E5	σ
134	86	å	166	A6	ª	198	C6	╞	230	E6	µ
135	87	ç	167	A7	º	199	C7	╟	231	E7	τ
136	88	ê	168	A8	¿	200	C8	╚	232	E8	Φ
137	89	ë	169	A9	⌐	201	C9	╔	233	E9	Θ
138	8A	è	170	AA	¬	202	CA	╩	234	EA	Ω
139	8B	ï	171	AB	½	203	CB	╦	235	EB	δ
140	8C	î	172	AC	¼	204	CC	╠	236	EC	∞
141	8D	ì	173	AD	¡	205	CD	═	237	ED	φ
142	8E	Ä	174	AE	«	206	CE	╬	238	EE	ε
143	8F	Å	175	AF	»	207	CF	╧	239	EF	∩
144	90	É	176	B0	░	208	D0	╨	240	F0	≡
145	91	æ	177	B1	▒	209	D1	╤	241	F1	±
146	92	Æ	178	B2	▓	210	D2	╥	242	F2	≥
147	93	ô	179	B3	│	211	D3	╙	243	F3	≤
148	94	ö	180	B4	┤	212	D4	?	244	F4	⌠
149	95	ò	181	B5	╡	213	D5	╒	245	F5	⌡
150	96	û	182	B6	╢	214	D6	╓	246	F6	÷
151	97	ù	183	B7	╖	215	D7	╫	247	F7	≈
152	98	ÿ	184	B8	╕	216	D8	╪	248	F8	≈
153	99	Ö	185	B9	╣	217	D9	┘	249	F9	∙
154	9A	Ü	186	BA	║	218	DA	┌	250	FA	·
155	9B	¢	187	BB	╗	219	DB	█	251	FB	√
156	9C	£	188	BC	╝	220	DC	▄	252	FC	ⁿ
157	9D	¥	189	BD	╜	221	DD	▌	253	FD	²
158	9E	₧	190	BE	╛	222	DE	▐	254	FE	■
159	9F	ƒ	191	BF	┐	223	DF	▀	255	FF	

APPENDIX G

ASSEMBLERS, DEVELOPMENT RESOURCES, AND SUPPLIERS

This appendix provides various sources for PIC18 assemblers and trainers. In addition, it lists some suppliers for chips and other hardware needs. While these are all established products from well-known companies, neither the authors nor the publisher assumes responsibility for any problem that may arise with any of them. You are neither encouraged nor discouraged from purchasing any of the products mentioned; you must make your own judgment in evaluating the products. This list is simply provided as a service to the reader. It also must be noted that the list of products is by no means complete or exhaustive.

PIC18 assemblers

The PIC18 assembler is provided by Microchip and other companies. Some of the companies provide shareware versions of their products, which you can download from their Web sites. However, the size of code for these shareware versions is limited to a few KB. Figure G-1 lists some suppliers of assemblers.

PIC18 trainers

There are many companies that produce and market PIC18 trainers. Figure G-2 provides a list of some of them.

Microchip Corp.
www.microchip.com

Custom Computer Services Inc
www.ccsinfo.com

Figure G-1. Suppliers of Assemblers and Compilers

Microchip Corp.
www.microchip.com

www.MicroDigitalEd.com

Custom Computer Services Inc.
www.ccsinfo.com

RSR Electronics
www.elexp.com

Figure G-2. Trainer Suppliers

Parts Suppliers

Figure G-3 provides a list of suppliers for many electronics parts.

RSR Electronics
Electronix Express
365 Blair Road
Avenel, NJ 07001
Fax: (732) 381-1572
Mail Order: 1-800-972-2225
In New Jersey: (732) 381-8020
www.elexp.com

Altex Electronics
11342 IH-35 North
San Antonio, TX 78233
Fax: (210) 637-3264
Mail Order: 1-800-531-5369
www.altex.com

Digi-Key
1-800-344-4539 (1-800-DIGI-KEY)
Fax: (218) 681-3380
www.digikey.com

Radio Shack
www.radioshack.com

JDR Microdevices
1850 South 10th St.
San Jose, CA 95112-4108
Sales 1-800-538-5000
(408) 494-1400
Fax: 1-800-538-5005
Fax: (408) 494-1420
www.jdr.com

Mouser Electronics
958 N. Main St.
Mansfield, TX 76063
1-800-346-6873
www.mouser.com

Jameco Electronic
1355 Shoreway Road
Belmont, CA 94002-4100
1-800-831-4242
(415) 592-8097
Fax: 1-800-237-6948
Fax: (415) 592-2503
www.jameco.com

B. G. Micro
P. O. Box 280298
Dallas, TX 75228
1-800-276-2206 (orders only)
(972) 271-5546
Fax: (972) 271-2462
This is an excellent source of LCDs, ICs, keypads, etc.
www.bgmicro.com

Tanner Electronics
1100 Valwood Parkway, Suite #100
Carrollton, TX 75006
(972) 242-8702
www.tannerelectronics.com

Figure G-3. Electronics Suppliers

APPENDIX H

DATA SHEETS

25.0 INSTRUCTION SET SUMMARY

PIC18F2480/2580/4480/4580 devices incorporate the standard set of 75 PIC18 core instructions, as well as an extended set of 8 new instructions for the optimization of code that is recursive or that utilizes a software stack. The extended set is discussed later in this section.

25.1 Standard Instruction Set

The standard PIC18 instruction set adds many enhancements to the previous PICmicro® instruction sets, while maintaining an easy migration from these PICmicro instruction sets. Most instructions are a single program memory word (16 bits), but there are four instructions that require two program memory locations.

Each single-word instruction is a 16-bit word divided into an opcode, which specifies the instruction type and one or more operands, which further specify the operation of the instruction.

The instruction set is highly orthogonal and is grouped into four basic categories:

- **Byte-oriented** operations
- **Bit-oriented** operations
- **Literal** operations
- **Control** operations

The PIC18 instruction set summary in Table 25-2 lists **byte-oriented**, **bit-oriented**, **literal** and **control** operations. Table 25-1 shows the opcode field descriptions.

Most **byte-oriented** instructions have three operands:

1. The file register (specified by 'f')
2. The destination of the result (specified by 'd')
3. The accessed memory (specified by 'a')

The file register designator 'f' specifies which file register is to be used by the instruction. The destination designator 'd' specifies where the result of the operation is to be placed. If 'd' is zero, the result is placed in the WREG register. If 'd' is one, the result is placed in the file register specified in the instruction.

All **bit-oriented** instructions have three operands:

1. The file register (specified by 'f')
2. The bit in the file register (specified by 'b')
3. The accessed memory (specified by 'a')

The bit field designator 'b' selects the number of the bit affected by the operation, while the file register designator 'f' represents the number of the file in which the bit is located.

The **literal** instructions may use some of the following operands:

- A literal value to be loaded into a file register (specified by 'k')
- The desired FSR register to load the literal value into (specified by 'f')
- No operand required (specified by '—')

The **control** instructions may use some of the following operands:

- A program memory address (specified by 'n')
- The mode of the CALL or RETURN instructions (specified by 's')
- The mode of the table read and table write instructions (specified by 'm')
- No operand required (specified by '—')

All instructions are a single word, except for four double-word instructions. These instructions were made double word to contain the required information in 32 bits. In the second word, the 4 MSbs are '1's. If this second word is executed as an instruction (by itself), it will execute as a NOP.

All single word instructions are executed in a single instruction cycle, unless a conditional test is true or the program counter is changed as a result of the instruction. In these cases, the execution takes two instruction cycles with the additional instruction cycle(s) executed as a NOP.

The double word instructions execute in two instruction cycles.

One instruction cycle consists of four oscillator periods. Thus, for an oscillator frequency of 4 MHz, the normal instruction execution time is 1 μs. If a conditional test is true, or the program counter is changed as a result of an instruction, the instruction execution time is 2 μs. Two-word branch instructions (if true) would take 3 μs.

Figure 25-1 shows the general formats that the instructions can have. All examples use the convention 'nnh' to represent a hexadecimal number.

The Instruction Set Summary, shown in Table 25-2, lists the standard instructions recognized by the Microchip MPASM™ Assembler.

Section 25.1.1 "Standard Instruction Set" provides a description of each instruction.

TABLE 25-1: OPCODE FIELD DESCRIPTIONS

Field	Description
a	RAM access bit a = 0: RAM location in Access RAM (BSR register is ignored) a = 1: RAM bank is specified by BSR register
bbb	Bit address within an 8-bit file register (0 to 7).
BSR	Bank Select Register. Used to select the current RAM bank.
C, DC, Z, OV, N	ALU status bits: Carry, Digit Carry, Zero, Overflow, Negative.
d	Destination select bit d = 0: store result in WREG d = 1: store result in file register f
dest	Destination: either the WREG register or the specified register file location.
f	8-bit Register file address (00h to FFh), or 2-bit FSR designator (0h to 3h).
f_s	12-bit Register file address (000h to FFFh). This is the source address.
f_d	12-bit Register file address (000h to FFFh). This is the destination address.
GIE	Global Interrupt Enable bit.
k	Literal field, constant data or label (may be either an 8-bit, 12-bit or a 20-bit value).
label	Label name.
mm	The mode of the TBLPTR register for the table read and table write instructions. Only used with table read and table write instructions:
*	No change to register (such as TBLPTR with table reads and writes)
*+	Post-Increment register (such as TBLPTR with table reads and writes)
*-	Post-Decrement register (such as TBLPTR with table reads and writes)
+*	Pre-Increment register (such as TBLPTR with table reads and writes)
n	The relative address (2's complement number) for relative branch instructions or the direct address for Call/Branch and Return instructions.
PC	Program Counter.
PCL	Program Counter Low Byte.
PCH	Program Counter High Byte.
PCLATH	Program Counter High Byte Latch.
PCLATU	Program Counter Upper Byte Latch.
$\overline{PD}$	Power-down bit.
PRODH	Product of Multiply High Byte.
PRODL	Product of Multiply Low Byte.
s	Fast Call/Return mode select bit s = 0: do not update into/from shadow registers s = 1: certain registers loaded into/from shadow registers (Fast mode)
TBLPTR	21-bit Table Pointer (points to a Program Memory location).
TABLAT	8-bit Table Latch.
$\overline{TO}$	Time-out bit.
TOS	Top-of-Stack.
u	Unused or unchanged.
WDT	Watchdog Timer.
WREG	Working register (accumulator).
x	Don't care ('0' or '1'). The assembler will generate code with x = 0. It is the recommended form of use for compatibility with all Microchip software tools.
z_s	7-bit offset value for indirect addressing of register files (source).
z_d	7-bit offset value for indirect addressing of register files (destination).
{ }	Optional argument.
[text]	Indicates an indexed address.
(text)	The contents of text.
[expr]<n>	Specifies bit n of the register indicated by the pointer expr.
→	Assigned to.
< >	Register bit field.
∈	In the set of.
italics	User defined term (font is Courier).

FIGURE 25-1: GENERAL FORMAT FOR INSTRUCTIONS

Instruction format	Example Instruction
Byte-oriented file register operations	
Bits 15–10: OPCODE; bit 9: d; bit 8: a; bits 7–0: f (FILE #)	ADDWF MYREG, W, B
d = 0 for result destination to be WREG register d = 1 for result destination to be file register (f) a = 0 to force Access Bank a = 1 for BSR to select bank f = 8-bit file register address	
Byte to Byte move operations (2-word)	
Bits 15–12: OPCODE; bits 11–0: f (Source FILE #)	MOVFF MYREG1, MYREG2
Bits 15–12: 1111; bits 11–0: f (Destination FILE #)	
f = 12-bit file register address	
Bit-oriented file register operations	
Bits 15–12: OPCODE; bits 11–9: b (BIT #); bit 8: a; bits 7–0: f (FILE #)	BSF MYREG, bit, B
b = 3-bit position of bit in file register (f) a = 0 to force Access Bank a = 1 for BSR to select bank f = 8-bit file register address	
Literal operations	
Bits 15–8: OPCODE; bits 7–0: k (literal)	MOVLW 7Fh
k = 8-bit immediate value	
Control operations	
CALL, GOTO **and Branch** operations	
Bits 15–8: OPCODE; bits 7–0: n<7:0> (literal)	GOTO Label
Bits 15–12: 1111; bits 11–0: n<19:8> (literal)	
n = 20-bit immediate value	
Bits 15–9: OPCODE; bit 8: S; bits 7–0: n<7:0> (literal)	CALL MYFUNC
Bits 15–12: 1111; bits 11–0: n<19:8> (literal)	
S = Fast bit	
Bits 15–11: OPCODE; bits 10–0: n<10:0> (literal)	BRA MYFUNC
Bits 15–8: OPCODE; bits 7–0: n<7:0> (literal)	BC MYFUNC

TABLE 25-2: PIC18FXXXX INSTRUCTION SET

Mnemonic,	Operands	Description	Cycles	16-Bit Instruction Word MSb			LSb	Status Affected	Notes
BYTE-ORIENTED OPERATIONS									
ADDWF	f, d, a	Add WREG and f	1	0010	01da	ffff	ffff	C, DC, Z, OV, N	1, 2
ADDWFC	f, d, a	Add WREG and Carry bit to f	1	0010	00da	ffff	ffff	C, DC, Z, OV, N	1, 2
ANDWF	f, d, a	AND WREG with f	1	0001	01da	ffff	ffff	Z, N	1,2
CLRF	f, a	Clear f	1	0110	101a	ffff	ffff	Z	2
COMF	f, d, a	Complement f	1	0001	11da	ffff	ffff	Z, N	1, 2
CPFSEQ	f, a	Compare f with WREG, skip =	1 (2 or 3)	0110	001a	ffff	ffff	None	4
CPFSGT	f, a	Compare f with WREG, skip >	1 (2 or 3)	0110	010a	ffff	ffff	None	4
CPFSLT	f, a	Compare f with WREG, skip <	1 (2 or 3)	0110	000a	ffff	ffff	None	1, 2
DECF	f, d, a	Decrement f	1	0000	01da	ffff	ffff	C, DC, Z, OV, N	1, 2, 3, 4
DECFSZ	f, d, a	Decrement f, Skip if 0	1 (2 or 3)	0010	11da	ffff	ffff	None	1, 2, 3, 4
DCFSNZ	f, d, a	Decrement f, Skip if Not 0	1 (2 or 3)	0100	11da	ffff	ffff	None	1, 2
INCF	f, d, a	Increment f	1	0010	10da	ffff	ffff	C, DC, Z, OV, N	1, 2, 3, 4
INCFSZ	f, d, a	Increment f, Skip if 0	1 (2 or 3)	0011	11da	ffff	ffff	None	4
INFSNZ	f, d, a	Increment f, Skip if Not 0	1 (2 or 3)	0100	10da	ffff	ffff	None	1, 2
IORWF	f, d, a	Inclusive OR WREG with f	1	0001	00da	ffff	ffff	Z, N	1, 2
MOVF	f, d, a	Move f	1	0101	00da	ffff	ffff	Z, N	1
MOVFF	f_s, f_d	Move f_s (source) to 1st word	2	1100	ffff	ffff	ffff	None	
		f_d (destination) 2nd word		1111	ffff	ffff	ffff		
MOVWF	f, a	Move WREG to f	1	0110	111a	ffff	ffff	None	
MULWF	f, a	Multiply WREG with f	1	0000	001a	ffff	ffff	None	1, 2
NEGF	f, a	Negate f	1	0110	110a	ffff	ffff	C, DC, Z, OV, N	
RLCF	f, d, a	Rotate Left f through Carry	1	0011	01da	ffff	ffff	C, Z, N	1, 2
RLNCF	f, d, a	Rotate Left f (No Carry)	1	0100	01da	ffff	ffff	Z, N	
RRCF	f, d, a	Rotate Right f through Carry	1	0011	00da	ffff	ffff	C, Z, N	
RRNCF	f, d, a	Rotate Right f (No Carry)	1	0100	00da	ffff	ffff	Z, N	
SETF	f, a	Set f	1	0110	100a	ffff	ffff	None	1, 2
SUBFWB	f, d, a	Subtract f from WREG with borrow	1	0101	01da	ffff	ffff	C, DC, Z, OV, N	
SUBWF	f, d, a	Subtract WREG from f	1	0101	11da	ffff	ffff	C, DC, Z, OV, N	1, 2
SUBWFB	f, d, a	Subtract WREG from f with borrow	1	0101	10da	ffff	ffff	C, DC, Z, OV, N	
SWAPF	f, d, a	Swap nibbles in f	1	0011	10da	ffff	ffff	None	4
TSTFSZ	f, a	Test f, skip if 0	1 (2 or 3)	0110	011a	ffff	ffff	None	1, 2
XORWF	f, d, a	Exclusive OR WREG with f	1	0001	10da	ffff	ffff	Z, N	

Note 1: When a Port register is modified as a function of itself (e.g., `MOVF PORTB, 1, 0`), 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.

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`.

4: Some instructions are two-word instructions. The second word of these instructions will be executed as a `NOP` unless the first word of the instruction retrieves the information embedded in these 16 bits. This ensures that all program memory locations have a valid instruction.

5: If the table write starts the write cycle to internal memory, the write will continue until terminated.

 © 2004 Microchip Technology Inc.

TABLE 25-2: PIC18FXXXX INSTRUCTION SET (CONTINUED)

Mnemonic, Operands		Description	Cycles	16-Bit Instruction Word				Status Affected	Notes
				MSb			LSb		
BIT-ORIENTED OPERATIONS									
BCF	f, b, a	Bit Clear f	1	1001	bbba	ffff	ffff	None	1, 2
BSF	f, b, a	Bit Set f	1	1000	bbba	ffff	ffff	None	1, 2
BTFSC	f, b, a	Bit Test f, Skip if Clear	1 (2 or 3)	1011	bbba	ffff	ffff	None	3, 4
BTFSS	f, b, a	Bit Test f, Skip if Set	1 (2 or 3)	1010	bbba	ffff	ffff	None	3, 4
BTG	f, d, a	Bit Toggle f	1	0111	bbba	ffff	ffff	None	1, 2
CONTROL OPERATIONS									
BC	n	Branch if Carry	1 (2)	1110	0010	nnnn	nnnn	None	
BN	n	Branch if Negative	1 (2)	1110	0110	nnnn	nnnn	None	
BNC	n	Branch if Not Carry	1 (2)	1110	0011	nnnn	nnnn	None	
BNN	n	Branch if Not Negative	1 (2)	1110	0111	nnnn	nnnn	None	
BNOV	n	Branch if Not Overflow	1 (2)	1110	0101	nnnn	nnnn	None	
BNZ	n	Branch if Not Zero	1 (2)	1110	0001	nnnn	nnnn	None	
BOV	n	Branch if Overflow	1 (2)	1110	0100	nnnn	nnnn	None	
BRA	n	Branch Unconditionally	2	1101	0nnn	nnnn	nnnn	None	
BZ	n	Branch if Zero	1 (2)	1110	0000	nnnn	nnnn	None	
CALL	n, s	Call subroutine 1st word	2	1110	110s	kkkk	kkkk	None	
		2nd word		1111	kkkk	kkkk	kkkk		
CLRWDT	—	Clear Watchdog Timer	1	0000	0000	0000	0100	$\overline{TO}$, $\overline{PD}$	
DAW		Decimal Adjust WREG	1	0000	0000	0000	0111	C	
GOTO	n	Go to address 1st word	2	1110	1111	kkkk	kkkk	None	
		2nd word		1111	kkkk	kkkk	kkkk		
NOP	—	No Operation	1	0000	0000	0000	0000	None	
NOP		No Operation	1	1111	xxxx	xxxx	xxxx	None	4
POP	—	Pop top of return stack (TOS)	1	0000	0000	0000	0110	None	
PUSH	—	Push top of return stack (TOS)	1	0000	0000	0000	0101	None	
RCALL	n	Relative Call	2	1101	1nnn	nnnn	nnnn	None	
RESET		Software device Reset	1	0000	0000	1111	1111	All	
RETFIE	s	Return from interrupt enable	2	0000	0000	0001	000s	GIE/GIEH, PEIE/GIEL	
RETLW	k	Return with literal in WREG	2	0000	1100	kkkk	kkkk	None	
RETURN	s	Return from Subroutine	2	0000	0000	0001	001s	None	
SLEEP		Go into Standby mode	1	0000	0000	0000	0011	TO, PD	

Note 1: When a Port register is modified as a function of itself (e.g., MOVF PORTB, 1, 0), 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.

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.

4: Some instructions are two-word instructions. The second word of these instructions will be executed as a NOP unless the first word of the instruction retrieves the information embedded in these 16 bits. This ensures that all program memory locations have a valid instruction.

5: If the table write starts the write cycle to internal memory, the write will continue until terminated.

TABLE 25-2: PIC18FXXXX INSTRUCTION SET (CONTINUED)

Mnemonic, Operands		Description	Cycles	16-Bit Instruction Word MSb			LSb	Status Affected	Notes
LITERAL OPERATIONS									
ADDLW	k	Add literal and WREG	1	0000	1111	kkkk	kkkk	C, DC, Z, OV, N	
ANDLW	k	AND literal with WREG	1	0000	1011	kkkk	kkkk	Z, N	
IORLW	k	Inclusive OR literal with WREG	1	0000	1001	kkkk	kkkk	Z, N	
LFSR	f, k	Move literal (12-bit) 2nd word	2	1110	1110	00ff	kkkk	None	
		to FSR(f) 1st word		1111	0000	kkkk	kkkk		
MOVLB	k	Move literal to BSR<3:0>	1	0000	0001	0000	kkkk	None	
MOVLW	k	Move literal to WREG	1	0000	1110	kkkk	kkkk	None	
MULLW	k	Multiply literal with WREG	1	0000	1101	kkkk	kkkk	None	
RETLW	k	Return with literal in WREG	2	0000	1100	kkkk	kkkk	None	
SUBLW	k	Subtract WREG from literal	1	0000	1000	kkkk	kkkk	C, DC, Z, OV, N	
XORLW	k	Exclusive OR literal with WREG	1	0000	1010	kkkk	kkkk	Z, N	
DATA MEMORY ↔ PROGRAM MEMORY OPERATIONS									
TBLRD*		Table Read	2	0000	0000	0000	1000	None	
TBLRD*+		Table Read with post-increment		0000	0000	0000	1001	None	
TBLRD*-		Table Read with post-decrement		0000	0000	0000	1010	None	
TBLRD+*		Table Read with pre-increment		0000	0000	0000	1011	None	
TBLWT*		Table Write	2	0000	0000	0000	1100	None	5
TBLWT*+		Table Write with post-increment		0000	0000	0000	1101	None	5
TBLWT*-		Table Write with post-decrement		0000	0000	0000	1110	None	5
TBLWT+*		Table Write with pre-increment		0000	0000	0000	1111	None	5

Note 1: When a Port register is modified as a function of itself (e.g., `MOVF PORTB, 1, 0`), 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.

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`.

4: Some instructions are two-word instructions. The second word of these instructions will be executed as a `NOP` unless the first word of the instruction retrieves the information embedded in these 16 bits. This ensures that all program memory locations have a valid instruction.

5: If the table write starts the write cycle to internal memory, the write will continue until terminated.

 © 2004 Microchip Technology Inc.

25.1.1 STANDARD INSTRUCTION SET

ADDLW — ADD Literal to W

Syntax:	ADDLW k
Operands:	0 ≤ k ≤ 255
Operation:	(W) + k → W
Status Affected:	N, OV, C, DC, Z
Encoding:	0000 | 1111 | kkkk | kkkk
Description:	The contents of W are added to the 8-bit literal 'k' and the result is placed in W.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'	Process Data	Write to W

Example: `ADDLW 15h`

Before Instruction
W = 10h
After Instruction
W = 25h

ADDWF — ADD W to f

Syntax:	ADDWF f {,d {,a}}
Operands:	0 ≤ f ≤ 255 d ∈ [0,1] a ∈ [0,1]
Operation:	(W) + (f) → dest
Status Affected:	N, OV, C, DC, Z
Encoding:	0010 | 01da | ffff | ffff
Description:	Add W to register 'f'. If 'd' is '0', the result is stored in W. If 'd' is '1', the result is stored back in register 'f' (default). If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the GPR bank (default). If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh). See **Section 25.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example: `ADDWF REG, 0, 0`

Before Instruction
W = 17h
REG = 0C2h
After Instruction
W = 0D9h
REG = 0C2h

Note:	All PIC18 instructions may take an optional label argument preceding the instruction mnemonic for use in symbolic addressing. If a label is used, the instruction format then becomes: {label} instruction argument(s).

ADDWFC **ADD W and Carry bit to f**

Syntax: ADDWFC f {,d {,a}}

Operands: $0 \le f \le 255$
$d \in [0,1]$
$a \in [0,1]$

Operation: (W) + (f) + (C) → dest

Status Affected: N,OV, C, DC, Z

Encoding:

0010	00da	ffff	ffff

Description: Add W, the Carry flag and data memory location 'f'. If 'd' is '0', the result is placed in W. If 'd' is '1', the result is placed in data memory location 'f'.
If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the GPR bank (default).
If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever $f \le 95$ (5Fh). See **Section 25.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example: ADDWFC REG, 0, 1

Before Instruction
Carry bit = 1
REG = 02h
W = 4Dh
After Instruction
Carry bit = 0
REG = 02h
W = 50h

ANDLW **AND Literal with W**

Syntax: ANDLW k

Operands: $0 \le k \le 255$

Operation: (W) .AND. k → W

Status Affected: N, Z

Encoding:

0000	1011	kkkk	kkkk

Description: The contents of W are ANDed with the 8-bit literal 'k'. The result is placed in W.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'	Process Data	Write to W

Example: ANDLW 05Fh

Before Instruction
W = A3h
After Instruction
W = 03h

 © 2004 Microchip Technology Inc.

ANDWF — AND W with f

Syntax:	ANDWF f {,d {,a}}
Operands:	0 ≤ f ≤ 255 d ∈ [0,1] a ∈ [0,1]
Operation:	(W) .AND. (f) → dest
Status Affected:	N, Z
Encoding:	0001 / 01da / ffff / ffff
Description:	The contents of W are AND'ed with register 'f'. If 'd' is '0', the result is stored in W. If 'd' is '1', the result is stored back in register 'f' (default). If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the GPR bank (default). If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh). See **Section 26.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example: `ANDWF REG, 0, 0`

Before Instruction
W = 17h
REG = C2h
After Instruction
W = 02h
REG = C2h

BC — Branch if Carry

Syntax:	BC n
Operands:	-128 ≤ n ≤ 127
Operation:	if Carry bit is '1' (PC) + 2 + 2n → PC
Status Affected:	None
Encoding:	1110 / 0010 / nnnn / nnnn
Description:	If the Carry bit is '1', then the program will branch. The 2's complement number '2n' is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be PC + 2 + 2n. This instruction is then a two-cycle instruction.
Words:	1
Cycles:	1(2)

Q Cycle Activity:

If Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	Write to PC
No operation	No operation	No operation	No operation

If No Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	No operation

Example: `HERE BC 5`

Before Instruction
PC = address (HERE)
After Instruction
If Carry = 1;
PC = address (HERE + 12)
If Carry = 0;
PC = address (HERE + 2)

BCF — Bit Clear f

Syntax:	BCF f, b {,a}
Operands:	0 ≤ f ≤ 255 0 ≤ b ≤ 7 a ∈ [0,1]
Operation:	0 → f<b>
Status Affected:	None
Encoding:	1001 | bbba | ffff | ffff
Description:	Bit 'b' in register 'f' is cleared. If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the GPR bank (default). If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh). See **Section 25.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write register 'f'

Example: `BCF FLAG_REG, 7, 0`

Before Instruction
FLAG_REG = C7h
After Instruction
FLAG_REG = 47h

BN — Branch if Negative

Syntax:	BN n
Operands:	-128 ≤ n ≤ 127
Operation:	if Negative bit is '1' (PC) + 2 + 2n → PC
Status Affected:	None
Encoding:	1110 | 0110 | nnnn | nnnn
Description:	If the Negative bit is '1', then the program will branch. The 2's complement number '2n' is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be PC + 2 + 2n. This instruction is then a two-cycle instruction.
Words:	1
Cycles:	1(2)

Q Cycle Activity:

If Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	Write to PC
No operation	No operation	No operation	No operation

If No Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	No operation

Example: `HERE BN Jump`

Before Instruction
PC = address (HERE)
After Instruction
If Negative = 1;
PC = address (Jump)
If Negative = 0;
PC = address (HERE + 2)

© 2004 Microchip Technology Inc.

BNC **Branch if Not Carry**

Syntax:	BNC n
Operands:	$-128 \le n \le 127$
Operation:	if Carry bit is '0' (PC) + 2 + 2n → PC
Status Affected:	None
Encoding:	1110 | 0011 | nnnn | nnnn
Description:	If the Carry bit is '0', then the program will branch. The 2's complement number '2n' is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be PC + 2 + 2n. This instruction is then a two-cycle instruction.
Words:	1
Cycles:	1(2)

Q Cycle Activity:

If Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	Write to PC
No operation	No operation	No operation	No operation

If No Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	No operation

Example: HERE BNC Jump

Before Instruction
PC = address (HERE)
After Instruction
If Carry = 0;
PC = address (Jump)
If Carry = 1;
PC = address (HERE + 2)

BNN **Branch if Not Negative**

Syntax:	BNN n
Operands:	$-128 \le n \le 127$
Operation:	if Negative bit is '0' (PC) + 2 + 2n → PC
Status Affected:	None
Encoding:	1110 | 0111 | nnnn | nnnn
Description:	If the Negative bit is '0', then the program will branch. The 2's complement number '2n' is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be PC + 2 + 2n. This instruction is then a two-cycle instruction.
Words:	1
Cycles:	1(2)

Q Cycle Activity:

If Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	Write to PC
No operation	No operation	No operation	No operation

If No Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	No operation

Example: HERE BNN Jump

Before Instruction
PC = address (HERE)
After Instruction
If Negative = 0;
PC = address (Jump)
If Negative = 1;
PC = address (HERE + 2)

BNOV — Branch if Not Overflow

Syntax:	BNOV n
Operands:	-128 ≤ n ≤ 127
Operation:	if Overflow bit is '0' (PC) + 2 + 2n → PC
Status Affected:	None
Encoding:	1110 | 0101 | nnnn | nnnn
Description:	If the Overflow bit is '0', then the program will branch. The 2's complement number '2n' is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be PC + 2 + 2n. This instruction is then a two-cycle instruction.
Words:	1
Cycles:	1(2)

Q Cycle Activity:

If Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	Write to PC
No operation	No operation	No operation	No operation

If No Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	No operation

Example: HERE BNOV Jump

Before Instruction
PC = address (HERE)

After Instruction
If Overflow = 0;
PC = address (Jump)
If Overflow = 1;
PC = address (HERE + 2)

BNZ — Branch if Not Zero

Syntax:	BNZ n
Operands:	-128 ≤ n ≤ 127
Operation:	if Zero bit is '0' (PC) + 2 + 2n → PC
Status Affected:	None
Encoding:	1110 | 0001 | nnnn | nnnn
Description:	If the Zero bit is '0', then the program will branch. The 2's complement number '2n' is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be PC + 2 + 2n. This instruction is then a two-cycle instruction.
Words:	1
Cycles:	1(2)

Q Cycle Activity:

If Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	Write to PC
No operation	No operation	No operation	No operation

If No Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	No operation

Example: HERE BNZ Jump

Before Instruction
PC = address (HERE)

After Instruction
If Zero = 0;
PC = address (Jump)
If Zero = 1;
PC = address (HERE + 2)

 © 2004 Microchip Technology Inc.

BRA — Unconditional Branch

Syntax:	BRA n
Operands:	-1024 ≤ n ≤ 1023
Operation:	(PC) + 2 + 2n → PC
Status Affected:	None
Encoding:	1101 0nnn nnnn nnnn
Description:	Add the 2's complement number '2n' to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be PC + 2 + 2n. This instruction is a two-cycle instruction.
Words:	1
Cycles:	2

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	Write to PC
No operation	No operation	No operation	No operation

Example: HERE BRA Jump

Before Instruction
PC = address (HERE)
After Instruction
PC = address (Jump)

BSF — Bit Set f

Syntax:	BSF f, b {,a}
Operands:	0 ≤ f ≤ 255 0 ≤ b ≤ 7 a ∈ [0,1]
Operation:	1 → f<b>
Status Affected:	None
Encoding:	1000 bbba ffff ffff
Description:	Bit 'b' in register 'f' is set. If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the GPR bank (default). If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh). See **Section 25.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write register 'f'

Example: BSF FLAG_REG, 7, 1

Before Instruction
FLAG_REG = 0Ah
After Instruction
FLAG_REG = 8Ah

BTFSC — **Bit Test File, Skip if Clear**

Syntax:	BTFSC f, b {,a}
Operands:	0 ≤ f ≤ 255 0 < b < 7 a ∈ [0,1]
Operation:	skip if (f<b>) = 0
Status Affected:	None
Encoding:	1011 | bbba | ffff | ffff
Description:	If bit 'b' in register 'f' is '0', then the next instruction is skipped. If bit 'b' is '0', then the next instruction fetched during the current instruction execution is discarded and a NOP is executed instead, making this a two-cycle instruction. If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the GPR bank (default). If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh). See **Section 25.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.
Words:	1
Cycles:	1(2) **Note:** 3 cycles if skip and followed by a 2-word instruction.

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	No operation

If skip:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation

If skip and followed by 2-word instruction:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation
No operation	No operation	No operation	No operation

Example:

```
HERE    BTFSC   FLAG, 1, 0
FALSE   :
TRUE    :
```

Before Instruction
PC = address (HERE)
After Instruction
If FLAG<1> = 0;
PC = address (TRUE)
If FLAG<1> = 1;
PC = address (FALSE)

BTFSS — **Bit Test File, Skip if Set**

Syntax:	BTFSS f, b {,a}
Operands:	0 ≤ f ≤ 255 0 < b < 7 a ∈ [0,1]
Operation:	skip if (f<b>) = 1
Status Affected:	None
Encoding:	1010 | bbba | ffff | ffff
Description:	If bit 'b' in register 'f' is '1', then the next instruction is skipped. If bit 'b' is '1', then the next instruction fetched during the current instruction execution is discarded and a NOP is executed instead, making this a two-cycle instruction. If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the GPR bank (default). If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh). See **Section 25.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.
Words:	1
Cycles:	1(2) **Note:** 3 cycles if skip and followed by a 2-word instruction.

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	No operation

If skip:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation

If skip and followed by 2-word instruction:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation
No operation	No operation	No operation	No operation

Example:

```
HERE    BTFSS   FLAG, 1, 0
FALSE   :
TRUE    :
```

Before Instruction
PC = address (HERE)
After Instruction
If FLAG<1> = 0;
PC = address (FALSE)
If FLAG<1> = 1;
PC = address (TRUE)

 © 2004 Microchip Technology Inc.

BTG — Bit Toggle f

Syntax: BTG f, b {,a}

Operands: 0 ≤ f ≤ 255
0 ≤ b ≤ 7
a ∈ [0,1]

Operation: $(\overline{f<b>}) \rightarrow f<b>$

Status Affected: None

Encoding:

0111	bbba	ffff	ffff

Description: Bit 'b' in data memory location 'f' is inverted.
If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the GPR bank (default).
If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh). See **Section 25.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write register 'f'

Example: `BTG PORTC, 4, 0`

Before Instruction:
PORTC = 0111 0101 [75h]
After Instruction:
PORTC = 0110 0101 [65h]

BOV — Branch if Overflow

Syntax: BOV n

Operands: -128 ≤ n ≤ 127

Operation: if Overflow bit is '1'
(PC) + 2 + 2n → PC

Status Affected: None

Encoding:

1110	0100	nnnn	nnnn

Description: If the Overflow bit is '1', then the program will branch.
The 2's complement number '2n' is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be PC + 2 + 2n. This instruction is then a two-cycle instruction.

Words: 1

Cycles: 1(2)

Q Cycle Activity:
If Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	Write to PC
No operation	No operation	No operation	No operation

If No Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	No operation

Example: `HERE BOV Jump`

Before Instruction
PC = address (HERE)
After Instruction
If Overflow = 1;
PC = address (Jump)
If Overflow = 0;
PC = address (HERE + 2)

BZ — Branch if Zero

Syntax: BZ n

Operands: $-128 \le n \le 127$

Operation: if Zero bit is '1' (PC) + 2 + 2n → PC

Status Affected: None

Encoding:

1110	0000	nnnn	nnnn

Description: If the Zero bit is '1', then the program will branch.

The 2's complement number '2n' is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be PC + 2 + 2n. This instruction is then a two-cycle instruction.

Words: 1

Cycles: 1(2)

Q Cycle Activity:

If Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	Write to PC
No operation	No operation	No operation	No operation

If No Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	No operation

Example: HERE BZ Jump

Before Instruction
PC = address (HERE)

After Instruction
If Zero = 1;
PC = address (Jump)
If Zero = 0;
PC = address (HERE + 2)

CALL — Subroutine Call

Syntax: CALL k {,s}

Operands: $0 \le k \le 1048575$
$s \in [0,1]$

Operation: (PC) + 4 → TOS,
k → PC<20:1>,
if s = 1
(W) → WS,
(Status) → STATUSS,
(BSR) → BSRS

Status Affected: None

Encoding:

1st word (k<7:0>)	1110	110s	$k_7$kkk	kkkk_0
2nd word(k<19:8>)	1111	k_{19}kkk	kkkk	kkkk_8

Description: Subroutine call of entire 2-Mbyte memory range. First, return address (PC + 4) is pushed onto the return stack. If 's' = 1, the W, Status and BSR registers are also pushed into their respective shadow registers, WS, STATUSS and BSRS. If 's' = 0, no update occurs (default). Then, the 20-bit value 'k' is loaded into PC<20:1>. CALL is a two-cycle instruction.

Words: 2

Cycles: 2

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'<7:0>,	Push PC to stack	Read literal 'k'<19:8>, Write to PC
No operation	No operation	No operation	No operation

Example: HERE CALL THERE, 1

Before Instruction
PC = address (HERE)

After Instruction
PC = address (THERE)
TOS = address (HERE + 4)
WS = W
BSRS = BSR
STATUSS = Status

 © 2004 Microchip Technology Inc.

CLRF — Clear f

Syntax: CLRF f {,a}

Operands: $0 \le f \le 255$
$a \in [0,1]$

Operation: 000h → f
1 → Z

Status Affected: Z

Encoding:

0110	101a	ffff	ffff

Description: Clears the contents of the specified register.
If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the GPR bank (default).
If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever $f \le 95$ (5Fh). See **Section 25.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write register 'f'

Example: CLRF FLAG_REG, 1

Before Instruction
FLAG_REG = 5Ah
After Instruction
FLAG_REG = 00h

CLRWDT — Clear Watchdog Timer

Syntax: CLRWDT

Operands: None

Operation: 000h → WDT,
000h → WDT postscaler,
1 → $\overline{TO}$,
1 → $\overline{PD}$

Status Affected: $\overline{TO}$, $\overline{PD}$

Encoding:

0000	0000	0000	0100

Description: CLRWDT instruction resets the Watchdog Timer. It also resets the postscaler of the WDT. Status bits $\overline{TO}$ and $\overline{PD}$ are set.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	No operation	Process Data	No operation

Example: CLRWDT

Before Instruction
WDT Counter = ?
After Instruction
WDT Counter = 00h
WDT Postscaler = 0
$\overline{TO}$ = 1
$\overline{PD}$ = 1

COMF — Complement f

Syntax:	COMF f {,d {,a}}
Operands:	0 ≤ f ≤ 255 d ∈ [0,1] a ∈ [0,1]
Operation:	$(\overline{f}) \rightarrow$ dest
Status Affected:	N, Z
Encoding:	0001 | 11da | ffff | ffff
Description:	The contents of register 'f' are complemented. If 'd' is '1', the result is stored in W. If 'd' is '0', the result is stored back in register 'f' (default). If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the GPR bank (default). If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh). See **Section 25.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example: `COMF REG, 0, 0`

Before Instruction
REG = 13h
After Instruction
REG = 13h
W = ECh

CPFSEQ — Compare f with W, Skip if f = W

Syntax:	CPFSEQ f {,a}
Operands:	0 ≤ f ≤ 255 a ∈ [0,1]
Operation:	(f) – (W), skip if (f) = (W) (unsigned comparison)
Status Affected:	None
Encoding:	0110 | 001a | ffff | ffff
Description:	Compares the contents of data memory location 'f' to the contents of W by performing an unsigned subtraction. If 'f' = W, then the fetched instruction is discarded and a NOP is executed instead, making this a two-cycle instruction. If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the GPR bank (default). If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh). See **Section 25.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.
Words:	1
Cycles:	1(2) **Note:** 3 cycles if skip and followed by a 2-word instruction.

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	No operation

If skip:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation

If skip and followed by 2-word instruction:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation
No operation	No operation	No operation	No operation

Example:
```
HERE     CPFSEQ REG, 0
NEQUAL   :
EQUAL    :
```

Before Instruction
PC Address = HERE
W = ?
REG = ?
After Instruction
If REG = W;
PC = Address (EQUAL)
If REG ≠ W;
PC = Address (NEQUAL)

 © 2004 Microchip Technology Inc.

CPFSGT — Compare f with W, Skip if f > W

Syntax:	CPFSGT f {,a}
Operands:	0 ≤ f ≤ 255 a ∈ [0,1]
Operation:	(f) – (W), skip if (f) > (W) (unsigned comparison)
Status Affected:	None
Encoding:	0110 | 010a | ffff | ffff
Description:	Compares the contents of data memory location 'f' to the contents of the W by performing an unsigned subtraction. If the contents of 'f' are greater than the contents of WREG, then the fetched instruction is discarded and a NOP is executed instead, making this a two-cycle instruction. If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the GPR bank (default). If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh). See **Section 25.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.
Words:	1
Cycles:	1(2) **Note:** 3 cycles if skip and followed by a 2-word instruction.

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	No operation

If skip:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation

If skip and followed by 2-word instruction:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation
No operation	No operation	No operation	No operation

Example:

```
HERE      CPFSGT REG, 0
NGREATER  :
GREATER   :
```

Before Instruction
- PC = Address (HERE)
- W = ?

After Instruction
- If REG > W;
  - PC = Address (GREATER)
- If REG ≤ W;
  - PC = Address (NGREATER)

CPFSLT — Compare f with W, Skip if f < W

Syntax:	CPFSLT f {,a}
Operands:	0 ≤ f ≤ 255 a ∈ [0,1]
Operation:	(f) – (W), skip if (f) < (W) (unsigned comparison)
Status Affected:	None
Encoding:	0110 | 000a | ffff | ffff
Description:	Compares the contents of data memory location 'f' to the contents of W by performing an unsigned subtraction. If the contents of 'f' are less than the contents of W, then the fetched instruction is discarded and a NOP is executed instead, making this a two-cycle instruction. If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the GPR bank (default).
Words:	1
Cycles:	1(2) **Note:** 3 cycles if skip and followed by a 2-word instruction.

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	No operation

If skip:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation

If skip and followed by 2-word instruction:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation
No operation	No operation	No operation	No operation

Example:

```
HERE    CPFSLT REG, 1
NLESS   :
LESS    :
```

Before Instruction
- PC = Address (HERE)
- W = ?

After Instruction
- If REG < W;
- PC = Address (LESS)
- If REG ≥ W;
- PC = Address (NLESS)

DAW	Decimal Adjust W Register
Syntax:	DAW
Operands:	None

Operation:	If [W<3:0> >9] or [DC = 1] then
	(W<3:0>) + 6 → W<3:0>;
	else
	(W<3:0>) → W<3:0>;
	
	If [W<7:4> >9] or [C = 1] then
	(W<7:4>) + 6 → W<7:4>;
	C = 1;
	else
	(W<7:4>) → W<7:4>;

Status Affected:	C
Encoding:	0000	0000	0000	0111
Description:	`DAW` adjusts the eight-bit value in W, resulting from the earlier addition of two variables (each in packed BCD format) and produces a correct packed BCD result.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register W	Process Data	Write W

Example 1:

```
DAW
```

Before Instruction
W = A5h
C = 0
DC = 0
After Instruction
W = 05h
C = 1
DC = 0

Example 2:

Before Instruction
W = CEh
C = 0
DC = 0
After Instruction
W = 34h
C = 1
DC = 0

DECF	Decrement f
Syntax:	DECF f {,d {,a}}
Operands:	$0 \le f \le 255$ $d \in [0,1]$ $a \in [0,1]$
Operation:	(f) – 1 → dest
Status Affected:	C, DC, N, OV, Z
Encoding:	0000	01da	ffff	ffff
Description:	Decrement register 'f'. If 'd' is '0', the result is stored in W. If 'd' is '1', the result is stored back in register 'f' (default). If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the GPR bank (default). If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh). See **Section 25.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example:

```
DECF    CNT,   1,  0
```

Before Instruction
CNT = 01h
Z = 0
After Instruction
CNT = 00h
Z = 1

© 2004 Microchip Technology Inc.

DECFSZ — Decrement f, Skip if 0

Syntax: DECFSZ f {,d {,a}}

Operands: $0 \le f \le 255$; $d \in [0,1]$; $a \in [0,1]$

Operation: (f) – 1 → dest, skip if result = 0

Status Affected: None

Encoding:

0010	11da	ffff	ffff

Description: The contents of register 'f' are decremented. If 'd' is '0', the result is placed in W. If 'd' is '1', the result is placed back in register 'f' (default).

If the result is '0', the next instruction which is already fetched is discarded and a NOP is executed instead, making it a two-cycle instruction.

If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the GPR bank (default).

If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever $f \le 95$ (5Fh). See **Section 25.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.

Words: 1

Cycles: 1(2)

Note: 3 cycles if skip and followed by a 2-word instruction.

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

If skip:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation

If skip and followed by 2-word instruction:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation
No operation	No operation	No operation	No operation

Example:

```
HERE      DECFSZ   CNT, 1, 1
          GOTO     LOOP
CONTINUE
```

Before Instruction
- PC = Address (HERE)

After Instruction
- CNT = CNT – 1
- If CNT = 0;
  - PC = Address (CONTINUE)
- If CNT ≠ 0;
  - PC = Address (HERE + 2)

DCFSNZ — Decrement f, Skip if not 0

Syntax: DCFSNZ f {,d {,a}}

Operands: $0 \le f \le 255$; $d \in [0,1]$; $a \in [0,1]$

Operation: (f) – 1 → dest, skip if result ≠ 0

Status Affected: None

Encoding:

0100	11da	ffff	ffff

Description: The contents of register 'f' are decremented. If 'd' is '0', the result is placed in W. If 'd' is '1', the result is placed back in register 'f' (default).

If the result is not '0', the next instruction which is already fetched is discarded and a NOP is executed instead, making it a two-cycle instruction.

If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the GPR bank (default).

If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever $f \le 95$ (5Fh). See **Section 25.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.

Words: 1

Cycles: 1(2)

Note: 3 cycles if skip and followed by a 2-word instruction.

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

If skip:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation

If skip and followed by 2-word instruction:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation
No operation	No operation	No operation	No operation

Example:

```
HERE      DCFSNZ   TEMP, 1, 0
ZERO      :
NZERO     :
```

Before Instruction
- TEMP = ?

After Instruction
- TEMP = TEMP – 1,
- If TEMP = 0;
  - PC = Address (ZERO)
- If TEMP ≠ 0;
  - PC = Address (NZERO)

GOTO — Unconditional Branch

Syntax:	GOTO k
Operands:	0 ≤ k ≤ 1048575
Operation:	k → PC<20:1>
Status Affected:	None

Encoding:

1st word (k<7:0>)	1110	1111	$k_7$kkk	kkkk$_0$
2nd word(k<19:8>)	1111	k_{19}kkk	kkkk	kkkk$_8$

Description: GOTO allows an unconditional branch anywhere within entire 2-Mbyte memory range. The 20-bit value 'k' is loaded into PC<20:1>. GOTO is always a two-cycle instruction.

Words: 2

Cycles: 2

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'<7:0>,	No operation	Read literal 'k'<19:8>, Write to PC
No operation	No operation	No operation	No operation

Example: `GOTO THERE`

After Instruction
PC = Address (THERE)

INCF — Increment f

Syntax:	INCF f {,d {,a}}
Operands:	0 ≤ f ≤ 255 d ∈ [0,1] a ∈ [0,1]
Operation:	(f) + 1 → dest
Status Affected:	C, DC, N, OV, Z

Encoding:

0010	10da	ffff	ffff

Description: The contents of register 'f' are incremented. If 'd' is '0', the result is placed in W. If 'd' is '1', the result is placed back in register 'f' (default).

If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the GPR bank (default).

If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh). See **Section 25.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example: `INCF CNT, 1, 0`

Before Instruction
CNT = FFh
Z = 0
C = ?
DC = ?

After Instruction
CNT = 00h
Z = 1
C = 1
DC = 1

© 2004 Microchip Technology Inc.

INCFSZ — Increment f, Skip if 0

Syntax:	INCFSZ f {,d {,a}}
Operands:	$0 \leq f \leq 255$ $d \in [0,1]$ $a \in [0,1]$
Operation:	(f) + 1 → dest, skip if result = 0
Status Affected:	None
Encoding:	0011 | 11da | ffff | ffff
Description:	The contents of register 'f' are incremented. If 'd' is '0', the result is placed in W. If 'd' is '1', the result is placed back in register 'f' (default). If the result is '0', the next instruction which is already fetched is discarded and a NOP is executed instead, making it a two-cycle instruction. If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the GPR bank (default). If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh). See **Section 25.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.
Words:	1
Cycles:	1(2) **Note:** 3 cycles if skip and followed by a 2-word instruction.

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

If skip:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation

If skip and followed by 2-word instruction:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation
No operation	No operation	No operation	No operation

Example:

```
HERE    INCFSZ   CNT, 1, 0
NZERO   :
ZERO    :
```

Before Instruction
PC = Address (HERE)
After Instruction
CNT = CNT + 1
If CNT = 0;
PC = Address (ZERO)
If CNT ≠ 0;
PC = Address (NZERO)

INFSNZ — Increment f, Skip if not 0

Syntax:	INFSNZ f {,d {,a}}
Operands:	$0 \leq f \leq 255$ $d \in [0,1]$ $a \in [0,1]$
Operation:	(f) + 1 → dest, skip if result ≠ 0
Status Affected:	None
Encoding:	0100 | 10da | ffff | ffff
Description:	The contents of register 'f' are incremented. If 'd' is '0', the result is placed in W. If 'd' is '1', the result is placed back in register 'f' (default). If the result is not '0', the next instruction which is already fetched is discarded and a NOP is executed instead, making it a two cycle instruction. If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the GPR bank (default). If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh). See **Section 25.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.
Words:	1
Cycles:	1(2) **Note:** 3 cycles if skip and followed by a 2-word instruction.

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

If skip:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation

If skip and followed by 2-word instruction:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation
No operation	No operation	No operation	No operation

Example:

```
HERE    INFSNZ   REG, 1, 0
ZERO
NZERO
```

Before Instruction
PC = Address (HERE)
After Instruction
REG = REG + 1
If REG ≠ 0;
PC = Address (NZERO)
If REG = 0;
PC = Address (ZERO)

IORLW — Inclusive OR Literal with W

Syntax:	IORLW k
Operands:	$0 \le k \le 255$
Operation:	(W) .OR. k → W
Status Affected:	N, Z
Encoding:	`0000` `1001` `kkkk` `kkkk`
Description:	The contents of W are ORed with the eight-bit literal 'k'. The result is placed in W.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'	Process Data	Write to W

Example: `IORLW 35h`

Before Instruction
W = 9Ah
After Instruction
W = BFh

IORWF — Inclusive OR W with f

Syntax:	IORWF f {,d {,a}}
Operands:	$0 \le f \le 255$ d ∈ [0,1] a ∈ [0,1]
Operation:	(W) .OR. (f) → dest
Status Affected:	N, Z
Encoding:	`0001` `00da` `ffff` `ffff`
Description:	Inclusive OR W with register 'f'. If 'd' is '0', the result is placed in W. If 'd' is '1', the result is placed back in register 'f' (default). If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the GPR bank (default). If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever $f \le 95$ (5Fh). See **Section 25.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example: `IORWF RESULT, 0, 1`

Before Instruction
RESULT = 13h
W = 91h
After Instruction
RESULT = 13h
W = 93h

© 2004 Microchip Technology Inc.

LFSR — Load FSR

Syntax: LFSR f, k

Operands: $0 \le f \le 2$
$0 \le k \le 4095$

Operation: k → FSRf

Status Affected: None

Encoding:

1110	1110	00ff	k_{11}kkk
1111	0000	$k_7$kkk	kkkk

Description: The 12-bit literal 'k' is loaded into the file select register pointed to by 'f'.

Words: 2

Cycles: 2

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k' MSB	Process Data	Write literal 'k' MSB to FSRfH
Decode	Read literal 'k' LSB	Process Data	Write literal 'k' to FSRfL

Example: LFSR 2, 3ABh

After Instruction
FSR2H = 03h
FSR2L = ABh

MOVF — Move f

Syntax: MOVF f {,d {,a}}

Operands: $0 \le f \le 255$
$d \in [0,1]$
$a \in [0,1]$

Operation: f → dest

Status Affected: N, Z

Encoding:

0101	00da	ffff	ffff

Description: The contents of register 'f' are moved to a destination dependent upon the status of 'd'. If 'd' is '0', the result is placed in W. If 'd' is '1', the result is placed back in register 'f' (default). Location 'f' can be anywhere in the 256-byte bank.

If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the GPR bank (default).

If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever $f \le 95$ (5Fh). See **Section 25.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write W

Example: MOVF REG, 0, 0

Before Instruction
REG = 22h
W = FFh

After Instruction
REG = 22h
W = 22h

© 2004 Microchip Technology Inc.

MOVFF	Move f to f
Syntax:	MOVFF f_s,f_d
Operands:	$0 \le f_s \le 4095$ $0 \le f_d \le 4095$
Operation:	$(f_s) \rightarrow f_d$
Status Affected:	None

Encoding:

1st word (source)	1100	ffff	ffff	$ffff_s$
2nd word (destin.)	1111	ffff	ffff	$ffff_d$

Description: The contents of source register 'f_s' are moved to destination register 'f_d'. Location of source 'f_s' can be anywhere in the 4096-byte data space (000h to FFFh) and location of destination 'f_d' can also be anywhere from 000h to FFFh.

Either source or destination can be W (a useful special situation).

MOVFF is particularly useful for transferring a data memory location to a peripheral register (such as the transmit buffer or an I/O port).

The MOVFF instruction cannot use the PCL, TOSU, TOSH or TOSL as the destination register.

Words: 2

Cycles: 2 (3)

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f' (src)	Process Data	No operation
Decode	No operation No dummy read	No operation	Write register 'f' (dest)

Example: MOVFF REG1, REG2

Before Instruction
REG1 = 33h
REG2 = 11h
After Instruction
REG1 = 33h
REG2 = 33h

MOVLB	Move Literal to Low Nibble in BSR
Syntax:	MOVLW k
Operands:	$0 \le k \le 255$
Operation:	$k \rightarrow BSR$
Status Affected:	None
Encoding:	0000 0001 kkkk kkkk
Description:	The eight-bit literal 'k' is loaded into the Bank Select Register (BSR). The value of BSR<7:4> always remains '0', regardless of the value of $k_7:k_4$.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'	Process Data	Write literal 'k' to BSR

Example: MOVLB 5

Before Instruction
BSR Register = 02h
After Instruction
BSR Register = 05h

 © 2004 Microchip Technology Inc.

MOVLW	Move Literal to W
Syntax:	MOVLW k
Operands:	0 ≤ k ≤ 255
Operation:	k → W
Status Affected:	None
Encoding:	0000 1110 kkkk kkkk
Description:	The eight-bit literal 'k' is loaded into W.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'	Process Data	Write to W

Example: MOVLW 5Ah

After Instruction
W = 5Ah

MOVWF	Move W to f
Syntax:	MOVWF f {,a}
Operands:	0 ≤ f ≤ 255 a ∈ [0,1]
Operation:	(W) → f
Status Affected:	None
Encoding:	0110 111a ffff ffff
Description:	Move data from W to register 'f'. Location 'f' can be anywhere in the 256-byte bank. If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the GPR bank (default). If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh). See **Section 25.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write register 'f'

Example: MOVWF REG, 0

Before Instruction
W = 4Fh
REG = FFh
After Instruction
W = 4Fh
REG = 4Fh

MULLW — Multiply Literal with W

Syntax:	MULLW k
Operands:	$0 \le k \le 255$
Operation:	(W) x k → PRODH:PRODL
Status Affected:	None
Encoding:	0000 1101 kkkk kkkk
Description:	An unsigned multiplication is carried out between the contents of W and the 8-bit literal 'k'. The 16-bit result is placed in the PRODH:PRODL register pair. PRODH contains the high byte. W is unchanged. None of the status flags are affected. Note that neither overflow nor carry is possible in this operation. A zero result is possible but not detected.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'	Process Data	Write registers PRODH: PRODL

Example: `MULLW 0C4h`

Before Instruction
- W = E2h
- PRODH = ?
- PRODL = ?

After Instruction
- W = E2h
- PRODH = ADh
- PRODL = 08h

MULWF — Multiply W with f

Syntax:	MULWF f {,a}
Operands:	$0 \le f \le 255$ $a \in [0,1]$
Operation:	(W) x (f) → PRODH:PRODL
Status Affected:	None
Encoding:	0000 001a ffff ffff
Description:	An unsigned multiplication is carried out between the contents of W and the register file location 'f'. The 16-bit result is stored in the PRODH:PRODL register pair. PRODH contains the high byte. Both W and 'f' are unchanged. None of the status flags are affected. Note that neither overflow nor carry is possible in this operation. A zero result is possible but not detected. If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the GPR bank (default). If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever $f \le 95$ (5Fh). See **Section 25.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write registers PRODH: PRODL

Example: `MULWF REG, 1`

Before Instruction
- W = C4h
- REG = B5h
- PRODH = ?
- PRODL = ?

After Instruction
- W = C4h
- REG = B5h
- PRODH = 8Ah
- PRODL = 94h

 Preliminary © 2004 Microchip Technology Inc.

NEGF	Negate f
Syntax:	NEGF f {,a}
Operands:	0 ≤ f ≤ 255 a ∈ [0,1]
Operation:	(f) + 1 → f
Status Affected:	N, OV, C, DC, Z
Encoding:	0110 | 110a | ffff | ffff
Description:	Location 'f' is negated using two's complement. The result is placed in the data memory location 'f'. If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the GPR bank (default). If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh). See **Section 25.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write register 'f'

Example: NEGF REG, 1

Before Instruction
REG = 0011 1010 [3Ah]
After Instruction
REG = 1100 0110 [C6h]

NOP	No Operation
Syntax:	NOP
Operands:	None
Operation:	No operation
Status Affected:	None
Encoding:	0000 | 0000 | 0000 | 0000 1111 | xxxx | xxxx | xxxx
Description:	No operation.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	No operation	No operation	No operation

Example:

None.

POP — Pop Top of Return Stack

Syntax: POP

Operands: None

Operation: (TOS) → bit bucket

Status Affected: None

Encoding:

0000	0000	0000	0110

Description: The TOS value is pulled off the return stack and is discarded. The TOS value then becomes the previous value that was pushed onto the return stack.

This instruction is provided to enable the user to properly manage the return stack to incorporate a software stack.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	No operation	POP TOS value	No operation

Example:

```
POP
GOTO    NEW
```

Before Instruction
- TOS = 0031A2h
- Stack (1 level down) = 014332h

After Instruction
- TOS = 014332h
- PC = NEW

PUSH — Push Top of Return Stack

Syntax: PUSH

Operands: None

Operation: (PC + 2) → TOS

Status Affected: None

Encoding:

0000	0000	0000	0101

Description: The PC + 2 is pushed onto the top of the return stack. The previous TOS value is pushed down on the stack.

This instruction allows implementing a software stack by modifying TOS and then pushing it onto the return stack.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	PUSH PC + 2 onto return stack	No operation	No operation

Example:

```
PUSH
```

Before Instruction
- TOS = 345Ah
- PC = 0124h

After Instruction
- PC = 0126h
- TOS = 0126h
- Stack (1 level down) = 345Ah

 © 2004 Microchip Technology Inc.

RCALL **Relative Call**

Syntax: RCALL n

Operands: $-1024 \le n \le 1023$

Operation: $(PC) + 2 \rightarrow TOS$,
$(PC) + 2 + 2n \rightarrow PC$

Status Affected: None

Encoding:

1101	1nnn	nnnn	nnnn

Description: Subroutine call with a jump up to 1K from the current location. First, return address (PC + 2) is pushed onto the stack. Then, add the 2's complement number '2n' to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be PC + 2 + 2n. This instruction is a two-cycle instruction.

Words: 1

Cycles: 2

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'n' PUSH PC to stack	Process Data	Write to PC
No operation	No operation	No operation	No operation

Example: HERE RCALL Jump

Before Instruction
PC = Address (HERE)
After Instruction
PC = Address (Jump)
TOS = Address (HERE + 2)

RESET **Reset**

Syntax: RESET

Operands: None

Operation: Reset all registers and flags that are affected by a MCLR Reset.

Status Affected: All

Encoding:

0000	0000	1111	1111

Description: This instruction provides a way to execute a MCLR Reset in software.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Start Reset	No operation	No operation

Example: RESET

After Instruction
Registers = Reset Value
Flags* = Reset Value

RETFIE — **Return from Interrupt**

Syntax: RETFIE {s}

Operands: s ∈ [0,1]

Operation: (TOS) → PC,
1 → GIE/GIEH or PEIE/GIEL,
if s = 1
(WS) → W,
(STATUSS) → Status,
(BSRS) → BSR,
PCLATU, PCLATH are unchanged.

Status Affected: GIE/GIEH, PEIE/GIEL

Encoding:

0000	0000	0001	000s

Description: Return from interrupt. Stack is popped and Top-of-Stack (TOS) is loaded into the PC. Interrupts are enabled by setting either the high or low priority global interrupt enable bit. If 's' = 1, the contents of the shadow registers, WS, STATUSS and BSRS, are loaded into their corresponding registers, W, Status and BSR. If 's' = 0, no update of these registers occurs (default).

Words: 1

Cycles: 2

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	No operation	No operation	POP PC from stack Set GIEH or GIEL
No operation	No operation	No operation	No operation

Example: RETFIE 1

After Interrupt
PC = TOS
W = WS
BSR = BSRS
Status = STATUSS
GIE/GIEH, PEIE/GIEL = 1

RETLW — **Return Literal to W**

Syntax: RETLW k

Operands: $0 \le k \le 255$

Operation: k → W,
(TOS) → PC,
PCLATU, PCLATH are unchanged

Status Affected: None

Encoding:

0000	1100	kkkk	kkkk

Description: W is loaded with the eight-bit literal 'k'. The program counter is loaded from the top of the stack (the return address). The high address latch (PCLATH) remains unchanged.

Words: 1

Cycles: 2

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'	Process Data	POP PC from stack, Write to W
No operation	No operation	No operation	No operation

Example:

```
    CALL  TABLE  ; W contains table
                 ; offset value
                 ; W now has
                 ; table value
    :
TABLE
    ADDWF PCL    ; W = offset
    RETLW k0     ; Begin table
    RETLW k1     ;
    :
    :
    RETLW kn     ; End of table
```

Before Instruction
W = 07h
After Instruction
W = value of kn

© 2004 Microchip Technology Inc.

RETURN	Return from Subroutine
Syntax:	RETURN {s}
Operands:	s ∈ [0,1]
Operation:	(TOS) → PC, if s = 1 (WS) → W, (STATUSS) → Status, (BSRS) → BSR, PCLATU, PCLATH are unchanged
Status Affected:	None
Encoding:	0000 0000 0001 001s
Description:	Return from subroutine. The stack is popped and the top of the stack (TOS) is loaded into the program counter. If 's'= 1, the contents of the shadow registers, WS, STATUSS and BSRS, are loaded into their corresponding registers, W, Status and BSR. If 's' = 0, no update of these registers occurs (default).
Words:	1
Cycles:	2

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	No operation	Process Data	POP PC from stack
No operation	No operation	No operation	No operation

Example: `RETURN`

After Interrupt
PC = TOS

RLCF	Rotate Left f through Carry
Syntax:	RLCF f {,d {,a}}
Operands:	$0 \le f \le 255$ d ∈ [0,1] a ∈ [0,1]
Operation:	(f<n>) → dest<n + 1>, (f<7>) → C, (C) → dest<0>
Status Affected:	C, N, Z
Encoding:	0011 01da ffff ffff
Description:	The contents of register 'f' are rotated one bit to the left through the Carry flag. If 'd' is '0', the result is placed in W. If 'd' is '1', the result is stored back in register 'f' (default). If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the GPR bank (default). If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh). See **Section 25.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.

Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example: `RLCF REG, 0, 0`

Before Instruction
REG = 1110 0110
C = 0
After Instruction
REG = 1110 0110
W = 1100 1100
C = 1

RLNCF — Rotate Left f (No Carry)

Syntax: RLNCF f {,d {,a}}

Operands: $0 \le f \le 255$
$d \in [0,1]$
$a \in [0,1]$

Operation: (f<n>) → dest<n + 1>,
(f<7>) → dest<0>

Status Affected: N, Z

Encoding:

0100	01da	ffff	ffff

Description: The contents of register 'f' are rotated one bit to the left. If 'd' is '0', the result is placed in W. If 'd' is '1', the result is stored back in register 'f' (default).

If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the GPR bank (default).

If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever $f \le 95$ (5Fh). See **Section 25.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example: RLNCF REG, 1, 0

Before Instruction
REG = 1010 1011
After Instruction
REG = 0101 0111

RRCF — Rotate Right f through Carry

Syntax: RRCF f {,d {,a}}

Operands: $0 \le f \le 255$
$d \in [0,1]$
$a \in [0,1]$

Operation: (f<n>) → dest<n – 1>,
(f<0>) → C,
(C) → dest<7>

Status Affected: C, N, Z

Encoding:

0011	00da	ffff	ffff

Description: The contents of register 'f' are rotated one bit to the right through the Carry flag. If 'd' is '0', the result is placed in W. If 'd' is '1', the result is placed back in register 'f' (default).

If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the GPR bank (default).

If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever $f \le 95$ (5Fh). See **Section 25.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example: RRCF REG, 0, 0

Before Instruction
REG = 1110 0110
C = 0
After Instruction
REG = 1110 0110
W = 0111 0011
C = 0

© 2004 Microchip Technology Inc.

RRNCF — Rotate Right f (No Carry)

Syntax: RRNCF f {,d {,a}}

Operands: $0 \le f \le 255$
$d \in [0,1]$
$a \in [0,1]$

Operation: (f<n>) → dest<n – 1>,
(f<0>) → dest<7>

Status Affected: N, Z

Encoding:

0100	00da	ffff	ffff

Description: The contents of register 'f' are rotated one bit to the right. If 'd' is '0', the result is placed in W. If 'd' is '1', the result is placed back in register 'f' (default).

If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' is '1', then the bank will be selected as per the BSR value (default).

If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever $f \le 95$ (5Fh). See **Section 25.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example 1: RRNCF REG, 1, 0

Before Instruction
REG = 1101 0111
After Instruction
REG = 1110 1011

Example 2: RRNCF REG, 0, 0

Before Instruction
W = ?
REG = 1101 0111
After Instruction
W = 1110 1011
REG = 1101 0111

SETF — Set f

Syntax: SETF f {,a}

Operands: $0 \le f \le 255$
$a \in [0,1]$

Operation: FFh → f

Status Affected: None

Encoding:

0110	100a	ffff	ffff

Description: The contents of the specified register are set to FFh.

If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the GPR bank (default).

If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever $f \le 95$ (5Fh). See **Section 25.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write register 'f'

Example: SETF REG, 1

Before Instruction
REG = 5Ah
After Instruction
REG = FFh

SLEEP — Enter Sleep mode

Syntax:	SLEEP
Operands:	None
Operation:	00h → WDT, 0 → WDT postscaler, 1 → $\overline{TO}$, 0 → $\overline{PD}$
Status Affected:	$\overline{TO}$, $\overline{PD}$
Encoding:	0000 | 0000 | 0000 | 0011
Description:	The Power-Down status bit ($\overline{PD}$) is cleared. The Time-out status bit ($\overline{TO}$) is set. Watchdog Timer and its postscaler are cleared. The processor is put into Sleep mode with the oscillator stopped.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	No operation	Process Data	Go to Sleep

Example: `SLEEP`

Before Instruction
- $\overline{TO}$ = ?
- $\overline{PD}$ = ?

After Instruction
- $\overline{TO}$ = 1 †
- $\overline{PD}$ = 0

† If WDT causes wake-up, this bit is cleared.

SUBFWB — Subtract f from W with Borrow

Syntax:	SUBFWB f {,d {,a}}
Operands:	$0 \le f \le 255$ $d \in [0,1]$ $a \in [0,1]$
Operation:	(W) – (f) – ($\overline{C}$) → dest
Status Affected:	N, OV, C, DC, Z
Encoding:	0101 | 01da | ffff | ffff
Description:	Subtract register 'f' and Carry flag (borrow) from W (2's complement method). If 'd' is '0', the result is stored in W. If 'd' is '1', the result is stored in register 'f' (default). If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the GPR bank (default). If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever $f \le 95$ (5Fh). See **Section 25.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example 1: `SUBFWB REG, 1, 0`

Before Instruction
- REG = 3
- W = 2
- C = 1

After Instruction
- REG = FF
- W = 2
- C = 0
- Z = 0
- N = 1 ; result is negative

Example 2: `SUBFWB REG, 0, 0`

Before Instruction
- REG = 2
- W = 5
- C = 1

After Instruction
- REG = 2
- W = 3
- C = 1
- Z = 0
- N = 0 ; result is positive

Example 3: `SUBFWB REG, 1, 0`

Before Instruction
- REG = 1
- W = 2
- C = 0

After Instruction
- REG = 0
- W = 2
- C = 1
- Z = 1 ; result is zero
- N = 0

 © 2004 Microchip Technology Inc.

SUBLW **Subtract W from Literal**

Syntax:	SUBLW k
Operands:	$0 \le k \le 255$
Operation:	k – (W) → W
Status Affected:	N, OV, C, DC, Z
Encoding:	`0000` `1000` `kkkk` `kkkk`
Description:	W is subtracted from the eight-bit literal 'k'. The result is placed in W.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'	Process Data	Write to W

Example 1: `SUBLW 02h`

```
Before Instruction
    W   =   01h
    C   =   ?
After Instruction
    W   =   01h
    C   =   1      ; result is positive
    Z   =   0
    N   =   0
```

Example 2: `SUBLW 02h`

```
Before Instruction
    W   =   02h
    C   =   ?
After Instruction
    W   =   00h
    C   =   1      ; result is zero
    Z   =   1
    N   =   0
```

Example 3: `SUBLW 02h`

```
Before Instruction
    W   =   03h
    C   =   ?
After Instruction
    W   =   FFh, (2's complement)
    C   =   0    ; result is negative
    Z   =   0
    N   =   1
```

SUBWF **Subtract W from f**

Syntax:	SUBWF f {,d {,a}}
Operands:	$0 \le f \le 255$ d ∈ [0,1] a ∈ [0,1]
Operation:	(f) – (W) → dest
Status Affected:	N, OV, C, DC, Z
Encoding:	`0101` `11da` `ffff` `ffff`
Description:	Subtract W from register 'f' (2's complement method). If 'd' is '0', the result is stored in W. If 'd' is '1', the result is stored back in register 'f' (default). If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the GPR bank (default). If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever $f \le 95$ (5Fh). See **Section 25.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example 1: `SUBWF REG, 1, 0`

```
Before Instruction
    REG  =   3
    W    =   2
    C    =   ?
After Instruction
    REG  =   1
    W    =   2
    C    =   1    ; result is positive
    Z    =   0
    N    =   0
```

Example 2: `SUBWF REG, 0, 0`

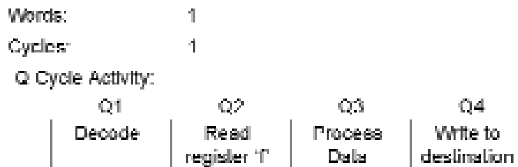
```
Before Instruction
    REG  =   2
    W    =   2
    C    =   ?
After Instruction
    REG  =   2
    W    =   0
    C    =   1    ; result is zero
    Z    =   1
    N    =   0
```

Example 3: `SUBWF REG, 1, 0`

```
Before Instruction
    REG  =   1
    W    =   2
    C    =   ?
After Instruction
    REG  =   FFh   ;(2's complement)
    W    =   2
    C    =   0     ; result is negative
    Z    =   0
    N    =   1
```

SUBWFB	Subtract W from f with Borrow
Syntax:	SUBWFB f {,d {,a}}
Operands:	0 ≤ f ≤ 255 d ∈ [0,1] a ∈ [0,1]
Operation:	(f) – (W) – ($\overline{C}$) → dest
Status Affected:	N, OV, C, DC, Z
Encoding:	0101 | 10da | ffff | ffff
Description:	Subtract W and the Carry flag (borrow) from register 'f' (2's complement method). If 'd' is '0', the result is stored in W. If 'd' is '1', the result is stored back in register 'f' (default). If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the GPR bank (default). If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh). See **Section 25.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.
Words:	1
Cycles:	1

Q Cycle Activity:



Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example 1: `SUBWFB REG, 1, 0`

Before Instruction
- REG = 19h (0001 1001)
- W = 0Dh (0000 1101)
- C = 1

After Instruction
- REG = 0Ch (0000 1011)
- W = 0Dh (0000 1101)
- C = 1
- Z = 0
- N = 0 ; result is positive

Example 2: `SUBWFB REG, 0, 0`

Before Instruction
- REG = 1Bh (0001 1011)
- W = 1Ah (0001 1010)
- C = 0

After Instruction
- REG = 1Bh (0001 1011)
- W = 00h
- C = 1
- Z = 1 ; result is zero
- N = 0

Example 3: `SUBWFB REG, 1, 0`

Before Instruction
- REG = 03h (0000 0011)
- W = 0Eh (0000 1101)
- C = 1

After Instruction
- REG = F5h (1111 0100) ; [2's comp]
- W = 0Eh (0000 1101)
- C = 0
- Z = 0
- N = 1 ; result is negative

SWAPF	Swap f
Syntax:	SWAPF f {,d {,a}}
Operands:	0 ≤ f ≤ 255 d ∈ [0,1] a ∈ [0,1]
Operation:	(f<3:0>) → dest<7:4>, (f<7:4>) → dest<3:0>
Status Affected:	None
Encoding:	0011 | 10da | ffff | ffff
Description:	The upper and lower nibbles of register 'f' are exchanged. If 'd' is '0', the result is placed in W. If 'd' is '1', the result is placed in register 'f' (default). If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the GPR bank (default). If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh). See **Section 25.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example: `SWAPF REG, 1, 0`

Before Instruction
- REG = 53h

After Instruction
- REG = 35h

 Preliminary © 2004 Microchip Technology Inc.

TBLRD Table Read

Syntax: TBLRD (*; *+; *-; +*)

Operands: None

Operation:
if TBLRD *,
(Prog Mem (TBLPTR)) → TABLAT;
TBLPTR – No Change;
if TBLRD *+,
(Prog Mem (TBLPTR)) → TABLAT;
(TBLPTR) + 1 → TBLPTR;
if TBLRD *-,
(Prog Mem (TBLPTR)) → TABLAT;
(TBLPTR) – 1 → TBLPTR;
if TBLRD +*,
(TBLPTR) + 1 → TBLPTR;
(Prog Mem (TBLPTR)) → TABLAT;

Status Affected: None

Encoding:

0000	0000	0000	10nn nn=0 * =1 *+ =2 *- =3 +*

Description: This instruction is used to read the contents of Program Memory (P.M.). To address the program memory, a pointer, called Table Pointer (TBLPTR), is used.

The TBLPTR (a 21-bit pointer) points to each byte in the program memory. TBLPTR has a 2 Mbyte address range.

TBLPTR[0] = 0: Least Significant Byte of Program Memory Word
TBLPTR[0] = 1: Most Significant Byte of Program Memory Word

The `TBLRD` instruction can modify the value of TBLPTR as follows:

- no change
- post-increment
- post-decrement
- pre-increment

Words: 1

Cycles: 2

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	No operation	No operation	No operation
No operation	No operation (Read Program Memory)	No operation	No operation (Write TABLAT)

TBLRD Table Read (Continued)

Example 1: `TBLRD *+ ;`

Before Instruction

TABLAT	=	55h
TBLPTR	=	00A356h
MEMORY(00A356h)	=	34h

After Instruction

TABLAT	=	34h
TBLPTR	=	00A357h

Example 2: `TBLRD +* ;`

Before Instruction

TABLAT	=	0AAh
TBLPTR	=	01A357h
MEMORY(01A357h)	=	12h
MEMORY(01A358h)	=	34h

After Instruction

TABLAT	=	34h
TBLPTR	=	01A358h

TBLWT — Table Write

Syntax:	TBLWT (*; *+; *-; +*)
Operands:	None
Operation:	if TBLWT*, (TABLAT) → Holding Register; TBLPTR – No Change; if TBLWT*+, (TABLAT) → Holding Register; (TBLPTR) + 1 → TBLPTR; if TBLWT*-, (TABLAT) → Holding Register; (TBLPTR) – 1 → TBLPTR; if TBLWT+*, (TBLPTR) + 1 → TBLPTR; (TABLAT) → Holding Register;
Status Affected:	None

Encoding:

0000	0000	0000	11nn nn=0 * =1 *+ =2 *- =3 +*

Description: This instruction uses the 3 LSBs of the TBLPTR to determine which of the 8 holding registers the TABLAT is written to. The holding registers are used to program the contents of Program Memory (P.M.). (Refer to **Section 6.0 "Flash Program Memory"** for additional details on programming Flash memory.)

The TBLPTR (a 21-bit pointer) points to each byte in the program memory. TBLPTR has a 2-MByte address range. The LSb of the TBLPTR selects which byte of the program memory location to access.

TBLPTR[0] = 0: Least Significant Byte of Program Memory Word

TBLPTR[0] = 1: Most Significant Byte of Program Memory Word

The `TBLWT` instruction can modify the value of TBLPTR as follows:

- no change
- post-increment
- post-decrement
- pre-increment

Words: 1

Cycles: 2

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	No operation	No operation	No operation
No operation	No operation (Read TABLAT)	No operation	No operation (Write to Holding Register)

TBLWT — Table Write (Continued)

Example 1: `TBLWT *+;`

Before Instruction

TABLAT	=	55h
TBLPTR	=	00A356h
HOLDING REGISTER (00A356h)	=	FFh

After Instructions (table write completion)

TABLAT	=	55h
TBLPTR	=	00A357h
HOLDING REGISTER (00A356h)	=	55h

Example 2: `TBLWT +*;`

Before Instruction

TABLAT	=	34h
TBLPTR	=	01389Ah
HOLDING REGISTER (01389Ah)	=	FFh
HOLDING REGISTER (01389Bh)	=	FFh

After Instruction (table write completion)

TABLAT	=	34h
TBLPTR	=	01389Bh
HOLDING REGISTER (01389Ah)	=	FFh
HOLDING REGISTER (01389Bh)	=	34h

© 2004 Microchip Technology Inc.

TSTFSZ — Test f, Skip if 0

Syntax:	TSTFSZ f {,a}
Operands:	$0 \leq f \leq 255$ $a \in [0,1]$
Operation:	skip if f = 0
Status Affected:	None
Encoding:	0110 | 011a | ffff | ffff
Description:	If 'f' = 0, the next instruction fetched during the current instruction execution is discarded and a NOP is executed, making this a two-cycle instruction. If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the GPR bank (default). If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever $f \leq 95$ (5Fh). See **Section 25.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.
Words:	1
Cycles:	1(2) **Note:** 3 cycles if skip and followed by a 2-word instruction.

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	No operation

If skip:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation

If skip and followed by 2-word instruction:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation
No operation	No operation	No operation	No operation

Example:

```
HERE    TSTFSZ  CNT, 1
NZERO   :
ZERO    :
```

Before Instruction
PC = Address (HERE)
After Instruction
If CNT = 00h,
PC = Address (ZERO)
If CNT ≠ 00h,
PC = Address (NZERO)

XORLW — Exclusive OR Literal with W

Syntax:	XORLW k
Operands:	$0 \leq k \leq 255$
Operation:	(W) .XOR. k → W
Status Affected:	N, Z
Encoding:	0000 | 1010 | kkkk | kkkk
Description:	The contents of W are XORed with the 8-bit literal 'k'. The result is placed in W.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'	Process Data	Write to W

Example:

```
XORLW   0AFh
```

Before Instruction
W = B5h
After Instruction
W = 1Ah

© 2004 Microchip Technology Inc.

XORWF	Exclusive OR W with f
Syntax:	XORWF f {,d {,a}}
Operands:	$0 \le f \le 255$ $d \in [0,1]$ $a \in [0,1]$
Operation:	(W) .XOR. (f) → dest
Status Affected:	N, Z
Encoding:	0001 | 10da | ffff | ffff
Description:	Exclusive OR the contents of W with register 'f'. If 'd' is '0', the result is stored in W. If 'd' is '1', the result is stored back in the register 'f' (default). If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the GPR bank (default). If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever $f \le 95$ (5Fh). See **Section 25.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example: `XORWF REG, 1, 0`

Before Instruction
- REG = AFh
- W = B5h

After Instruction
- REG = 1Ah
- W = B5h

 © 2004 Microchip Technology Inc.

25.2 Extended Instruction Set

In addition to the standard 75 instructions of the PIC18 instruction set, PIC18F2480/2580/4480/4580 devices also provide an optional extension to the core CPU functionality. The added features include eight additional instructions that augment indirect and indexed addressing operations and the implementation of Indexed Literal Offset Addressing mode for many of the standard PIC18 instructions.

The additional features are disabled by default. To enable them, users must set the XINST configuration bit.

The instructions in the extended set can all be classified as literal operations, which either manipulate the File Select Registers or use them for indexed addressing. Two of the instructions, ADDFSR and SUBFSR, each have an additional special instantiation for using FSR2. These versions (ADDULNK and SUBULNK) allow for automatic return after execution.

The extended instructions are specifically implemented to optimize re-entrant program code (that is, code that is recursive or that uses a software stack) written in high-level languages, particularly C. Among other things, they allow users working in high-level languages to perform certain operations on data structures more efficiently. These include:

- dynamic allocation and de-allocation of software stack space when entering and leaving subroutines
- function pointer invocation
- software Stack Pointer manipulation
- manipulation of variables located in a software stack

A summary of the instructions in the extended instruction set is provided in Table 25-3. Detailed descriptions are provided in **Section 25.2.2 "Extended Instruction Set"**. The opcode field descriptions in Table 25-1 apply to both the standard and extended PIC18 instruction sets.

> **Note:** The instruction set extension and the Indexed Literal Offset Addressing mode were designed for optimizing applications written in C; the user may likely never use these instructions directly in assembler. The syntax for these commands is provided as a reference for users who may be reviewing code that has been generated by a compiler.

25.2.1 EXTENDED INSTRUCTION SYNTAX

Most of the extended instructions use indexed arguments, using one of the File Select Registers and some offset to specify a source or destination register. When an argument for an instruction serves as part of indexed addressing, it is enclosed in square brackets ("[ ]"). This is done to indicate that the argument is used as an index or offset. MPASM™ Assembler will flag an error if it determines that an index or offset value is not bracketed.

When the extended instruction set is enabled, brackets are also used to indicate index arguments in byte-oriented and bit-oriented instructions. This is in addition to other changes in their syntax. For more details, see **Section 25.2.3.1 "Extended Instruction Syntax with Standard PIC18 Commands"**.

> **Note:** In the past, square brackets have been used to denote optional arguments in the PIC18 and earlier instruction sets. In this text and going forward, optional arguments are denoted by braces ("{ }").

TABLE 25-3: EXTENSIONS TO THE PIC18 INSTRUCTION SET

Mnemonic, Operands		Description	Cycles	16-Bit Instruction Word (MSb … LSb)	Status Affected
ADDFSR	f, k	Add literal to FSR	1	1110 1000 ffkk kkkk	None
ADDULNK	k	Add literal to FSR2 and return	2	1110 1000 11kk kkkk	None
CALLW		Call subroutine using WREG	2	0000 0000 0001 0100	None
MOVSF	z_s, f_d	Move z_s (source) to 1st word	2	1110 1011 0zzz zzzz	None
		f_d (destination) 2nd word		1111 ffff ffff ffff	
MOVSS	z_s, z_d	Move z_s (source) to 1st word	2	1110 1011 1zzz zzzz	None
		z_d (destination) 2nd word		1111 xxxx xzzz zzzz	
PUSHL	k	Store literal at FSR2, decrement FSR2	1	1110 1010 kkkk kkkk	None
SUBFSR	f, k	Subtract literal from FSR	1	1110 1001 ffkk kkkk	None
SUBULNK	k	Subtract literal from FSR2 and return	2	1110 1001 11kk kkkk	None

25.2.2 EXTENDED INSTRUCTION SET

ADDFSR — Add Literal to FSR

Syntax:	ADDFSR f, k
Operands:	0 ≤ k ≤ 63 f ∈ [0, 1, 2]
Operation:	FSR(f) + k → FSR(f)
Status Affected:	None
Encoding:	1110 1000 ffkk kkkk
Description:	The 6-bit literal 'k' is added to the contents of the FSR specified by 'f'.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'	Process Data	Write to FSR

Example: `ADDFSR 2, 23h`

Before Instruction
FSR2 = 03FFh
After Instruction
FSR2 = 0422h

ADDULNK — Add Literal to FSR2 and Return

Syntax:	ADDULNK k
Operands:	0 ≤ k ≤ 63
Operation:	FSR2 + k → FSR2, PC = (TOS)
Status Affected:	None
Encoding:	1110 1000 11kk kkkk
Description:	The 6-bit literal 'k' is added to the contents of FSR2. A RETURN is then executed by loading the PC with the TOS. The instruction takes two cycles to execute; a NOP is performed during the second cycle. This may be thought of as a special case of the ADDFSR instruction, where f = 3 (binary '11'); it operates only on FSR2.
Words:	1
Cycles:	2

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'	Process Data	Write to FSR
No Operation	No Operation	No Operation	No Operation

Example: `ADDULNK 23h`

Before Instruction
FSR2 = 03FFh
PC = 0100h
TOS = 02AFh
After Instruction
FSR2 = 0422h
PC = 02AFh
TOS = TOS – 1

Note: All PIC18 instructions may take an optional label argument preceding the instruction mnemonic for use in symbolic addressing. If a label is used, the instruction syntax then becomes: {label} instruction argument(s).

 © 2004 Microchip Technology Inc.

CALLW — **Subroutine Call Using WREG**

Syntax: CALLW

Operands: None

Operation: (PC + 2) → TOS,
(W) → PCL,
(PCLATH) → PCH,
(PCLATU) → PCU

Status Affected: None

Encoding:

0000	0000	0001	0100

Description: First, the return address (PC + 2) is pushed onto the return stack. Next, the contents of W are written to PCL; the existing value is discarded. Then, the contents of PCLATH and PCLATU are latched into PCH and PCU, respectively. The second cycle is executed as a NOP instruction while the new next instruction is fetched.

Unlike CALL, there is no option to update W, Status or BSR.

Words: 1

Cycles: 2

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read WREG	Push PC to stack	No operation
No operation	No operation	No operation	No operation

Example: HERE CALLW

Before Instruction
- PC = address (HERE)
- PCLATH = 10h
- PCLATU = 00h
- W = 06h

After Instruction
- PC = 001006h
- TOS = address (HERE + 2)
- PCLATH = 10h
- PCLATU = 00h
- W = 06h

MOVSF — **Move Indexed to f**

Syntax: MOVSF [z_s], f_d

Operands: $0 \le z_s \le 127$
$0 \le f_d \le 4095$

Operation: $((FSR2) + z_s) \rightarrow f_d$

Status Affected: None

Encoding:

1st word (source)	1110	1011	0zzz	$zzzz_s$
2nd word (destin.)	1111	ffff	ffff	$ffff_d$

Description: The contents of the source register are moved to destination register 'f_d'. The actual address of the source register is determined by adding the 7-bit literal offset 'z_s' in the first word to the value of FSR2. The address of the destination register is specified by the 12-bit literal 'f_d' in the second word. Both addresses can be anywhere in the 4096-byte data space (000h to FFFh).

The MOVSF instruction cannot use the PCL, TOSU, TOSH or TOSL as the destination register.

If the resultant source address points to an indirect addressing register, the value returned will be 00h.

Words: 2

Cycles: 2

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Determine source addr	Determine source addr	Read source reg
Decode	No operation No dummy read	No operation	Write register 'f' (dest)

Example: MOVSF [05h], REG2

Before Instruction
- FSR2 = 80h
- Contents of 85h = 33h
- REG2 = 11h

After Instruction
- FSR2 = 80h
- Contents of 85h = 33h
- REG2 = 33h

MOVSS — Move Indexed to Indexed

Syntax:	MOVSS [z_s], [z_d]
Operands:	$0 \le z_s \le 127$ $0 \le z_d \le 127$
Operation:	$((FSR2) + z_s) \rightarrow ((FSR2) + z_d)$
Status Affected:	None

Encoding:

1st word (source)	1110	1011	1zzz	$zzzz_s$
2nd word (dest.)	1111	xxxx	xzzz	$zzzz_d$

Description: The contents of the source register are moved to the destination register. The addresses of the source and destination registers are determined by adding the 7-bit literal offsets 'z_s' or 'z_d', respectively, to the value of FSR2. Both registers can be located anywhere in the 4096 byte data memory space (000h to FFFh).

The MOVSS instruction cannot use the PCL, TOSU, TOSH or TOSL as the destination register.

If the resultant source address points to an indirect addressing register, the value returned will be 00h. If the resultant destination address points to an indirect addressing register, the instruction will execute as a NOP.

Words: 2

Cycles: 2

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Determine source addr	Determine source addr	Read source reg
Decode	Determine dest addr	Determine dest addr	Write to dest reg

Example: `MOVSS [05h], [06h]`

Before Instruction

FSR2	=	80h
Contents of 85h	=	33h
Contents of 86h	=	11h

After Instruction

FSR2	=	80h
Contents of 85h	=	33h
Contents of 86h	=	33h

PUSHL — Store Literal at FSR2, Decrement FSR2

Syntax:	PUSHL k
Operands:	$0 \le k \le 255$
Operation:	$k \rightarrow (FSR2)$, $FSR2 - 1 \rightarrow FSR2$
Status Affected:	None

Encoding:

1111	1010	kkkk	kkkk

Description: The 8-bit literal 'k' is written to the data memory address specified by FSR2. FSR2 is decremented by 1 after the operation.

This instruction allows users to push values onto a software stack.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read 'k'	Process data	Write to destination

Example: `PUSHL 08h`

Before Instruction

FSR2H:FSR2L	=	01ECh
Memory (01ECh)	=	00h

After Instruction

FSR2H:FSR2L	=	01EBh
Memory (01ECh)	=	08h

 © 2004 Microchip Technology Inc.

SUBFSR **Subtract Literal from FSR**

Syntax:	SUBFSR f, k
Operands:	0 ≤ k ≤ 63 f ∈ [0, 1, 2]
Operation:	FSRf – k → FSRf
Status Affected:	None
Encoding:	1110 | 1001 | ffkk | kkkk
Description:	The 6-bit literal 'k' is subtracted from the contents of the FSR specified by 'f'.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example: `SUBFSR 2, 23h`

Before Instruction
FSR2 = 03FFh
After Instruction
FSR2 = 03DCh

SUBULNK **Subtract Literal from FSR2 and Return**

Syntax:	SUBULNK k
Operands:	0 ≤ k ≤ 63
Operation:	FSR2 – k → FSR2 (TOS) → PC
Status Affected:	None
Encoding:	1110 | 1001 | 11kk | kkkk
Description:	The 6-bit literal 'k' is subtracted from the contents of the FSR2. A RETURN is then executed by loading the PC with the TOS. The instruction takes two cycles to execute; a NOP is performed during the second cycle. This may be thought of as a special case of the SUBFSR instruction, where f = 3 (binary '11'); it operates only on FSR2.
Words:	1
Cycles:	2

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination
No Operation	No Operation	No Operation	No Operation

Example: `SUBULNK 23h`

Before Instruction
FSR2 = 03FFh
PC = 0100h
After Instruction
FSR2 = 03DCh
PC = (TOS)

25.2.3 BYTE-ORIENTED AND BIT-ORIENTED INSTRUCTIONS IN INDEXED LITERAL OFFSET MODE

Note:	Enabling the PIC18 instruction set extension may cause legacy applications to behave erratically or fail entirely.

In addition to eight new commands in the extended set, enabling the extended instruction set also enables Indexed Literal Offset Addressing mode (**Section 5.6.1 "Indexed Addressing with Literal Offset"**). This has a significant impact on the way that many commands of the standard PIC18 instruction set are interpreted.

When the extended set is disabled, addresses embedded in opcodes are treated as literal memory locations: either as a location in the Access Bank (a = 0), or in a GPR bank designated by the BSR (a = 1). When the extended instruction set is enabled and a = 0, however, a file register argument of 5Fh or less is interpreted as an offset from the pointer value in FSR2 and not as a literal address. For practical purposes, this means that all instructions that use the Access RAM bit as an argument – that is, all byte-oriented and bit-oriented instructions, or almost half of the core PIC18 instructions – may behave differently when the extended instruction set is enabled.

When the content of FSR2 is 00h, the boundaries of the Access RAM are essentially remapped to their original values. This may be useful in creating backward compatible code. If this technique is used, it may be necessary to save the value of FSR2 and restore it when moving back and forth between 'C' and assembly routines in order to preserve the Stack Pointer. Users must also keep in mind the syntax requirements of the extended instruction set (see **Section 25.2.3.1 "Extended Instruction Syntax with Standard PIC18 Commands"**).

Although the Indexed Literal Offset Addressing mode can be very useful for dynamic stack and pointer manipulation, it can also be very annoying if a simple arithmetic operation is carried out on the wrong register. Users who are accustomed to the PIC18 programming must keep in mind that, when the extended instruction set is enabled, register addresses of 5Fh or less are used for Indexed Literal Offset Addressing.

Representative examples of typical byte-oriented and bit-oriented instructions in the Indexed Literal Offset Addressing mode are provided on the following page to show how execution is affected. The operand conditions shown in the examples are applicable to all instructions of these types.

25.2.3.1 Extended Instruction Syntax with Standard PIC18 Commands

When the extended instruction set is enabled, the file register argument, 'f', in the standard byte-oriented and bit-oriented commands is replaced with the literal offset value, 'k'. As already noted, this occurs only when 'f' is less than or equal to 5Fh. When an offset value is used, it must be indicated by square brackets ("[ ]"). As with the extended instructions, the use of brackets indicates to the compiler that the value is to be interpreted as an index or an offset. Omitting the brackets, or using a value greater than 5Fh within brackets, will generate an error in the MPASM™ Assembler.

If the index argument is properly bracketed for Indexed Literal Offset Addressing, the Access RAM argument is never specified; it will automatically be assumed to be '0'. This is in contrast to standard operation (extended instruction set disabled) when 'a' is set on the basis of the target address. Declaring the Access RAM bit in this mode will also generate an error in the MPASM Assembler.

The destination argument, 'd', functions as before.

In the latest versions of the MPASM assembler, language support for the extended instruction set must be explicitly invoked. This is done with either the command line option, /y, or the PE directive in the source listing.

25.2.4 CONSIDERATIONS WHEN ENABLING THE EXTENDED INSTRUCTION SET

It is important to note that the extensions to the instruction set may not be beneficial to all users. In particular, users who are not writing code that uses a software stack may not benefit from using the extensions to the instruction set.

Additionally, the Indexed Literal Offset Addressing mode may create issues with legacy applications written to the PIC18 assembler. This is because instructions in the legacy code may attempt to address registers in the Access Bank below 5Fh. Since these addresses are interpreted as literal offsets to FSR2 when the instruction set extension is enabled, the application may read or write to the wrong data addresses.

When porting an application to the PIC18F2480/2580/4480/4580, it is very important to consider the type of code. A large, re-entrant application that is written in 'C' and would benefit from efficient compilation will do well when using the instruction set extensions. Legacy applications that heavily use the Access Bank will most likely not benefit from using the extended instruction set.

 © 2004 Microchip Technology Inc.

ADDWF — ADD W to Indexed (Indexed Literal Offset mode)

Syntax: ADDWF [k] {,d}

Operands: $0 \le k \le 95$
$d \in [0,1]$
$a = 0$

Operation: (W) + ((FSR2) + k) → dest

Status Affected: N, OV, C, DC, Z

Encoding:

0010	01d0	kkkk	kkkk

Description: The contents of W are added to the contents of the register indicated by FSR2, offset by the value 'k'.
If 'd' is '0', the result is stored in W. If 'd' is '1', the result is stored back in register 'f' (default).

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read 'k'	Process Data	Write to destination

Example: ADDWF [OFST] ,0

Before Instruction
W = 17h
OFST = 2Ch
FSR2 = 0A00h
Contents of 0A2Ch = 20h
After Instruction
W = 37h
Contents of 0A2Ch = 20h

BSF — Bit Set Indexed (Indexed Literal Offset mode)

Syntax: BSF [k], b

Operands: $0 \le f \le 95$
$0 \le b \le 7$
$a = 0$

Operation: 1 → ((FSR2 + k))<b>

Status Affected: None

Encoding:

1000	bbb0	kkkk	kkkk

Description: Bit 'b' of the register indicated by FSR2, offset by the value 'k', is set.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example: BSF [FLAG_OFST], 7

Before Instruction
FLAG_OFST = 0Ah
FSR2 = 0A00h
Contents of 0A0Ah = 55h
After Instruction
Contents of 0A0Ah = D5h

SETF — Set Indexed (Indexed Literal Offset mode)

Syntax: SETF [k]

Operands: $0 \le k \le 95$

Operation: FFh → ((FSR2) + k)

Status Affected: None

Encoding:

0110	1000	kkkk	kkkk

Description: The contents of the register indicated by FSR2, offset by 'k', are set to FFh.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read 'k'	Process Data	Write register

Example: SETF [OFST]

Before Instruction
OFST = 2Ch
FSR2 = 0A00h
Contents of 0A2Ch = 00h
After Instruction
Contents of 0A2Ch = FFh

25.2.5 SPECIAL CONSIDERATIONS WITH MICROCHIP MPLAB® IDE TOOLS

The latest versions of Microchip's software tools have been designed to fully support the extended instruction set of the PIC18F2480/2580/4480/4580 family of devices. This includes the MPLAB C18 C compiler, MPASM assembly language and MPLAB Integrated Development Environment (IDE).

When selecting a target device for software development, MPLAB IDE will automatically set default configuration bits for that device. The default setting for the XINST configuration bit is '0', disabling the extended instruction set and Indexed Literal Offset Addressing mode. For proper execution of applications developed to take advantage of the extended instruction set, XINST must be set during programming.

To develop software for the extended instruction set, the user must enable support for the instructions and the Indexed Addressing mode in their language tool(s). Depending on the environment being used, this may be done in several ways.

- A menu option, or dialog box within the environment, that allows the user to configure the language tool and its settings for the project
- A command line option
- A directive in the source code

These options vary between different compilers, assemblers and development environments. Users are encouraged to review the documentation accompanying their development systems for the appropriate information.