PIC Instruction Set

Instruction Set:

The instruction set for PIC16Cxx consists of only 35 instructions.
Some of these instructions are **byte oriented** instructions and some are **bit oriented** instructions.

The **byte oriented instructions** that require two parameters .
For example, movf f, F(W) expect the f to be replaced by the name of a special purpose register (e.g., PORTA) or the name of a RAM variable (e.g., NUM1), which serves as the source of the operand. 'f' stands for file register.
The F(W) parameter is the destination of the result of the operation. It should be replaced by:

F, if the destination is to be the source register.
W, if the destination is to be the working register (i.e., Accumulator or W register).

The **bit oriented instructions** also expect parameters.
(bfs f, b). Here 'f' is to be replaced by the name of a special purpose register or the name of a RAM variable. The 'b' parameter is to be replaced by a bit number ranging from 0 to 7.

For example:
Z equ 2
bfs STATUS, Z

Z has been equated to 2. Here, the instruction will set the Z bit of the STATUS register.

The **literal instructions** require an operand having a known value (e.g., 0AH) or a label that represents a known value.

For example:

NUM equu 0AH ; Assigns 0AH to the label NUM (a constant)

movlw NUM ; will move 0AH to the W register.

Instruction Set:

1. Single-bit manipulation:

bcf f, b	Clear bit b of register f
bsf f, b	Set bit b of register f

Example:

```
bcf    PORTB,0      ;Clear bit 0 of PORTB
bsf    STATUS,C     ;Set the carry bit
```

2. Clear/Move

clrw	Clear working register W
clrf f	Clear f
movlw k	Move literal 'k' to W
movwf f	Move W to f
movf f, F(W)	Move f to F or W
swapf f, F(W)	Swap nibbles of f, putting result in F or W

Example:

```
clrw                     ;Clear the working register, W
clrf    TEMP1            ;Clear temporary variable TEMP1
movlw   5                ;Load 5 into W
movlw   10               ;Load D'10' or H'10' or B'10' into W
                         ;depending upon default representation
movwf   TEMP1            ;Move W into TEMP1
movwf   TEMP1,F          ;Incorrect syntax
movf    TEMP1,W          ;Move TEMP1 into W
swapf   TEMP1,F          ;Swap 4-bit nibbles of TEMP1
swapf   TEMP1,W          ;Move TEMP1 to W, swapping nibbles
                         ;and leave TEMP1 unchanged
```

Increment/decrement/complement

```
incf   TEMP1,F      ;Increment TEMP1
incf   TEMP1,W      ;W <- TEMP1 + 1;  TEMP1 unchanged
decf   TEMP1,F      ;Decrement TEMP1
comf   TEMP1,F      ;Change 0s to 1s and 1s to 0s
```

Multiple Bit Manipulation:

andlw k	And literal value into W
andwf f, F(W)	And W with F and put the result in W or F
andwf f, F(W)	And W with F and put the result in W or F
iorlw k	inclusive-OR literal value into W
iorwf f, F(W)	inclusive-OR W with f and put the result in F or W
xorlw k	Exclusive-OR literal value into W
xorwf f, F(W)	Exclusive-OR W with f and put the result in F or W

Example:

```
andlw   B'00000111'     ;Force upper 5 bits of W to zero
andwf   TEMP1,F         ;TEMP1 <- TEMP1 AND W
andwf   TEMP1,W         ;W <- TEMP1 AND W
iorlw   B'00000111'     ;Force lower 3 bits of W to one
iorwf   TEMP1,F         ;TEMP1 <- TEMP1 OR W
xorlw   B'00000111'     ;Complement lower 3 bits of W
xorwf   TEMP1,W         ;W <- TEMP1 XOR W
```

Addition/Subtraction:

addlw k	Add the literal value to W and store the result in W
addwf f, F(W)	Add W to f and store the result in F or W
sublw k	Subtract the literal value from W and store the result in W
subwf f, F(W)	Subtract f from W and store the result in F or W

Example:

```
addlw   5             ;Add 5 to W
addwf   TEMP1,F       ;TEMP1 <- TEMP1 + W
sublw   5             ;W <- 5 - W (not W <- W - 5!)
subwf   TEMP1,F       ;TEMP1 <- TEMP1 - W
```

Rotate:

rlf f, F(W)	Copy f into F or W; rotate F or W left through the carry bit
rrf f, F(W)	Copy f into F or W; rotate F or W right through the carry bit

Example:

```
rlf    TEMP1,F        ;Nine-bit left rotate through C
                      ;(C <- TEMP1,7 ; TEMP1,i+1 <- TEMP1,i
                      ;TEMP1,0 <- C)
rrf    TEMP1,W        ;Leave TEMP1 unchanged
                      ;copy to W and rotate W right through C
```

Conditional Branch:

btfsc f, b	Test 'b' bit of the register f and skip the next instruction if bit is clear
btfss f, b	Test 'b' bit of the register f and skip the next instruction if bit is set
decfsz f, F(W)	Decrement f and copy the result to F or W; skip the next instruction if the result is zero
incfcz f, F(W)	Increment f and copy the result to F or W; skip the next instruction if the result is zero

Example:

```
btfsc   TEMP1,0        ;Skip the next instruction if bit 0 of
                       ;TEMP1 equals zero
btfss   STATUS,C       ;Skip if C = 1

decfsz TEMP1,F         ;Decrement TEMP1; skip if zero
incfsz TEMP1,W         ;Leave TEMP1 unchanged, skip if
                       ;TEMP1 = H'FF'; W <- TEMP1 + 1
```

Call/Go to/Return/Return from interrupt

goto label	Go to the instruction with the label "label"
call label	Go to the subroutine "label", push the Program Counter in the stack
retrun	Return from the subroutine, POP the Program Counter from the stack
retlw k	Retrun from the subroutine, POP the Program Counter from the stack; put k in W
retie	Return from Interrupt Service Routine and re-enable interrupt

Miscellaneous:

clrwdt	Clear Watch Dog Timer
sleep	Go into sleep/ stand by mode
nop	No operation

Program Documentation

- Good code documentation is essential if programs are to be maintained.
- The header should provide all the important processor details and identify the programmer. Most importantly, it should contain a FUNCTION statement which tells the reader what the processor needs to be connected to, exactly which I/O pins are connected to which devices and what the program does.
- Labels should be meaningful. They should help to make your code more readable. Try to avoid using labels which may be reserved words (see assembler directives).
- Comments should be clear and concise. They should summarise important functionality. Comments often summarise the function of several lines by using \ and / characters to tie lines together (see code examples).
- A clear columnar structure also helps code to be more readable. Separating equate and sub-routine components and providing short headings for each also makes the code easier to understand.

Code Structure and Documentation

```
; ----- GENERAL EQUATES -----------------------------------------------
W       EQU     0
F       EQU     1
RBIF    EQU     0
RBIE    EQU     3
GIE     EQU     7
; ----- I/O EQUATES     -----------------------------------------------
PORTA   EQU     0X05    ;
PORTB   EQU     0X06    ;
; ----- REGISTER EQUATES -----------------------------------------------
INTCON  EQU     0X0B    ;
MCOUNT  EQU     0X0C    ;
NCOUNT  EQU     0X0D    ;
LED_VAL EQU     0X0E    ;
TEMP_W  EQU     0X0F    ;
; ---------------------------------------------------------------------
        ORG     0X00
        GOTO    START
        ORG     0X04
        GOTO    INT_SER
```

Code Structure and Documentation

```
; ----------------------- MAIN PROGRAMME ------------------------------------
START   MOVLW    0XFF     ;- CONFIGURE PORTA AS INPUTS
        TRIS     PORTA    ;/
        MOVLW    0X80

        …


        END
```

Subroutines

- Subroutines are a sequence of instructions for performing a particular task. They generally make code more efficient because their functions can be re-used.
- Subroutines are normally placed before the main program after the ORG and GOTO lines.
- They are implemented using CALL and RETURN (or RETLW).
- When a CALL instruction is encountered, the program counter is "pushed" onto the stack. A new value is loaded into the program counter and instruction execution starts from the new address.
- When a RETURN or RETLW instruction is encountered, the program counter is restored by "popping" the stack.
- You should use a subroutine when you need to perform a task and then continue with a previous task (otherwise, use GOTO.)
- Can a subroutine be called from within another? Yes. The limit to the depth of nesting is the depth of the program counter stack. The PIC16F84 has a program stack depth of 8.