Bit-wise Logical operations

Bitwise AND operation

(w AND f)	andwf	*floc*, d	*d* ← [*floc*] & w	j = j & i;
(w AND literal)	andlw	*k*	w ← 0xkk & w	j = j & 0xkk;

Bitwise OR operation

(w OR f)	iorwf	*floc*, d	*d* ← [*floc*] | w	j = j | i;
(w OR literal)	iorwf	*floc*, d	*d* ← 0xkk | w	j = j | 0xkk;

Bitwise XOR operation

(w XOR f)	xorwf	*floc*, d	*d* ← [*floc*] ^ w	j = j ^ i;
(w XOR literal)	xorwf	*floc*, d	*d* ← 0xkk ^ w	j = j ^ 0xkk

Bitwise complement operation;

(~ f)	comf	*floc*, d	*d* ← ~[*floc*]	j = ~ i ;

Clearing a group of bits

Data Memory

Location	contents
(i) 0x20	0x2C
(j) 0x21	0xB2
(k) 0x22	0x8A

Clear upper four bits of i .

In *C*:

```
i = i &0x0f;      ← The 'mask'
```

In PIC assembly

```
movf   0x20, w   ; w = i
andlw  0x0f      ; w = w & 0x0f
movwf  0x20      ; i = w
```

```
i =    0x2C  =  0010 1100
                &&&& &&&&
mask= 0x0F   =  0000 1111
                ---------
result =        0000 1100
       =        0x0C
```

AND: mask bit = '1', result bit is same as operand.
mask bit = '0', result bit is cleared

Setting a group of bits

Data Memory

Location	contents
(i) 0x20	0x2C
(j) 0x21	0xB2
(k) 0x22	0x8A

Set bits b3:b1 of j

In *C*:

```
j = j |0x0E;      ← The 'mask'
```

In PIC assembly

```
movf   0x21, w   ; w = j
iorlw  0x0E      ; w = w | 0x0E
movwf  0x21      ; j = w
```

```
j =    0xB2  =  1011 0010
                |||| ||||
mask= 0x0E   =  0000 1110
                ---------
result =        1011 1110
       =        0xBE
```

OR: mask bit = '0', result bit is same as operand.
mask bit = '1', result bit is set

Complementing a group of bits

Data Memory

Location	contents
(i) 0x20	0x2C
(j) 0x21	0xB2
(k) 0x22	0x8A

Complement bits b7:b6 of k

In *C*:

```
k = k ^0xC0;      ← The 'mask'
```

In PIC assembly

```
movf   0x22, w   ; w = k
xorlw  0xC0      ; w = w ^ 0xC0
movwf  0x22      ; k = w
```

```
k =    0x8A  =  1000 1010
                ^^^^ ^^^^
mask= 0xC0   =  1100 0000
                ---------
result =        0100 1010
       =        0x4A
```

XOR: mask bit = '0', result bit is same as operand.
mask bit = '1', result bit is complemented

Complementing all bits

Data Memory

Location	contents
(i) 0x20	0x2C
(j) 0x21	0xB2
(k) 0x22	0x8A

Complement all bits of k

In *C*:

```
k = !k ;
```

In PIC assembly

```
comf  0x22, f    ; k = !k
```

```
k =    0x8A  =  1000 1010

After complement

result =        0111 0101
       =        0x75
```


Bit set, Bit Clear instructions

Can set/clear **one** bit of a data memory location by using the AND/OR operations, but takes three instructions as previously seen.

The bit clear (**bcf**) and bit set (**bsf**) instructions clear/set one bit of data memory using one instruction.

	B B B B B B B B B B B B B B B B
	1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0
	3 2 1 0 9 8 7 6 5 4 3 2 1 0
bcf *floc*, b	0 1 0 0 b b b k k k k k k k
bsf *floc*, b	0 1 0 1 b b b k k k k k k k

'kkkkkkk' lower 7-bits of memory location
'bbb' three bit value that specifies affected bit

Bit clear/set examples

Clear b7 of k, Set b2 of j

In *C*:

```
k = k & 0x7F;
j = j | 0x04;
```

In PIC assembly

```
bcf  0x22, 7    ; k = bitclear(k,7)
bsf  0x21, 2    ; j = bitset(j,2)
```

Data Memory

Location	contents
(i) 0x20	0x2C
(j) 0x21	0xB2
(k) 0x22	0x8A

```
                        bbbb bbbb
                        7654 3210
k =   0x8A  =           1000 1010
bitclear(k,7)
k =   0x8A  =           0000 1010

j =   0xB2  =           1011 0010
bitset(j,2)
j =   0xB6  =           1011 0110
```

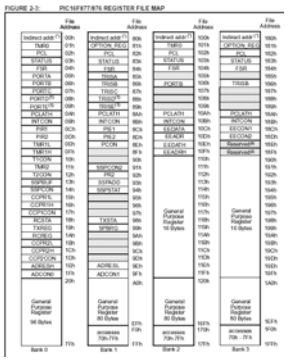

Changing Register Banks

The file registers are split into 4 banks, each bank has 128 locations (0x80).

The lower 7-bits of a location are specified in the machine code.

But need two more bits to specify the bank!!!

These bits are located in the status register (location 0x3) and the default value is 00 (bank 0).



status Register

The ***status*** register is a special purpose register (like the *w* register). Each bit in the status register has a particular meaning, and either controls an operation mode of the PIC or indicates the status of an operation.

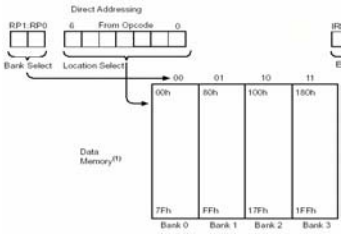
TABLE 2-1: SPECIAL FUNCTION REGISTER SUMMARY

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on: POR, BOR	Details on page:
Bank 0											
00h(3)	INDF	Addressing this location uses contents of FSR to address data memory (not a physical register)								xxxx xxxx	27
01h	TMR0	Timer0 Module Register								xxxx xxxx	47
02h(3)	PCL	Program Counter (PC) Least Significant Byte								0000 0000	26
03h(3)	STATUS	IRP	RP1	RP0	TO	PD	Z	DC	C	0001 1xxx	18

Table 2-1 in the PIC 16F873 data sheets contains the definitions of all special registers. We will cover the special registers as we get to the features of the PIC that use them.

Bank Select Bits (RP1:RP0)

Bits b6:b5 of the status register are known as RP1:RP0 and control the bank selection when the register file is accessed.



The status of the RP1:RP0 must be known to determine what data memory location is accessed.

Bank Selection Example

Complement bits b7:b6 of k

In *C*:

```
k = k ^ 0xC0;
```
← The 'mask'

In PIC assembly

```
movf  0xA2, w   ; w = k
xorlw 0xC0      ; w = w ^ 0xC0
movwf 0xA2      ; k = w
```

Data Memory

Location	contents
(i) 0xA0	0x2C
(j) 0xA1	0xB2
(k) 0xA2	0x8A

Locations now in Bank1

Upper bit of address is ignored, only works if RP1:RP0 = 01

Bank Selection Example (cont)

Complement bits b7:b6 of k

In *C*:

```
k = k ^ 0xC0;
```
← The 'mask'

In PIC assembly

```
bsf  0x3, 5     ; RP0 = 1
movf 0xA2, w    ; w = k
xorlw 0xC0      ; w = w ^ 0xC0
movwf 0xA2      ; k = w
bcf  0x3, 5     ; RP0 = 0
```

Status register at location 0x3. Change to bank 1 via RP0 = 1, assume RP1 already 0

Change back to bank0, not necessary, but a safe practice.

Data Memory

Location	contents
(i) 0xA0	0x2C
(j) 0xA1	0xB2
(k) 0xA2	0x8A

Locations now in Bank1

Using Labels for Clarity

```
INCLUDE "p16f873.inc"

;  Register Usage
CBLOCK 0x0A0
 i, j,k
ENDC

 org     0
 ; k = k ^ 0xC0;
 bsf     STATUS,RP0
 movf    k         ; w <- k
 xorlw   0xC0      ; w <- w^0xC0
 movwf   k         ;  k <- w
 bcf     STATUS, RP0

here
  goto    here      ; loop forever
  end
```

Bank1 space

Labels defined in "p16F873.inc". Improves code clarity.

Labels defined for special bits, registers in "p16F873.inc" are all uppercase, match datasheet names as closely as possible.

Carry, Zero Flags

Bit 0 of the status register is known as the **carry** (C) flag.

Bit 2 of the status register is known as the **zero** (Z) flag.

These flags are set as **side-effects** of particular instructions or can be set/cleared explicitly using the *bsf/bcf* instructions.

How do you know if an instruction affects C,Z flags?

Look at Table 13-2 in PIC datasheeet.– *addwf* affects C, DC, Z flags.

TABLE 13-2: PIC16F87X INSTRUCTION SET

Mnemonic, Operands	Description	Cycles	14-Bit Opcode MSb	LSb	Status Affected
BYTE-ORIENTED FILE REGISTER OPERATIONS					
ADDWF f, d	Add W and f	1	00	0111 dfff ffff	C,DC,Z

Addition: Carry, Zero Flags

Zero flag is set if result is zero.

In addition, carry flag is **set** if there is a carry out of the MSB (unsigned overflow, result is greater > 255)

```
 0xF0      0x00      0x01      0x80
+0x20     +0x00     +0xFF     +0x7F
--------  --------  --------  --------
 0x10     0x00      0x00      0xFF
 Z=0,     Z=1,      Z=1,      Z=0,
 C=1      C=0       C=1       C=0
```


Subtraction: Carry, Zero Flags

Zero flag is set if result is zero.

In subtraction, carry flag is **cleared** if there is a borrow from the MSB (unsigned underflow, result is < 0, larger number subtracted from smaller number). Carry flag is **set** if no borrow occurs.

```
 0xF0      0x00      0x01
- 0x20    -0x00     -0xFF
--------  --------  --------
 0xD0     0x00      0x02
 Z=0,     Z=1,      Z=0,
 C=1      C=1       C=0
```

For a subtraction, the combination of Z=1, C=0 will not occur.

Conditional Execution using Bit Test

The 'bit test f, skip if clear' (*btfsc*) and 'bit test f, skip if set' (btfss) instructions are used for conditional execution.

btfsc *floc*, b ; skips next instruction is bit 'b' of *floc* is clear ('0')

btfss floc, b ; skips next instruction is bit 'b' of *floc* is set ('1')

Bit test instructions used on status flags implements tests such as equality (==), inequality (!=), greater than (>), less than (<), greater than or equal (<=) , less than or equal (>=)

Equality Test (==)

C code

```
unsigned char i,j;


if (i == j) {
    j = i + j;
 }
 /* ..do stuff..*/
```

PIC Assembly

```
    movf  i,w          ; w ← i
    subwf j,w          ; w ← j - w
    btfss STATUS, Z    ; Z=1?
    goto  skip         ; Z=0, i != j
    movf  i,w          ; w ← i
    addwf j,f          ; j ← j + w
skip
    ..do stuff..
```

Subtraction operation of j-i performed to check equality; if i == j then subtraction yields '0', setting the Z flag.

Unsigned greater-than Test (>)

C code

```
unsigned char i,j;


if (i > j) {
    j = i + j;
 }
 /* ..do stuff..*/
```

PIC Assembly

```
   movf  i,w          ; w ← i
   subwf j,w          ; w ← j - i
   btfsc STATUS, C    ; C=0?
   goto  skip         ; C=1,i <= j
   movf  i,w          ; w ← i
   addwf j,f          ; j ← j + w
skip
    ..do stuff..
```

If i > j, then j-i will result in a borrow (C=0). Subtraction operation of j-i performed so test on C flag could be done.

Unsigned greater-than-or-equal Test (>=)

C code

```
unsigned char i,j;


if (i >= j) {
    j = i + j;
 }
 /* ..do stuff..*/
```

PIC Assembly

```
   movf  j,w          ; w ← j
   subwf i,w          ; w ← i - j
   btfss STATUS, C    ; C=1?
   goto  skip         ; C=0,i < j
   movf  i,w          ; w ← i
   addwf j,f          ; j ← j + w
skip
    ..do stuff..
```

If (i >= j), then i – j will produce no borrow if i > j or i == j.

Zero/Non-Zero Test

C code

```
unsigned char i,j;


if (!i) {
 /* do this if i
zero */
    j = i + j;
 }
 /* ..do stuff..*/
```

PIC Assembly

```
   movf  i,f          ; i ← i
   btfss STATUS, Z    ; Z=1?
   goto  skip         ; Z=0, i!=0
   movf  i,w          ; w ← i
   addwf j,f          ; j ← j + w
skip
    ..do stuff..
```

The *movf* instruction just moves *i* back onto itself! Does no useful work except to affect the Z flag.

Aside: *C* conditional tests

A *C* conditional test is true if the result is non-zero; false if the result is zero.

The ! operator is a logical test that returns 1 if the operator is equal to '0', returns '0' if the operator is non-zero.

```
if (!i) {
/* do this if i zero */
    j = i + j;
 }
```

```
 if (i) {
 /* do this if i non-zero */
     j = i + j;
  }
```

Could also write:

```
if (i == 0) {
/* do this if i zero */
    j = i + j;
 }
```

```
 if (i != 0) {
 /* do this if i non-zero */
     j = i + j;
  }
```


C equality tests

A common C code mistake is shown below (= vs ==)

```
if (i = 5) {
    j = i + j;
 }/*wrong*/
```

Always executes because i=5 returns 5, so conditional test is always non-zero, a true value. The = is the assignment operator.

```
if (i == 5) {
    j = i + j;
 }/*right*/
```

The test i == 5 returns a 1 only when i is 5. The == is the equality operator.

while loop

C code

```
unsigned char i,j;


while (i > j) {
    j = i + j;
 }
 /* ..do stuff..*/
```

PIC Assembly

```
loop_top
   movf  i,w          ; w ← i
   subwf j,w          ; w ← j - i
   btfsc STATUS, C    ; C=0?
   goto  skip         ; C=1,i <= j
   movf  i,w          ; w ← i
   addwf j,f          ; j ← j + i
   goto loop_top
skip
    ..do stuff..
```

Jump back to *loop_top* after body is performed. The body of a while loop may not execute if loop test is initially false.

do-while loop

C code

```
unsigned char i,j;


do {
    j = i + j;
 }while (i > j)
/* ..do stuff..*/
```

PIC Assembly

```
loop_top
    movf  i,w          ; w ← i
    addwf j,f          ; j ← j + i
   ; conditional test
    movf  i,w          ; w ← i
    subwf j,w          ; w ← j – i
    btfss STATUS, C    ; C=1?
    goto  loop_top     ; C=0,i >j
skip
     ..do stuff..
```

In *do-while* loop, body is always executed at least once.

Aside: *for* loops in *C*

A *for* loop is just another way to write a *while* loop. Typically used to implement a counting loop (a loop that is executed a fixed number of times).

```
unsigned char i,j;

i = 0;
while (i != 10) {
    k = k + j;
    i++;
 }
 /* ..do stuff..*/
```

executed once, before test.

executed each loop iteration after body

```
unsigned char i,j;

for (i = 0; i!= 10; i++){
    k = k + j;
 }
 /* do stuff */
```

loop test

These statements executed 10 times. Both code blocks are equivalent.

Decrement/Increment, skip if 0

For simple counting loops, where goal is to execute a block of statements a fixed number of times, the 'decrement/increment, skip if 0' instructions can be useful.

decfsz *floc* ; decrement floc, skips next instruction if result == 0

incfsz *floc* ; increment floc, skips next instruction if result == 0

Can use these for counting loops; replaces multiple instructions with single instruction. The reason to use these instructions would be to save code space, and decrease loop execution time.

Counting Loop Example

C code

```
unsigned char i,j;

 i = 10;
 do {
    k = k + j;
    i--;
 }while(i != 0);
/* do stuff */
```

PIC Assembly

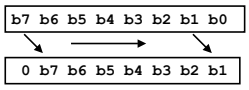
```
    movlw 0x0A
    movwf i          ; i = 10
loop_top
    movf  j,w        ; w ← j
    addwf k,f        ; k ← k + j
    decfsz i         ; i--, skip if 0
    goto loop_top    ; i non-zero

    ..do stuff..
```

Makes for an efficient end-of-loop action and test in some cases. Usage of incfsz/decfsz is optional as other instruction sequences can accomplish the same thing. Use what you understand!!!

C unsigned Shift Left, Shift Right

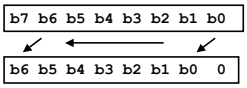
unsigned Shift right i >> 1
all bits shift to right by one, '0' into MSB.



original value

i >> 1 (right shift by one)

unsigned Shift left i << 1
all bits shift to left by one, '0' into LSB.



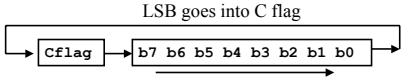
original value

i << 1 (left shift by one)

PIC Rotate Left/Right

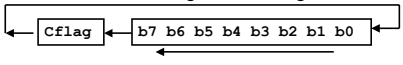
PIC has *rotate right by 1* and *rotate left by 1* instructions

rrf *floc*, d ; d shifted to right by 1, MSB gets C flag, LSB goes into C flag



rotate right

rlf *floc*, d ; d shifted to right by 1, LSB gets C flag, MSB goes into C flag



rotate left

Rotate Left/Right Examples

C code

```
unsigned char i,j,k;
i = i >> 1;
j = j << 1;
k = k >> 3;
```

PIC Assembly

```
bcf STATUS,C    ; clear C
rrf  i,f        ; i >> 1

bcf STATUS,C    ; clear C
rlf  j,f        ; j << 1

bcf STATUS,C    ; clear C
rrf  k,f        ; k >> 1
bcf STATUS,C    ; clear C
rrf  k,f        ; k >> 1
bcf STATUS,C    ; clear C
rrf  k,f        ; k >> 1
```

For multiple shift, repeat single shift. Must clear C flag as status is unknown usually.

Why Shift?

Shift right by 1 is divide by 2 (i >> 1 == i / 2)

Shift left by 1 is multiply by 2 (i << 1 == i * 2)

If need to multiply a variable by a constant can do it by shifts, adds or shifts/subtracts.

i = i * 7
= i (8 – 1)
= (i * 8) - i
= (i << 3) - i

```
bcf   STATUS,C    ; clear C
rlf   i,f         ; i << 1
bcf   STATUS,C    ; clear C
rlf   i,f         ; i << 1
bcf   STATUS,C    ; clear C
rlf   i,f         ; i << 1
movf  i,w         ; w ← i
subwf i,f         ; i ← i -w
;; new i is old_i * 7
```


PIC18 Comparison Instructions

The PIC18 has three instructions that directly implement ==, > (unsigned), and < (unsigned).

cmpfseq *floc* ; if *floc* == w, skip next instruction

cmpfsgt *floc* ; if *floc* > w, skip next instruction

cmpfslt *floc* ; if *floc* < w, skip next instruction

Instructions like this are commonly found in other μP instruction sets.

PIC18 comparison example (==)

```
if (i == j) {
    j = i + j;
}
```

C code

PIC16 Assembly

```
movf  i,w      ; w ← i
subwf j,w      ; w ← j – i
btfss STATUS, Z ; Z=1?
goto  skip     ; Z=0,i != j
movf  i,w      ; w ← i
addwf j,f      ; j ← j + w
skip
  ..do stuff..
```

PIC 18 assembly

```
movf  i,w   ; w ← i
cmpfseq j   ; j == i?
goto  skip  ; j != i
addwf j,f   ; j ← j + i
skip
    ..do stuff..
```

PIC 18 *cmpfseq* instruction does not change *w*, so *w* still has *i* in it for *addwf* instruction. Takes four PIC16 instructions vs six PIC18 instructions.

What do you need to know?

- Logical operations (and,or,xor, complement)
- Clearing/Setting/Complementing groups of bits
- Accessing different banks via RP1, RP0
- Bit set/clear/test instructions
- ==, !=, >, <, >=, <= tests on unsigned variables
- Loop structures
- Shift left (>>), Shift Right (<<) using rotate instructions
- Multiplication by a constant via shifts/adds/subtracts
- PIC18 unsigned comparison