C Arithmetic operators

<table>
<tr><th>Operator</th><th>Description</th></tr>
<tr><td>+, -</td><td>addition (i+j), subtraction (i-j)</td></tr>
<tr><td>*, /</td><td>multiplication (i*j), division (i/j)</td></tr>
<tr><td>++, --</td><td>increment (i++), decrement (j--)</td></tr>
<tr><td>&, |, ^</td><td>bitwise AND (i &j), OR (i | j), Exclusive OR (i ^ j)</td></tr>
<tr><td>~</td><td>bitwise complement (~i)</td></tr>
<tr><td>>>, <<</td><td>Right shift (i << 1), left shift (j >> 1)</td></tr>
</table>

The above are *C* operators that we would like to implement in PIC18 assembly language. Multiplication and division will be covered in a later lecture.

Bit-wise Logical operations

Bitwise AND operation

(w AND f)	andwf *floc*, d	$d \leftarrow$ (*floc*) & (w)	j = j & i;
(w AND literal)	andlw *k*	w $\leftarrow$ (0x*kk*) & (w)	j = j & 0x*kk*;

Bitwise OR operation

(w OR f)	iorwf *floc*, d	$d \leftarrow$ (*floc*) | (w)	j = j | i;
(w OR literal)	iorlw *floc*, d	$d \leftarrow$ 0x*kk* | (w)	j = j | 0x*kk*;

Bitwise XOR operation

(w XOR f)	xorwf *floc*, d	$d \leftarrow$ (*floc*) ^ (w)	j = j ^ i;
(w XOR literal)	xorlw *floc*, d	$d \leftarrow$ 0x*kk* ^ (w)	j = j ^ 0x*kk*

Bitwise complement operation;

(~ f)	comf *floc*, d	$d \leftarrow$ ~ (*floc*)	j = ~ i ;

Clearing a group of bits

Data Memory

	Location	contents
(i)	0x020	0x2C
(j)	0x021	0xB2
(k)	0x022	0x8A

Clear upper four bits of i .

In *C*:

i = i & 0x0f; ← The 'mask'

In PIC18 assembly

```
movf    0x020, w    ; w = i
andlw   0x0f        ; w = w & 0x0f
movwf   0x020       ; 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)	0x020	0x2C
(j)	0x021	0xB2
(k)	0x022	0x8A

Set bits b3:b1 of j

In *C*:

j = j | 0x0E; ← The 'mask'

In PIC18 assembly

```
movf    0x021, w    ; w = j
iorlw   0x0E        ; w = w | 0x0E
movwf   0x021       ; 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)	0x020	0x2C
(j)	0x021	0xB2
(k)	0x022	0x8A

Complement bits b7:b6 of k

In *C*:

```
k = k ^ 0xC0;
```

← The 'mask' (0xC0)

In PIC18 assembly

```
movf    0x022, w    ; w = k
xorlw   0xC0        ; w = w ^ 0xC0
movwf   0x022       ; 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

Complement all bits of k

In *C*:

k = ~k ;

In PIC18 assembly

```
comf  0x022, f    ; k = ~k
```

Data Memory

Location		contents
(i)	0x020	0x2C
(j)	0x021	0xB2
(k)	0x022	0x8A

```
k =    0x8A   =  1000 1010

After complement

    result    =  0111 0101
              =  0x75
```

Bit set, Bit Clear, Bit Toggle instructions

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

The bit clear (**bcf**), bit set (**bsf**), bit toggle (**btg**) instructions clear/set/complement one bit of data memory using one instruction.

	B15	B14	B13	B12	B11	B10	B09	B08	B07	B06	B05	B04	B03	B02	B01	B00
bcf *floc, b [,a]*	1	0	0	1	*b*	*b*	*b*	*a*	*f*	*f*	*f*	*f*	*f*	*f*	*f*	*f*
bsf *floc*, *b [,a]*	1	0	0	0	*b*	*b*	*b*	*a*	*f*	*f*	*f*	*f*	*f*	*f*	*f*	*f*
btg *floc*, *b [,a]*	0	1	1	1	*b*	*b*	*b*	*a*	*f*	*f*	*f*	*f*	*f*	*f*	*f*	*f*

'fffffff' lower 7-bits of memory location

'bbb' three bit value that specifies affected bit

Bit clear/set/toggle examples

Data Memory

Location	contents
(i) 0x020	0x2C
(j) 0x021	0xB2
(k) 0x022	0x8A

Clear bit 7 of k, Set bit 2 of j, complement bit 5 of i.

In *C*:

```
k = k & 0x7F;
j = j | 0x04;
i = i ^ 0x20;
```

In PIC18 assembly

```
bcf  0x022, 7    ; k = bitclear(k,7)
bsf  0x021, 2    ; j = bitset(j,2)
btg  0x020, 5
```

```
                     bbbb bbbb
                     7654 3210

k =   0x8A    =   1000 1010
bcf k,7
k =   0x0A    =   0000 1010

j =   0xB2    =   1011 0010
bsf j,2
j =   0xB6    =   1011 0110

i =   0x2C    =   0010 1100
btg  i,5
i =   0x0C    =   0000 1100
```

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 (>=)

Number Sequencing Task using *btfsc*

```
(1)       CBLOCK 0x0
 (2)        loc,out        ;byte variables
 (3)       ENDC

 (4)       org      0
 (5)       goto     main
 (6)       org      0x0200
 (7)     main
 (8)       ;movlw    0       ;uncomment for loc=0
 (9)       movlw    1       ;uncomment for loc=1
 (10)      movwf    loc     ;initialize loc
 (11)    Ltop
 (12)      btfsc    loc,0   ; skip next if loc(0) is '0'
 (13)      goto     loc_lsb_is_1
 (14)      ;LSb of loc = 0 if reach here
 (15)      movlw    3       ; W ← 3
 (16)      movwf    out     ; out ← (W)
 (17)      movlw    2       ; W ← 2
 (18)      movwf    out     ; out ← (W)
 (19)      movlw    4       ; W ← 4
 (20)      movwf    out     ; out ← (W)
 (21)    loc_lsb_is_1
 (22)      movlw    8       ; W ← 8
 (23)      movwf    out     ; out ← (W)
 (24)      movlw    5       ; W ← 5
 (25)      movwf    out     ; out ← (W)
 (26)      movlw    6       ; W ← 6
 (27)      movwf    out     ; out ← (W)
 (28)      movlw    1       ; W ← 1
 (29)      movwf    out     ; out ← (w)
 (30)      goto     Ltop    ; loop forever
 (31)      end
```

Skip “goto loc_lsb_is_1”

location if loc = 1.

Copyright Thomson/Delmar Learning 2005. All Rights Reserved.

status Register

The ***STATUS*** register is a special purpose register (like the *w* register). The lower 5 bits of the status register are one bit flags that are set/cleared as side effects of an instruction execution.

7	6	5	4	3	2	1	0
u	u	u	N	OV	Z	DC	C

u – unimplemented
N – negative flag (set if MSB of result = 1)
OV – 2's complement overflow flag
Z – zero flag (set if result = 0)
DC – decimal carry (carry/(~borrow) from bit 3 to bit 4)
C – carry flag (carry/(~borrow) out of MSB

We will **not** discuss the DC flag, it is used in Binary Coded Decimal arithmetic.

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 20-2 in PIC18Fxx2 datasheeet.– *addwf* affects all flags, *movf* only Z,N flags.

```
  Mnemonic     Desc.            Instr     Mach. Code        Status
                               Cycles                       affected
ADDWF f,d,a |ADD WREG and F| 1|0010 01da ffff ffff|C,DC,Z,OV,N
MOVF  f,d,a |Move f         | 1|0101 00da ffff ffff| Z,N
```

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 Z=0, C=1	0x00 Z=1, C=0	0x00 Z=1, C=1	0xFF Z=0, 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.

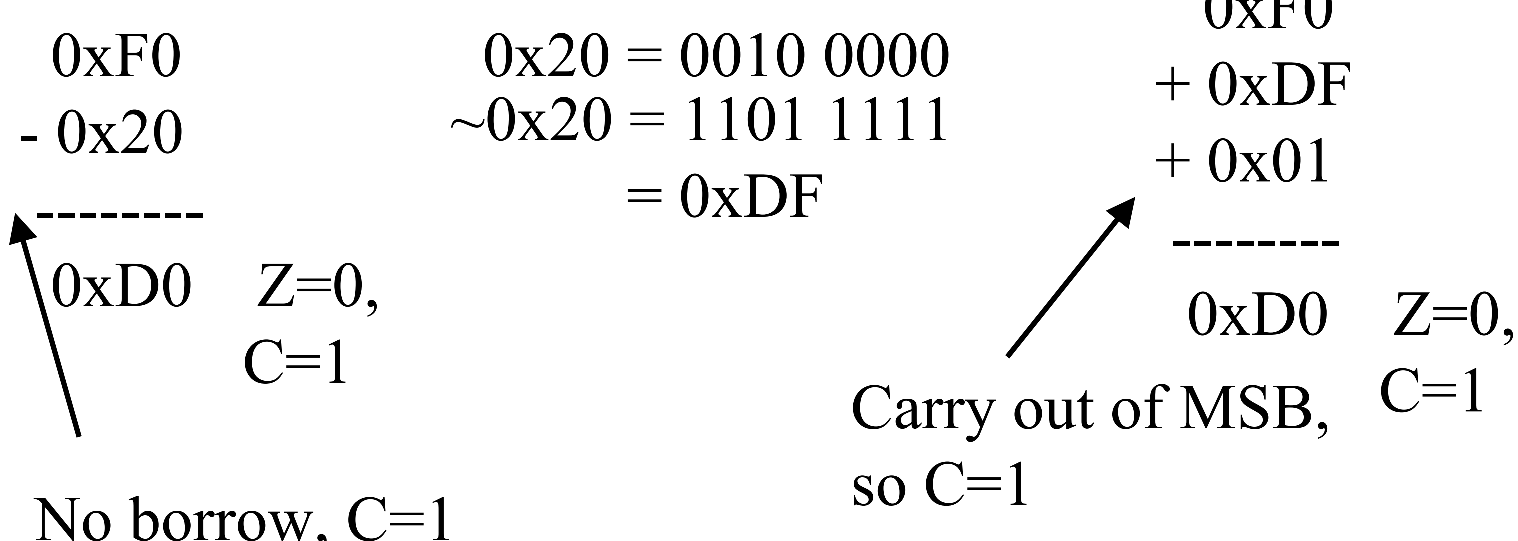
0xF0	0x00	0x01
- 0x20	-0x00	-0xFF
--------	--------	--------
0xD0 Z=0, C=1	0x00 Z=1, C=1	0x02 Z=0, C=0

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

How do you remember setting of C flag for Subtraction?

Subtraction of A – B is actually performed in hardware as A + (~B) + 1

The value (~B) + 1 is called the **two's complement** of B (more on this later). The C flag is affected by the addition of A + (~B) + 1



C Conditional Tests

<table>
<tr><th>Operator</th><th>Description</th></tr>
<tr><td>== , !=</td><td>equal, not-equal</td></tr>
<tr><td>>, >=</td><td>greater than, greater than or equal</td></tr>
<tr><td><, <=</td><td>less than, less than or equal</td></tr>
<tr><td>&&</td><td>logical AND</td></tr>
<tr><td>| |</td><td>logical OR</td></tr>
<tr><td>!</td><td>logical negation</td></tr>
</table>

If an operator used in a *C* conditional test, such as an IF statement or WHILE statement, returns nonzero, then the condition test is TRUE.

Logical Negation vs. Bitwise Complement

`!i` is not the same as `~i`

`i = 0xA0`

`!(i)` ⇨ `0`

`i = 0xA0`

`~(i)` ⇨ `0x5F`

Logical operations: !, &&, || always treat their operands as either being zero or non-zero, and the returned result is always either 0 or 1.

Examples of *C* Equality, Inequality, Logical, Bitwise Logical Tests

```
unsigned char a,b,a_lt_b, a_eq_b, a_gt_b, a_ne_b;

a = 5; b = 10;
a_lt_b = (a < b);    // a_lt_b result is 1
a_eq_b = (a == b);   // a_eq_b result is 0
a_gt_b = (a > b);    // a_gt_b result is 0
a_ne_b = (a != b);   // a_ne_b result is 1
```

```
unsigned char a_lor_b, a_bor_b, a_lneg_b, a_bcom_b;

(2)     a = 0xF0; b = 0x0F;
(3)     a_land_b = (a && b); //logical and, result is 1
(4)     a_band_b = (a & b);  //bitwise and, result is 0
(5)     a_lor_b = (a || b);  //logical or, result is 1
(6)     a_bor_b = (a | b);   //bitwise or, result is 0xFF
(7)     a_lneg_b = (!b);     //logical negation, result is 0
(8)     a_bcom_b = (~b);     //bitwise negation, result is 0xF0
```


Copyright Thomson/Delmar Learning 2005. All Rights Reserved.

if{} Statement Format in *C*

```
if (condition_test) {
    if_body      ←—— Executed when condition_test is non-zero (true)
} else {
    else_body    ←—— Executed when condition_test is zero (false)
}
```

if_body and *else_body* can contain multiple statements.

else_body is optional.

C zero/non-zero 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

'==' is the equality test in C; '=' is the assignment operator.

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

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

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

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

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

C Bitwise logical vs. Logical AND

The '&' operator is a bitwise logical AND. The '&&' operator is a logical AND and treats its operands as either zero or non-zero.

```
if (i && j) {
/* do this */
 }
```

is read as: If ((i is nonzero) AND (j is nonzero)) then *do this*.

```
if (i & j) {
/* do this */
 }
```

is read as: If ((i bitwise AND j) is nonzero)) then *do this*.

```
i = 0xA0, j = 0x0B;

(i && j)   →   1
```

```
i = 0xA0, j = 0x0B;

(i & j)   →   0x0
```

C Bitwise logical vs. Logical OR

The ‘|’ operator is a bitwise logical OR. The ‘||’ operator is a logical OR and treats its operands as either zero or non-zero.

```
if (i || j) {
/* do this */
 }
```

is read as: ⇒ If ((i is nonzero) OR (j is nonzero)) { do...

```
if (i | j) {
/* do this */
 }
```

is read as: ⇒ If ((i bitwise OR j) is nonzero)) { do....

```
i = 0xA0, j = 0x0B;

(i || j)  ⇒  1
```

```
i = 0xA0, j = 0x0B;

(i | j)  ⇒  0xAB
```

Non-Zero Test

labels for SFRs and bit fields defined in *pic18f242.inc;* use

In *C*

```
unsigned char i, j;

if (i) {
 // do this if i is non-zero
j = i + j;
}
// ...rest of code...
```

In Assembly

```
  movf  i,f           ; i = i
  btfsc STATUS,Z      ; skip if Z=0
  goto  end_if        ; Z=1, i is 0
  movf  i,w           ; w = i
  addwf j,f           ; j = j + i
end_if
  ..rest of code..
```

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

Copyright Thomson/Delmar Learning 2005. All Rights Reserved.

Conditional Execution using branches

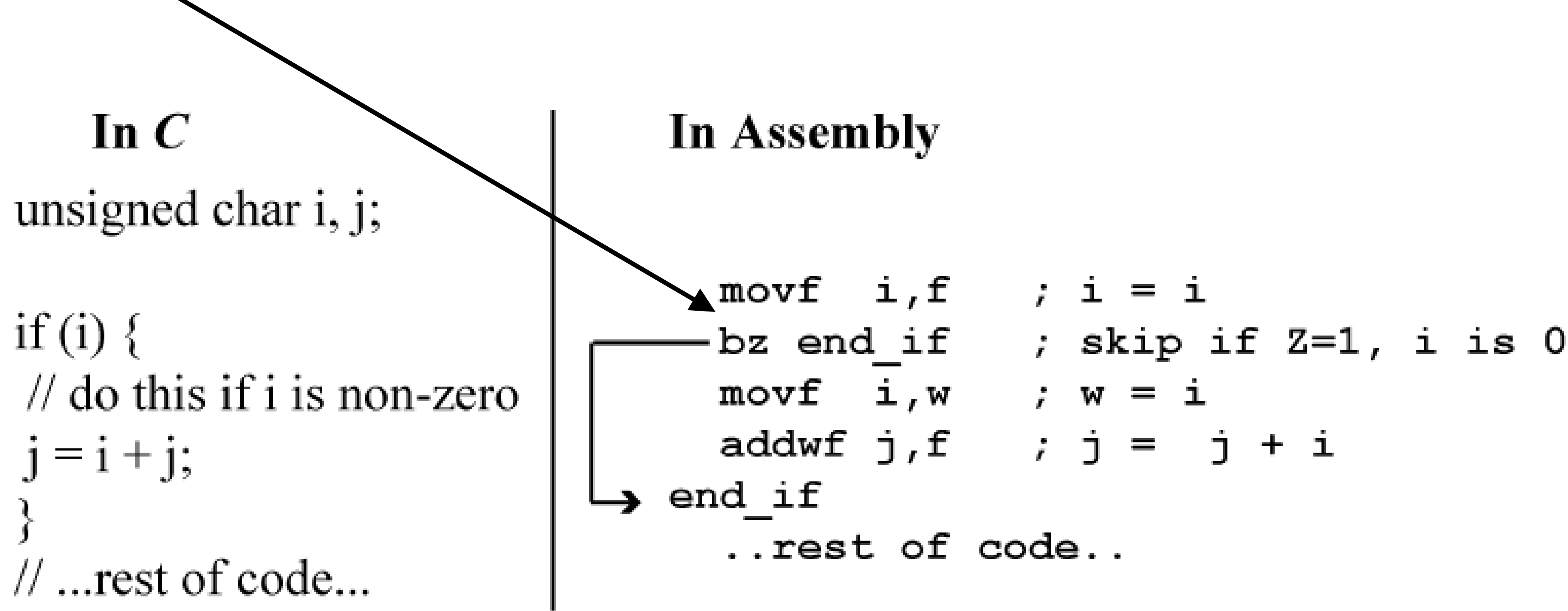
A *branch* functions as a conditional *goto* based upon the setting of a single flag

bc (branch if carry), *bnc* (branch if not carry)
bov (branch on overflow), *bnov* (branch if no overflow)
bn (branch if negative), *bnn* (branch if not negative)
bz (branch if zero), *bnz* (branch if not zero)
bra (branch always)

Using branch instructions instead of btfsc/btfss generally results in fewer instructions, and improves code clarity.

Non-Zero Test

The *bz* (branch if Zero, Z=1) replaces the *btfsc/goto* combination.



For a non-zero test *if(!i){}* replace *bz* with *bnz*

Copyright Thomson/Delmar Learning 2005. All Rights Reserved.

Equality Test (==)

In *C*

```
unsigned char i, j;

if (i == j) {
  // do this if i equal to j
  j = i + j;
}
// ...rest of code...
```

In Assembly

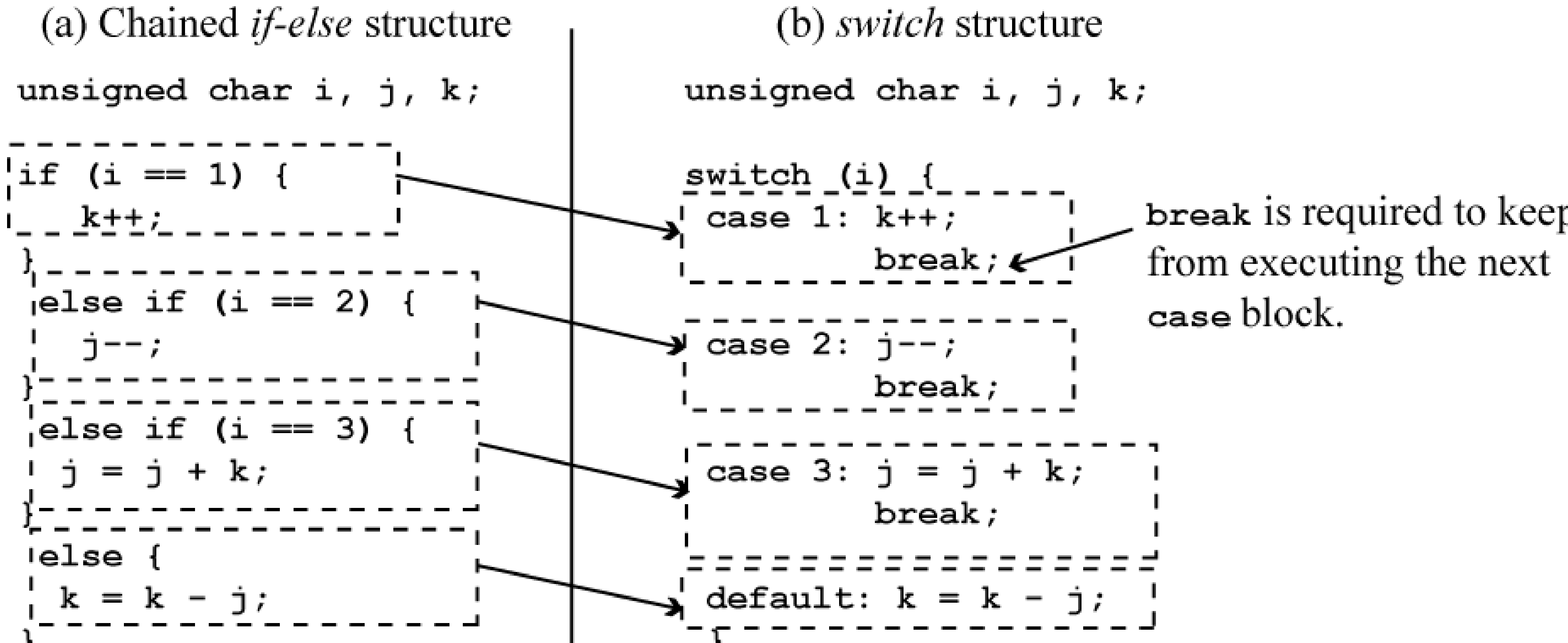
```
    movf  j,w     ; w = j
    subwf i,w     ; w =  i - j
 ┌─ bnz end_if    ; skip if Z=0, i != j
 │  movf  i,w     ; w = i
 │  addwf j,f     ; j =  j + i
 └→end_if
    ..rest of code..
```

Subtraction operation of i-j performed to check equality;

if i == j then subtraction yields ‘0’, setting the Z flag. Does not matter if i-j or j-i is performed.

Copyright Thomson/Delmar Learning 2005. All Rights Reserved.

switch Statement in *C*



A *switch* statement is a shorthand version of an *if-else* chain where the same variable is compared for equality against different values.

Copyright Thomson/Delmar Learning 2005. All Rights Reserved.

switch Statement in assembly language

In *C*

```
unsigned char i, j, k;
switch (i) {

  case 1: k++;
       break;

  case 2: j--;
       break;

  case 3: j = j + k;
       break;

  default: k = k - j;

}// end switch
```

In Assembly

```
   movlw 1             ; w = 1
   subwf i,w           ; i == 1?
   bnz   case_2
   incf  k             ; k++
   bra   end_switch    ; break statement
case_2
   movlw 2             ; w = 2
   subwf i,w           ; i == 2?
   bnz   case_3
   decf  j             ; j--
   bra   end_switch    ; break statement
case_3
   movlw 3             ; w = 3
   subwf i,w           ; i == 3?
   bnz   default
   movf  k,w
   addwf j,f           ; j = j + k
   bra   end_switch    ; break statement
default
   movf  j,w
   subwf k,f           ; k = k - j
end_switch
   ..rest of code..
```

A *bra* is a 'branch always' – unconditional branch.

Copyright Thomson/Delmar Learning 2005. All Rights Reserved.

Unsigned greater-than Test (>)

In *C*

```
unsigned char i, j;

if (i > j) {
 // done if i greater than j
  j = i + j;
}
// ...rest of code...
```

In Assembly

```
      movf  i,w    ; w = i
      subwf j,w    ; w =  j - i
   ┌─ bc end_if    ; skip if C=1, i <= j
   │  movf  i,w    ; w = i
   │  addwf j,f    ; j =  j + i
   └→ end_if
      ..rest of code..
```

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.

Copyright Thomson/Delmar Learning 2005. All Rights Reserved.

Unsigned greater-than Test (>) Variation

In *C*

```c
unsigned char i, j;

if (i > j) {
 // done if i greater than j
  j = i + j;
}
// ...rest of code...
```

In Assembly

```
    movf  j,w     ; w = j
    subwf i,w     ; w =  i - j
    bz   end_if   ; skip if Z=1, i == j
    bnc end_if    ; skip if C=0, i < j
    movf  i,w     ; w = i
    addwf j,f     ; j =  j + i
end_if
    ..rest of code..
```

In this case, *i-j* is performed instead of *j-i*. This requires checking two flag conditions as the Z flag must be checked in case *i* is equal to *j*. Obviously, the code is more complex because of the need for two branches, so avoid doing the > comparison in this way.

Copyright Thomson/Delmar Learning 2005. All Rights Reserved.

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

In *C*

```c
unsigned char i, j;

if (i >= j) {
 // done if i greater than j
 // done if i equal to j
  j = i + j;
} else {
  // done if i less than j
  j++;
}
// ...rest of code...
```

In Assembly

```
   movf  j,w          ; w = j
   subwf i,w          ; w =  i - j
   bnc else_body      ; skip if C=1, i < j
   movf  i,w          ; w = i
   addwf j,f          ; j =  j + i
   bra   end_if
else_body
   incf  j,f          ; j++
end_if
   ..rest of code..
```

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

Copyright Thomson/Delmar Learning 2005. All Rights Reserved.

Unsigned Comparison Summary

Comparison	Operation	If true, then
if (i > k) {}	k – i	C = 0 (borrow) if i > k
if (i >= k) {}	i – k	C=1 (no borrow) if i>= k
if (i = = k) {}	i – k OR k - i	Z = 1 if i = = k
if (i != k) {}	i – k OR i - k	Z = 0 if i != k

If you do ‘i-k’ for the i>k comparison, then will have to test both C and Z flags. If you do k-i for the i>=k comparison, then will have to test both C and Z flags.

PIC18 Comparison Instructions

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

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

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

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

You can use these to implement 8-bit, unsigned comparisons in fewer instructions than using the subtract/flag test approach.

HOWEVER, the subtract/flag test is a more ***general approach***, and is useful for any comparison type (8-bit unsigned/signed, 16-bit signed/unsigned, etc). The subtract/flag test is emphasized in this class because it can be generally applied; the comparison instructions only work for 8-bit unsigned comparisons.

Comparison example (==) using *cpfseq*

```
unsigned char i,j;
if (i == j) {
    j = i + j;
 }
```

C code

subtract/Z flag test

```
 movf   j,w      ; w ← j
 subwf i,w       ; w ← i - j
 bnz   end_if   ;skip if Z=0
 movf   i,w      ; w ← i
 addwf j,f       ; j ← j + w
end_if
  ..do stuff..
```

using *cmpfseq*

```
   movf   i,w   ; w ← i
   cpfseq j    ; j == i?
   bra   end_if ; i != j
   addwf j,f   ; j ← j + i
end_if
    ..do stuff..
```

did j==i so that w already had i in it to reduce instr. count

The *cmpfseq* approach takes fewer instructions. Use this approach if you want to, but you will still have to understand the subtract/flag test approach for other comparison types later.

Comparison example (>) using *cpfsgt*

C code

```
unsigned char i,j;
if (i > j) {
    j = i + j;
 }
```

subtract/C flag test

```
  movf  i,w     ;w←i
  subwf j,w     ; w←j-i
  bc   end_if   ; skip if C=1
  movf  i,w   ; w←i
  addwf j,f   ; j←j + w
end_if

  ..do stuff..
```

using *cpfsgt*

```
  movf  j,w   ; w ← j
  cpfsgt i   ; i > j?
  bra  end_if ; i <= j
  movf  i,w   ; w←i
  addwf j,f   ; j ← j + i
end_if
  ..do stuff..
```

No advantage – the subtract/flag test is more general and you will need to understand it as it is used for other comparison types later.

Unsigned Literal Comparison

In *C*

```c
unsigned char i,j;

if (i >  0x40) {
  i = i + j;
}
// ...rest of code...
```

In Assembly
Using `cpfsgt`

```
  movlw   0x40 ; w = 0x40
  cpfsgt i     ; i > 0x40?
  bra end_if   ; no,skip
  movf   j,w   ; w = j
  addwf i,f    ; i =  i + j
end_if
  ..rest of code..
```

In Assembly
Using `sublw`

```
  movf    i,w   ; w = i
  sublw   0x40  ; w = 0x040 - i
  bc   end_if   ; skip if 0x40>=i
  movf   j,w    ; w = j
  addwf i,f     ; i =  i + j
end_if
  ..rest of code..
```

Careful! The *sublw* instruction performs *literal* – *w*, not *w* – *literal*.

This is useful for comparing *w* to a *literal*.

Copyright Thomson/Delmar Learning 2005. All Rights Reserved.

while loop

In *C*

```
unsigned char i, j;

while (i > j) {
 // done while i greater than j
  j = i + j;
}
// ...rest of code...
```

In Assembly

```
loop_top
  movf   i,w          ; w = i
  subwf  j,w          ; w =  j - i
  bc end_while        ; skip if C=1, i <= j
  movf   i,w          ; w = i
  addwf  j,f          ; j =  j + i
  bra loop_top
end_while
  ..rest of code..
```

Observe that at the end of the loop, there is a jump back to *loop_top* after body is performed. The body of a *while* loop may not execute if loop test is initially false.

Copyright Thomson/Delmar Learning 2005. All Rights Reserved.

do-while loop

In *C*

```
unsigned char i, j;

do {
 // done while i greater than j
  j = i + j;
} while (i > j);
// ...rest of code...
```

In Assembly

```
loop_top
  movf  i,w          ; w =  i
  addwf j,f          ; j =  j + i
  movf  i,w          ; w =  i
  subwf j,w          ; w =  j - i
  bnc loop_top       ; loop if C=0, i > j
..rest of code..
```

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

Copyright Thomson/Delmar Learning 2005. All Rights Reserved.

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..*/
```

```
unsigned char i,j;

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

executed once, before test.

executed each loop iteration after body

loop test

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

Decrement/Increment, skip if 0, !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

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

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

infsnz *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

In *C*

```
unsigned char i, j, k;

i = 10;
do {
  k =  k + j;
  i--;
} while (i != 0);
// ...rest of code...
```

In Assembly

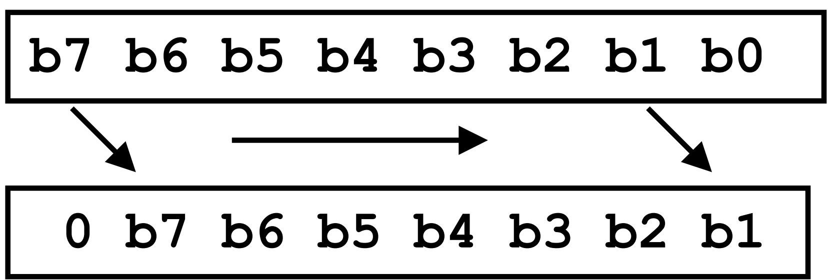
```
  movlw   0x0A      ; w = 10
  movwf   i         ; i = 10
loop_top
  movf   j,w        ; w = j
  addwf k,f         ; k =  k + j
  decfsz i,f        ; i--, skip if zero
  bra loop_top      ; loop if i non-zero
 ..rest of code..
```

Can be used for an end-of-loop action and test in some cases. Usage of incfsz/icfsnz/decfsz/dcfsnz is optional as other instruction sequences can accomplish the same thing. In this case the 'decfsz/bra' can be replaced by 'decf/bnz' combination that is just as efficient.

Copyright Thomson/Delmar Learning 2005. All Rights Reserved.

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)

---

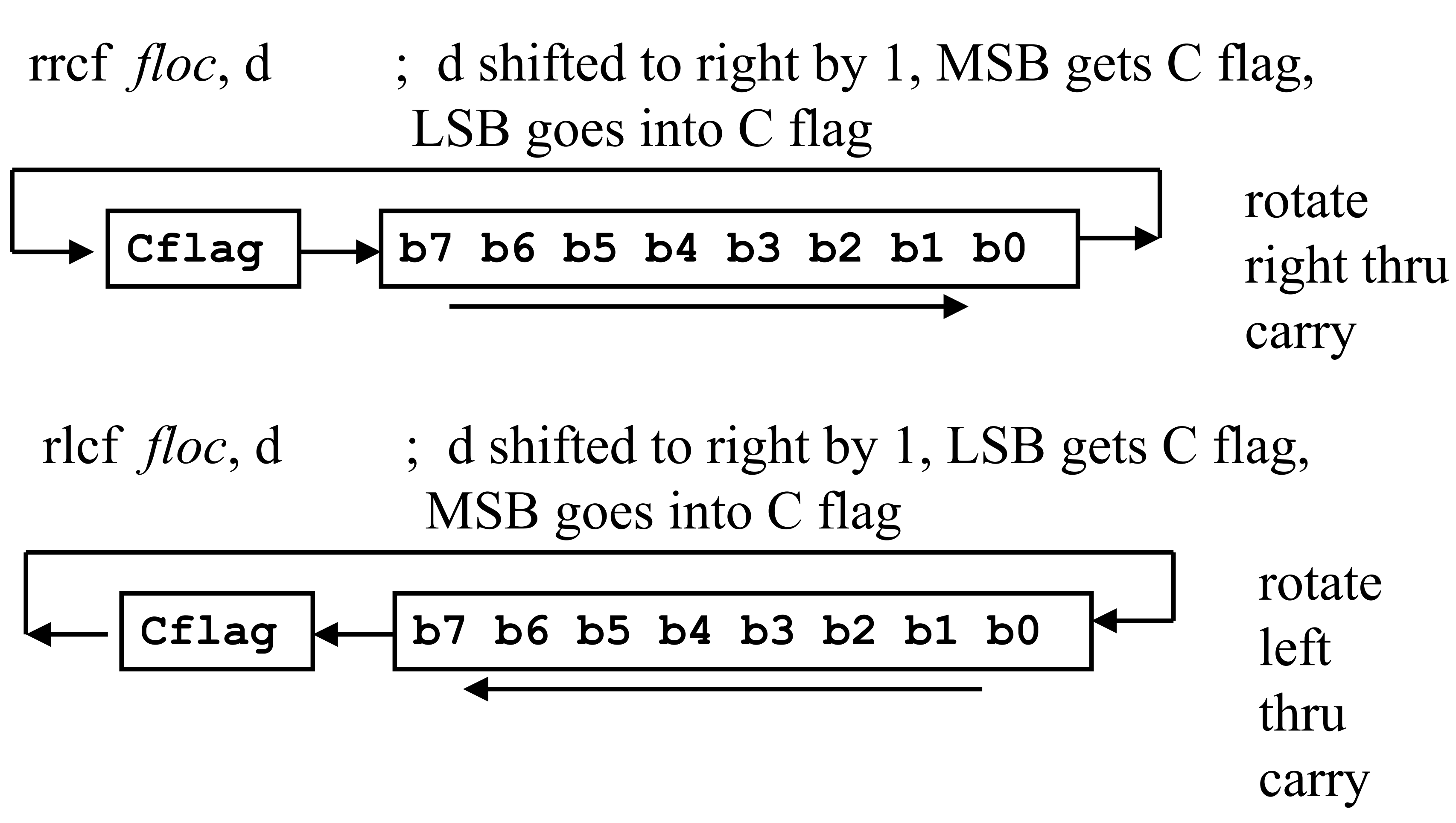
unsigned Shift left i << 1
all bits shift to left by one, ‘0’ into LSB.

`b7 b6 b5 b4 b3 b2 b1 b0` original value

`b6 b5 b4 b3 b2 b1 b0 0` i << 1 (left shift by one)

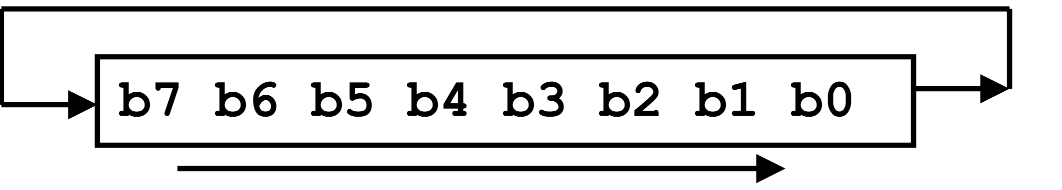
PIC18 Rotate Instructions

PIC18 has *rotate right through carry, rotate right, rotate left through carry, rotate left* instructions



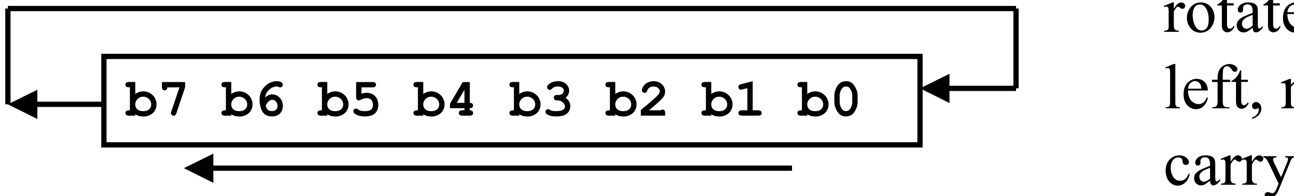
PIC18 Rotate Instructions (cont)

rrncf *floc*, d ; d rotated to right by 1



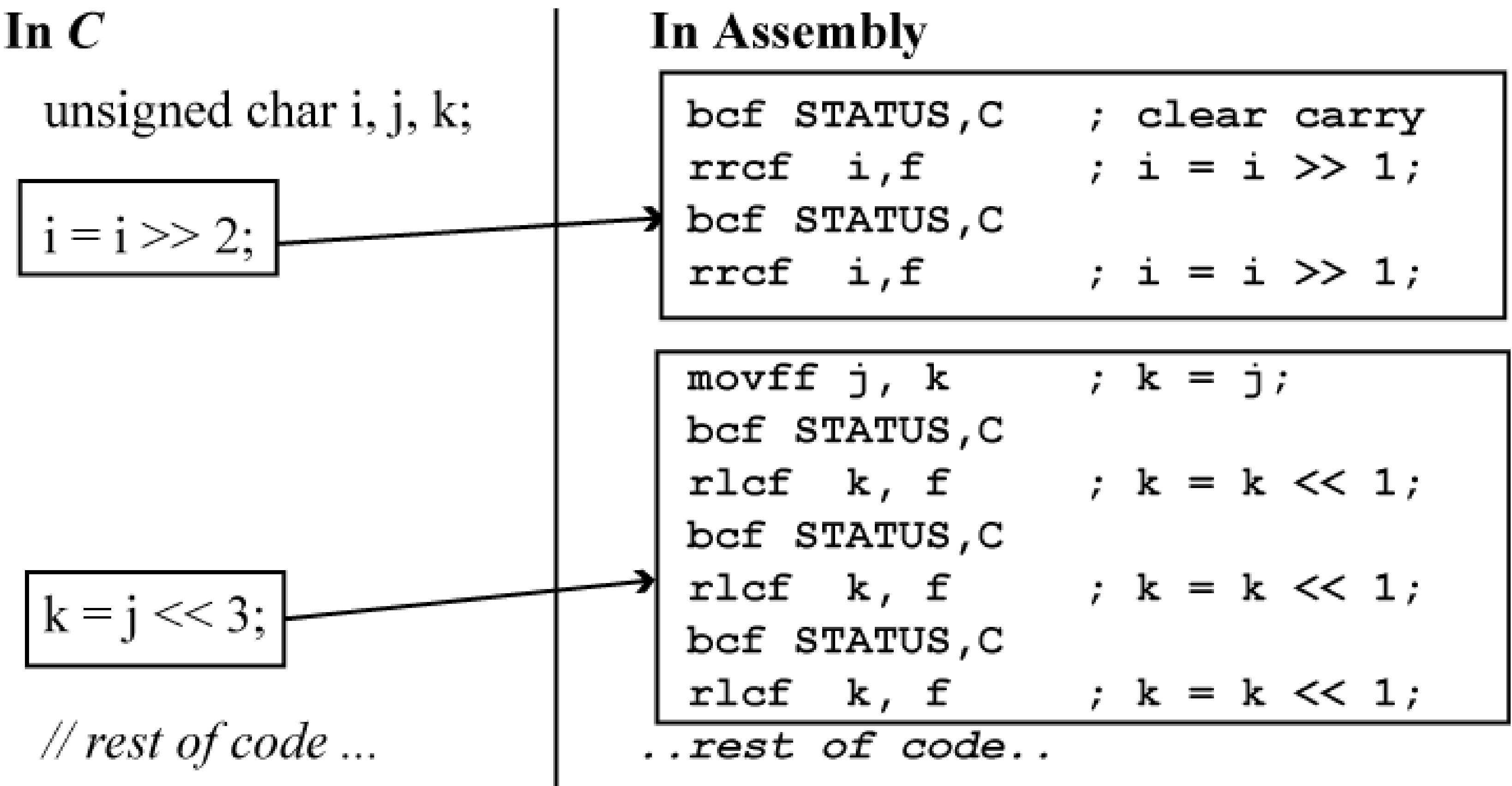
rotate right, no carry

rlncf *floc*, d ; d rotated to left by 1



rotate left, no carry

C Shift operations using Rotates thru Carry



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

Copyright Thomson/Delmar Learning 2005. All Rights Reserved.

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.

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

```
movf   i,w
movwf  org_i    ; save original i
bcf    STATUS,C     ; clear C
rlcf    i,f         ; i << 1
bcf    STATUS,C     ; clear C
rlcf    i,f         ; i << 1
bcf    STATUS,C     ; clear C
rlcf    i,f         ; i << 1
movf   org_i,w     ; w ← org_i
subwf  i,f         ; i ← i -w
;; final i is old_i * 7
```

What instructions do you use?

The PIC18 instruction set has gotten cluttered with instructions as new generations of PICmicro® microcontrollers have been produced. New instructions were added, and old instructions were kept in an effort to keep assembly source compatibility (so that compilers could be ported more easily).

There are many ways for accomplishing the same thing using different instructions sequences.

Which method do you use?

The method that you **understand**......(and have not MEMORIZED).

I will never take off points for 'inefficient code'.

I will always take off points for incorrect code – "close" does not count.

What do you need to know?

- Logical operations (and,or,xor, complement)
- Clearing/Setting/Complementing groups of bits
- Bit set/clear/test instructions
- ==, !=, >, <, >=, <= tests on 8-bit unsigned variables
- Loop structures
- Shift left (>>), Shift Right (<<) using rotate instructions
- Multiplication by a constant via shifts/adds/subtracts