C Arithmetic operators

Operator	Description
+, -	(+) addition, (–) subtraction
++, --	(++) increment, (– –) decrement
*, /	(*) multiplication, (/) division
>>, <<	right shift (>>), left shift (<<)
&, |, ^	bitwise AND (&), OR (|), XOR (^)
~	bitwise complement

The above are *C* operators that we would like to implement in PIC24 assembly language. Multiplication and division will be covered in a later lecture.

Bit-wise Logical operations

Bitwise AND operation

AND.{B}	Wb,Ws,Wd	(Wb)&(Ws)→Wd	j = k & i;
AND.{B}	f	(f)&(WREG) →f	j = j & k;
AND.{B}	f, WREG	(f)&(WREG) →WREG	j = j & k;
AND.{B}	#lit10,Wn	lit10 & (Wn) →Wn	j = j & literal;

Bitwise Inclusive OR operation

IOR.{B}	Wb,Ws,Wd	(Wb) | (Ws)→Wd	j = k | i;
IOR.{B}	f	(f) | (WREG) →f	j = j | k;
IOR.{B}	f, WREG	(f) | (WREG) →WREG	j = j | k;
IOR.{B}	#lit10,Wn	lit10 | (Wn) →Wn	j = j | literal;

Bit-wise Logical operations (cont.)

Bitwise XOR operation

XOR.{B}	Wb,Ws,Wd	(Wb) ^ (Ws)→Wd	j = k ^ i;
XOR.{B}	f	(f) ^ (WREG) →f	j = j ^ k;
XOR.{B}	f, WREG	(f) ^ (WREG) →WREG	j = j ^ k;
XOR.{B}	#lit10,Wn	lit10 ^ (Wn) →Wn	j = j ^ literal;

Bitwise complement operation

COM.{B}	Ws,Wd	~ (Ws)→Wd	j = ~k;
COM.{B}	f	~(f) →f	j = ~j ;
COM.{B}	f, WREG	~(f) →WREG	j = ~k;

Bit-wise Logical operations (cont.)

Clear ALL bits:

CLR.{B}	f	0 →f	j=0;
CLR.{B}	WREG	0 →WREG	j=0;
CLR.{B}	Wd	0 →Wd	j=0;

Set ALL Bits:

SETM.{B}	f	111…1111 →f
SETM.{B}	WREG	111…1111 →WREG
SETM.B}	Wd	111…1111 →Wd

Clearing a group of bits

Data Memory

Location	contents
(i) 0x0800	0x2C
(j) 0x0801	0xB2
(k) 0x0802	0x8A

Clear upper four bits of i .

In *C*:

```
uint8 i;
i = i & 0x0F;
```

← The 'mask' (0x0F)

In PIC24 μC assembly

```
mov.b    #0x0F, W0   ; W0 = mask
and.b    i           ; i = i & 0x0f
```

```
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)	0x0800	0x2C
(j)	0x0801	0xB2
(k)	0x0802	0x8A

Set bits b3:b1 of j

In *C*:

```
uint8 j;
j = j | 0x0E;
```

← The 'mask'

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

In PIC24 μC assembly

```
mov.b    #0x0E, W0     ; W0 = mask
ior.b    j             ; j = j | 0x0E
```

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) 0x0800	0x2C
(j) 0x0801	0xB2
(k) 0x0802	0x8A

Complement bits b7:b6 of k

In *C*:

```
uint8 k;
k = k ^ 0xC0;
```

← The 'mask' (0xC0)

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

In PIC24 µC assembly

```
mov.b    #0xC0, W0    ; W0 = mask
xor.b     k           ; k = k ^ 0xC0
```

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*:

```
uint8 k;
k = ~k ;
```

In PIC24 μC assembly

```
com.b  k        ; k = ~k
```

Data Memory

Location	contents
(i) 0x0800	0x2C
(j) 0x0801	0xB2
(k) 0x0802	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 multiple instructions as previously seen.

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

Name	Mnemonic	Operation
Bit Set	`bset{.b}` *`Ws, #bit4`* *`Ws`* `indirect modes` `bset{.b}` *`f,`* `#bit4`	1 → *Ws<bit4>* 1 → *f<bit4>*
Bit Clear	`bclr{.b}` *`Ws, #bit4`* *`Ws`* `indirect modes` `bclr{.b}` *`f,`* `#bit4`	0 → *Ws<bit4>* 0 → *f<bit4>*
Bit Toggle	`btg{.b}` *`Ws, #bit4`* *`Ws`* `indirect modes` `btg{.b}` *`f, #bit4`*	*~Ws<bit4>* → *Ws<bit4>* *~f<bit4>* → *f<bit4>*

Bit clear/set/toggle examples

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

In *C*:

```
uint8  i, j, k;
  k = k & 0x7F;
  j  = j  |  0x04;
  i  = i  ^  0x20;
```

In PIC24 µC assembly

```
bclr.b  k, #7
bset.b  j, #2
btg.b   i, #5
```

Data Memory

Location	contents
(i) 0x0800	0x2C
(j) 0x0801	0xB2
(k) 0x0802	0x8A

```
                bbbb bbbb
                7654 3210

k =   0x8A  =  1000 1010
bclr.b k,#7
k =   0x0A  =  0000 1010

j =   0xB2  =  1011 0010
bset.b j,#2
j =   0xB6  =  1011 0110

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

status Register

The ***STATUS*** register is a special purpose register (like the Wn registers).

Status Register

–	–	–	–	–	–	–	DC	IPL2	IPL1	IPL0	RA	N	OV	Z	C

|← Status Register high byte →|← Status Register low byte →|

C	Carry
Z	Zero
OV	Overflow
N	Negative
RA	Repeat Loop Active
IPL[2:0]	Interrupt Priority Level
DC	Decimal Carry
–	Unimplemented

The C, Z, OV, N, DC flags can be user set/cleared; also are set/cleared as a side effect of instruction exection.

The RA bit is read-only; set when a `repeat` instruction is active, cleared when `repeat` is finished.

The IPL[2:0] bits are user set/cleared.

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 1 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 *bset*/*bclr* instructions.

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

Look at Table 19-2 in PIC24HJ32GP202 μC datasheeet.– *add* affects all ALU flags, *mov f* only Z, N flags, and *mov f, Wn* no flags.

Mnemonic	Syntax.	Desc	# of words	Instr Cycles	Status affected
ADD	ADD f	f=f+WREG	1	1	C,DC,Z,OV,N
MOV	MOV f,Wn	Wn=(f)	1	1	none
MOV	MOV f	f = (f)	1	1	N,Z

Addition: Carry, Zero Flags

Zero flag is set if result is zero and cleared otherwise.

In addition, carry flag is **set** if there is a carry out of the MSbit and cleared otherwise.

In byte (8-bit) mode, C=1 if sum > 255 (0xFF)
In word (16-bit) mode, C=1 if sum > 65535 (0xFFFF)

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

Byte mode operations are shown.

Subtraction: Carry, Zero Flags

Zero flag is set if result is zero and cleared otherwise.

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

```
  0xF0
- 0x20
 --------
  0xD0    Z=0,
          C=1
```

```
  0x00
 -0x00
 --------
  0x00    Z=1,
          C=1
```

```
  0x01
 -0xFF
 --------
  0x02    Z=0,
          C=0
```

For a subtraction, the combination of Z=1, C=0 will not occur. Byte mode operations are shown.

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

```
  0xF0
- 0x20
--------
  0xD0    Z=0,
          C=1
```

No borrow, C=1

```
 0x20 = 0010 0000
~0x20 = 1101 1111
      = 0xDF
```

```
  0xF0
+ 0xDF
+ 0x01
--------
  0xD0    Z=0,
          C=1
```

Carry out of MSB, so C=1

C Shift Left, Shift Right

logical Shift right i >> 1

all bits shift to right by one, ‘0’ into MSB (8-bit right shift shown)

b7 b6 b5 b4 b3 b2 b1 b0	original value
0 b7 b6 b5 b4 b3 b2 b1	i >> 1 (right shift by one)

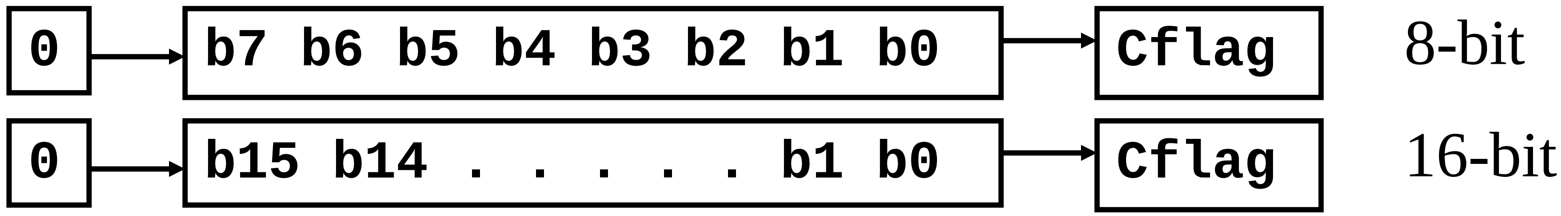
Shift left i << 1

all bits shift to left by one, ‘0’ into LSB (8-bit left shift shown)

b7 b6 b5 b4 b3 b2 b1 b0	original value
b6 b5 b4 b3 b2 b1 b0 0	i << 1 (left shift by one)

PIC24 Family Unsigned Right Shifts

Logical Shift Right

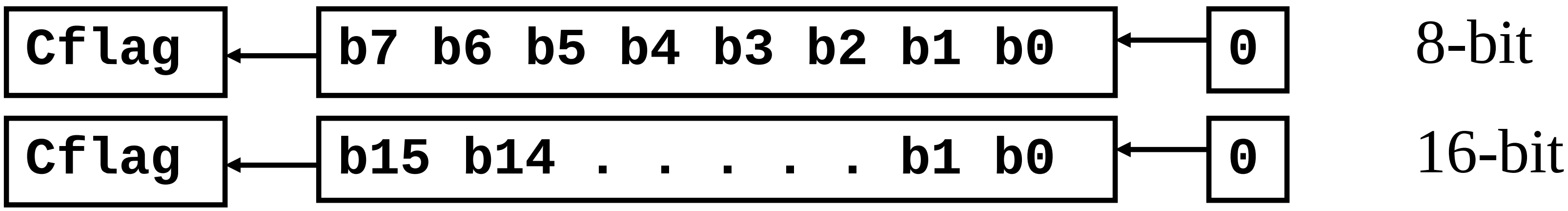


Descr:	Syntax	Operation
Log. Shift Right f	LSR{.B} f	f >> 1 → f
	LSR{.B} f,WREG	f >> 1 → WREG
Log. Shift Right Ws	LSR{.B} Ws,Wd	Ws>> 1 → Wd
Log. Shift Right by short Literal	LSR Wb, #lit4, Wd	Wb>>lit4 → Wd
Log. Shift Right by Ws	LSR Wb, Ws, Wd	Wb>>Ws → Wd

The last two logical shifts can shift multiple positions in one instruction cycle (up to 15 positions), but only as word operations. There is an ***arithmetic*** right shift that will be covered in a later lecture.

PIC24 Family Left Shifts

Shift left

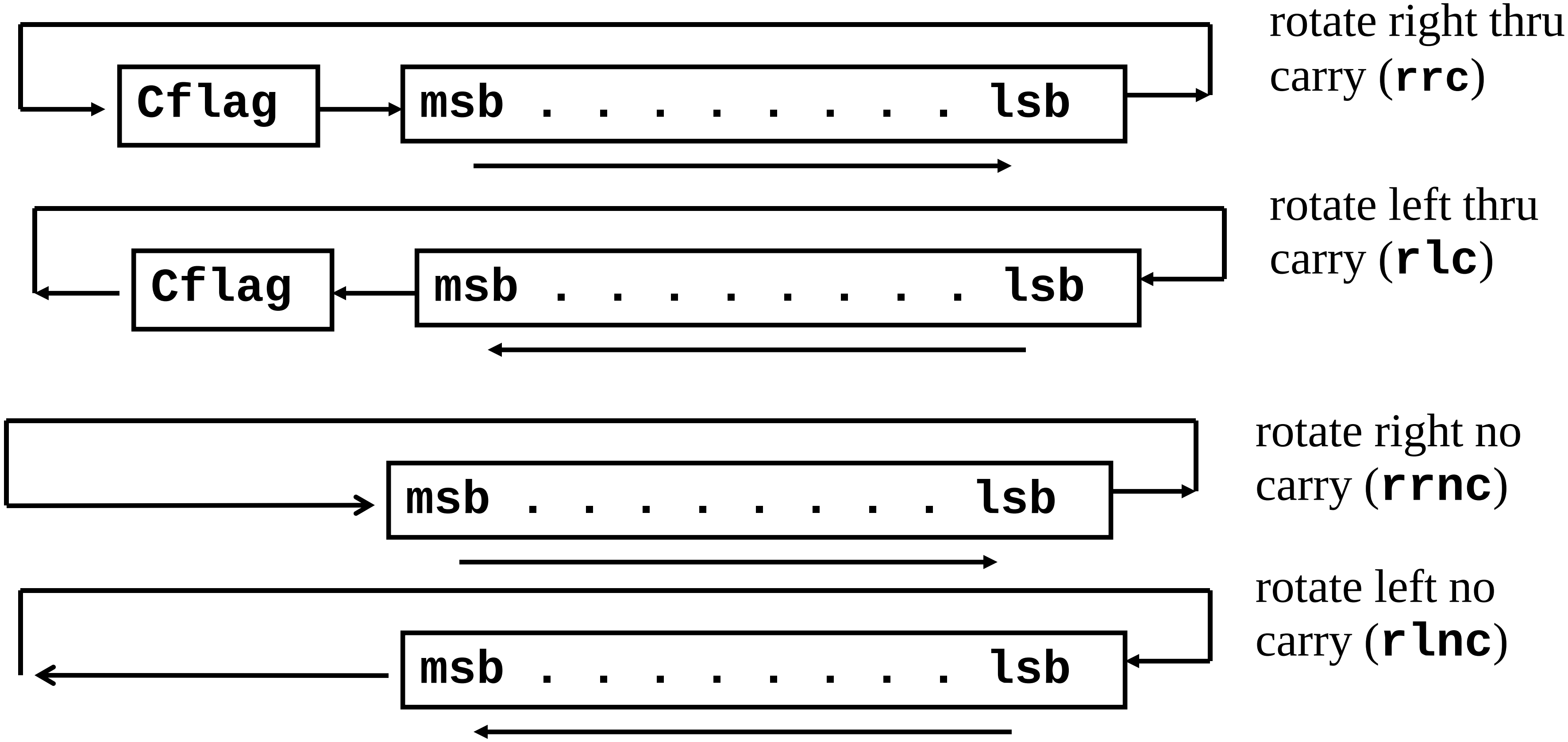


Descr:	Syntax	Operation
Shift left f	SL{.B} f	f << 1 → f
	SL{.B} f,WREG	f << 1 → WREG
Shift left Ws	SL{.B} Ws,Wd	Ws << 1 → Wd
Shift left by short Literal	SL Wb, #lit4, Wd	Wb << lit4 → Wd
Shift left by Ws	SL Wb, Ws, Wd	Wb << Ws → Wd

The last two shifts can shift multiple positions in one instruction cycle (up to 15 positions), but only as word operations.

PIC24 Rotate Instructions

PIC24 has some rotate left and rotate right instructions as well:



The **`rrc`**/**`rlc`** instructions are used in the next chapter for 32-bit shift operations. The **`rrnc`**/**`rlnc`** are not discussed further. The valid addressing modes are the same as for the shift operations that only shift by one position.

C Shift operations

In *C*

```c
uint16  u16_p, u16_k;
uint8   u8_i;

u8_i = u8_i >> 2;

u16_k = u16_p << 10;
```

In Assembly

```
lsr.b u8_i          ;i = i >> 1;
lsr.b u8_i          ;i = i >> 1;
```

```
mov u16_p,w1        ;w1 = p
sl  w1,#10,w1       ;w1 = w1 << 10
mov w1,u16_k        ;k = w1
```

It is sometimes more efficient to repeat a single position shift instruction performing a multi-bit shift.

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

Arithmetic Example

(a) In *C*

```
uint16  i,n,p;

k = n + (i<<3) - p;
```

(b) Steps:

Copy *n*, *i* to working registers
Perform *i* << 3
Add to *n*
Subtract *p*
Write to *k*

(c) In Assembly

```
mov   n,W0          ;W0 = n
mov   i,W1          ;W1 = i
sl    W1,#3,W1      ;W1 = i << 3;
add   W0,W1,W0      ;W0 = n + (i<<3)
mov   p,W1          ;W1 = p
sub   W0,W1,W0      ;W0 = (n + (i<<3))-p
mov   W0,k          ;k  = (n + (i<<3))-p
```

Use working registers for storage of intermediate results.

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

Mixed 8-bit, 16-bit operations

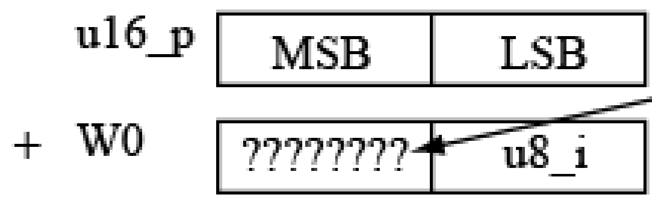
In *C*

```
uint16  u16_p;
uint8   u8_i;

u16_p= u16_p + u8_i;
```

(a) In Assembly (incorrect)

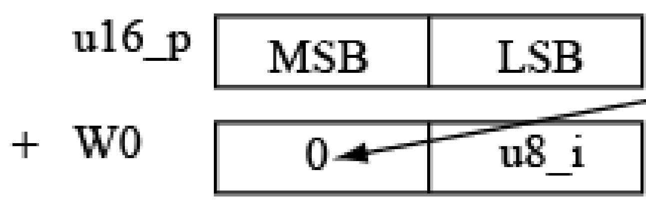
```
mov.b u8_i,WREG    ;W0.LSB = u8_i
add   u16_p        ;u16_p = u16_p + W0
```



The upper 8 bits of W0 are unknown; 16-bit sum is likely incorrect.

(b) In Assembly (correct)

```
mov.b u8_i,WREG    ;W0.lsb = u8_i
ze    W0,W0        ;Zero extend W0
add   u16_p        ;u16_p = u16_p + W0
```



The upper 8 bits of W0 are now zero; unsigned 8-bit variables should be zero-extended before use in 16-bit operations.

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

Conditional Execution using Bit Test

The ‘bit test f, skip if clear’ (btsc) and ‘bit test f, skip if set’ (btss) instructions are used for conditional execution.

btsc{.b} f, #bit4   ; skips next instruction is f<#bit4> is clear (‘0’)

btss{.b} f, #bit4   ; skips next instruction is f<#bit4> is set (‘1’)

Bit test instructions are just the first of many different methods of performing conditional execution in the PIC24 μC.

Number Sequencing Task using *btsc*

```
          .bss          ;unitialized data section
loc:      .space 1      ;byte variable
out:      .space 1      ;byte variable
         .text          ;Start of Code section
__reset:                ; first instruction
    mov #__SP_init, W15  ;Initalize the Stack Pointer
   ;bclr   loc, #0      ;uncomment for loc<0>=0
   bset    loc, #0      ;uncomment for loc<0>=1
loop_top:
   btsc.b  loc,#0       ;skip next if loc<0> is 0
   goto    loc_lsb_is_1
   ;loc<0> is 0 if reach here
   mov.b   #3,w0
   mov.b   wreg,out     ;out = 3
   mov.b   #2,w0
   mov.b   wreg,out     ;out = 2
   mov.b   #4,w0
   mov.b   wreg,out     ;out = 4
loc_lsb_is_1:
   mov.b   #8,w0
   mov.b   wreg,out     ;out = 8
   mov.b   #5,w0
   mov.b   wreg,out     ;out = 5
   mov.b   #6,w0
   mov.b   wreg,out     ;out = 6
   mov.b   #1,w0
   mov.b   wreg,out     ;out = 1
   goto    loop_top     ;loop forever
```

Skip **`goto loc_lsb_is_1`** if least significant bit of **`loc`** is 0.

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

C Conditional Tests

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

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 = 0xA0`**

`!(i)` ⇨ **`0`** **`~(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

```
uint8 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
```

```
uint8 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
```

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
```

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.

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 defined in *p24Hxxxx.inc;* use for clarity!!!!

In *C*

```
uint16  k;

if (k) {
 if-body
}
... rest of code
```

In Assembly

```
  mov   k              ; k = k, affects N,Z flags
  btsc  SR,#1          ; skip if Z = 0 (Z is SR<1>)
  goto  end_if         ; Z = 1, k is 0
  if-body stmt1
      ... stmtN
end_if:
  ... rest of code
```

The *mov i* instruction just moves *i* back onto itself! Does no useful work except to affect the Z, N flags.

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

Conditional Execution using branches

A *branch* functions as a conditional *goto* based upon the setting of one more flags

Simple branches test only one flag:

BRA	Z, <label>	branch to label if Z=1
BRA	NZ, <label>	branch to label if Z=0 (not zero)
BRA	C, <label>	branch to label if C=1
BRA	NC, <label>	branch to label if C=0 (no carry)
BRA	N, <label>	branch to label if N=1
BRA	NN, <label>	branch to label if N=0 (not negative)

BRA <label> unconditional branch to <label>

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

Non-Zero Test

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

In *C*

```
uint16 i, j;

if (i) {
 // if-body code
}
// ...rest of code...
```

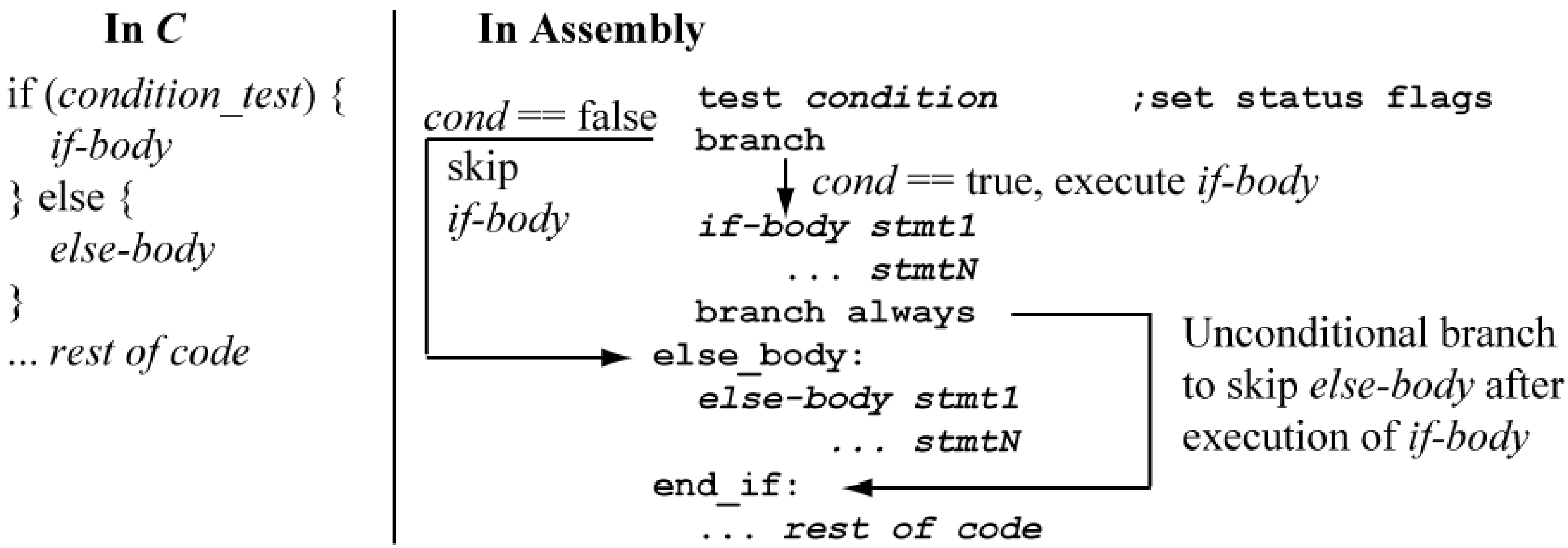
In Assembly

```
  mov   i             ; i = i, affects N,Z flags
  bra   Z,end_if      ; skip if-body when Z=1 (i is 0)
  ..if-body stmt1
  ..if-body stmtN
end_if:
  ..rest of code..
```

For a non-zero test *if(!i){}* replace *bra Z* with *bra NZ*

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

General if-else form with branches



Choose the branch instruction such that the branch is **TAKEN** when the condition is **FALSE**.

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

Equality Test (==)

In *C*

```
uint16  k, j;

if (k == j) {
  if-body
}
... rest of code
```

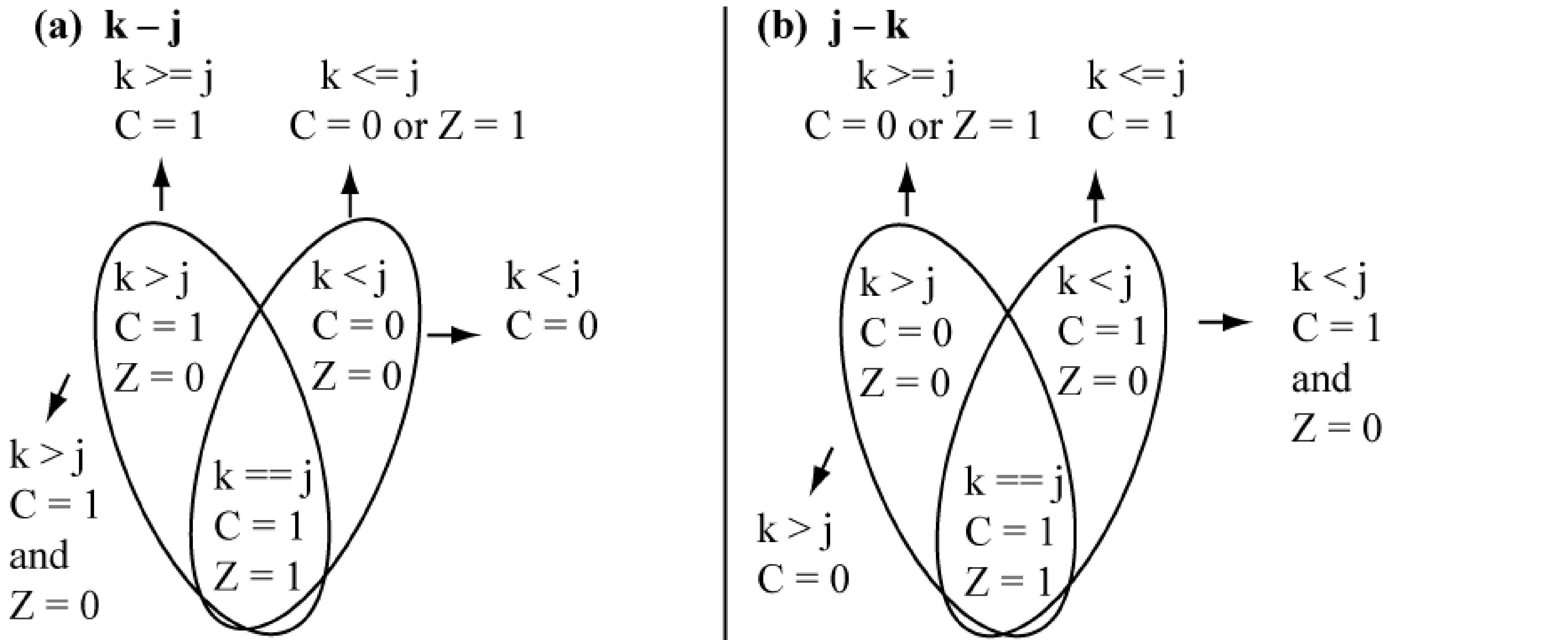
In Assembly

```
    mov   j,W0          ;W0 = j
    sub   k,WREG        ;W0 = k - j
  ┌─bra   NZ,end_if     ;skip if-body when Z=0 (k != j)
  │ if-body stmt1
  │     ... stmtN
  └►end_if:
    ... rest of code
```

Subtraction operation of k-j performed to check equality;

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

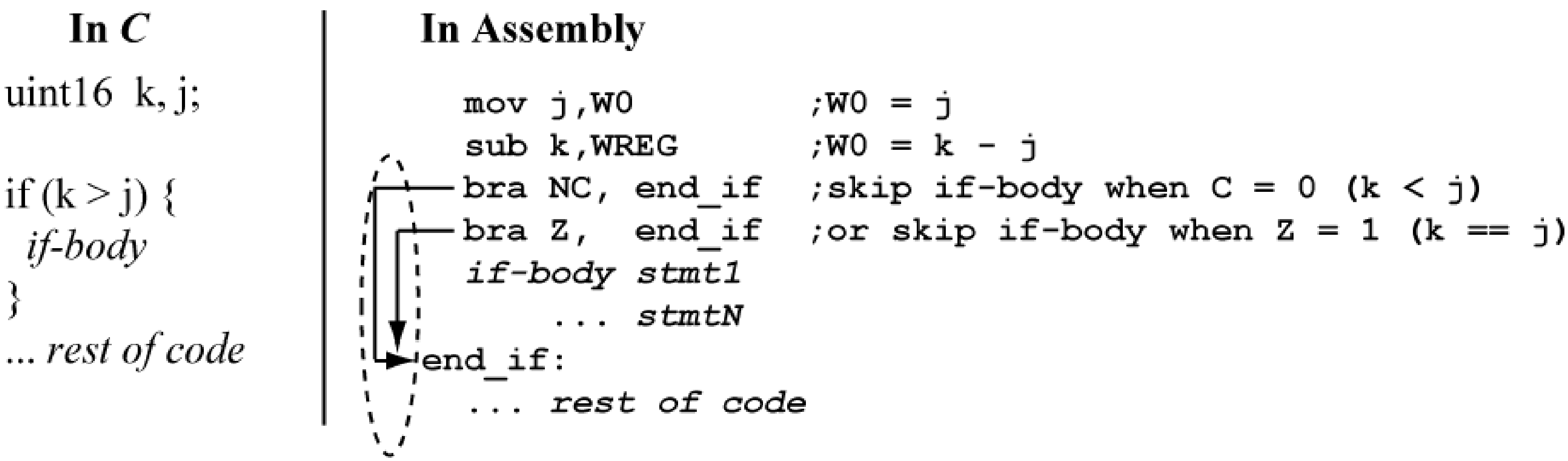
>,>=, <,<= tests using Z, C flags and subtraction



Note: k <= j is ~(k > j) is ~(C & ~Z) is (~C | Z) by DeMorgan's law. Similarly, k < j is ~(k >= j) is ~(C) is ~C.

Note: k < j is ~(k >= j) is ~(!C | Z) is (C & ~Z) by DeMorgan's law. Similarly, k <= j is ~(k > j) is ~(~C) is C.

k>j test using k-j



False condition of k > j is k <= j, so need branches that accomplish this.

The false condition of k>j is k<=j, so use k<=j to skip around the if-body. For the k-j test, this is accomplished by C=0 or Z=1, requiring two branches.

k>j test using j-k

In *C*

```
uint16  k, j;

if (k > j) {
 if-body
}
... rest of code
```

In Assembly

```
  mov   k,W0        ;W0 = k
  sub   j,WREG      ;W0 = j - k
  bra   C, end_if   ;skip if-body when C = 1 (k <= j)
  if-body stmt1
      ... stmtN
end_if:
  ... rest of code
```

The false condition of k>j is k<=j, so use k<=j to skip around the if-body. For the j-k test, this is accomplished by C=1, requiring one branch

Comparison, Unsigned Branches

Using subtraction, and simple branches can be confusing, since it can be difficult to remember which preferred subtraction to perform and which branch to use.

Also, the subtraction operation overwrites a register value.

The comparison instruction (CP) performs a subtraction without placing the result in register:

Descr:	Syntax	Operation
Compare f with WREG	CP{.B} f	f – WREG
Compare Wb with Ws	CP {.B} Wb,Ws	Wb – Ws
Compare Wb with #lit5	CP{.B} Wb,#lit5	Wb – #lit5
Compare f with zero	CP0{.B} f	f – 0
Compare Ws with zero	CP0{.B} Ws	Ws – 0

Comparison, Unsigned Branches (cont)

Unsigned branches are used for unsigned comparisons and tests a combination of the Z, C flags, depending on the comparison.

Descr:	Syntax	Branch taken when
Branch >, unsigned	BRA GTU, label	C=1 && Z=0
Branch >=, unsigned	BRA GEU, label	C=1
Branch <, unsigned	BRA LTU, label	C=0
Branch <=, unsigned	BRA LEU, label	C=0 || Z=1

Use a Compare instruction to affect the flags before using an unsigned branch.

Example:

```
CP  W0, W1          ; W0 – W1
BRA GTU, place      ; branch taken if W0 > W1
```

Unsigned Comparison (> test)

In *C*

```
uint16  k, j;

if (k > j) {
 if-body
}
... rest of code
```

In Assembly

```
  mov j,W0           ;W0 = j
  cp  k              ;k - WREG
  bra LEU, end_if    ;skip if-body when k <= j
  if-body stmt1
      ... stmtN
end_if:
  ... rest of code
```

For k > j test, use the LEU (less than or equal unsigned) branch to skip IF body if   k <= j

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

If-else Example

In *C*

```
uint16  k, j;

if (k <= j) {
 // if-body code
} else {
 //else-body code
}
// ...rest of code...
```

In Assembly

```
  mov j,W0               ;W0 = j
  cp  k                  ;k - WREG
  bra GTU, else_body     ;skip if-body when k > j
  ..if-body stmt1
  ..if-body stmtN
  bra  end_if            ;use unconditional branch
else_body:               ;to skip else-body after
  ..else-body stmt1      ;executing if-body
  ..else-body stmtN
end_if:
  ..rest of code..
```

Must use unconditional branch at end of if-body to skip the else-body.

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

Unsigned literal Comparison

(a) In *C*

```
uint16  k;

if (k > 10) {
 // if-body code
}
// ...rest of code...
```

In Assembly — 5-bit literal, unsigned range 0 to 31

```
  mov k,W0             ;W0 = k
  cp W0,#10            ;k - 10
  bra LEU, end_if      ;skip if-body when k <= 10
  ..if-body stmt1
  ..if-body stmtN
end_if:
  ..rest of code..
```

(b) In *C*

```
uint16  k;

if (k > 520) {
 // if-body code
}
// ...rest of code...
```

In Assembly — 16-bit literal, unsigned range 0 to 65535

```
  mov #520,W0          ;W0 = 520
  cp  k                ;k - WREG
  bra LEU, end_if      ;skip if-body when k <= 520
  ..if-body stmt1
  ..if-body stmtN
end_if:
  ..rest of code..
```


From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

switch Statement in *C*

(a) Chained *if-else* structure

```
uint8 u8_i;
uint16 u16_j, u16_k;

if (u8_i == 1) {
   u16_k++;
}
else if (u8_i == 2) {
   u16_j--;
}
else if (u8_i == 3) {
  u16_j = u16_j + u16_k;
}
else {
  u16_k = u16_k - u16_j;
}
```

(b) *switch* structure

```
uint8 u8_i;
uint16 u16_j, u16_k;

switch (u8_i) {
 case 1: u16_k++;
         break;
 case 2: u16_j--;
         break;
 case 3: u16_j = u16_j + u16_k;
         break;
 default: u16_k = u16_k - u16_j;
}
```

`break` is required to keep from executing the next `case` block.

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

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

switch Statement in assembly language

In *C*

```c
uint8 u8_i;
uint16 u16_j, u16_k;

switch (u8_i) {

  case 1: u16_k++;
      break;

  case 2: u16_j--;
      break;

  case 3:
   u16_j = u16_j + u16_k;
   break;

  default:
  u16_k = u16_k - u16_j;

}// end switch
```

In Assembly

```
  mov.b u8_i,WREG          ;W0 = u8_i
  cp.b W0,#1               ;u8_i == 1?
  bra NZ,case_2
  inc   u16_k              ;u16_k++
  bra   end_switch         ;break statement
case_2:
  cp.b W0,#2               ;u8_i == 2?
  bra NZ,case_3
  dec   u16_j              ;u16_j--
  bra   end_switch         ;break statement
case_3:
  cp.b W0,#3               ;u8_i == 3?
  bra NZ, default
  mov   u16_k,W0
  add   u16_j              ;u16_j = u16_j + u16_k
  bra   end_switch         ;break statement
default:
  mov   u16_j,W0
  sub   u16_k              ;u16_k = u16_k - u16_j
end_switch:
  ..rest of code..
```

OK to use W0 for computation after comparison is done.

Note: The literal size in the CP instruction is 5-bits (unsigned values of 0-31).

Unsigned, Zero, Equality Comparison Summary

Condition	Test	True Branch	False Branch
i == 0	i – 0	bra Z	bra NZ
i != 0	i – 0	bra NZ	bra Z
i == k	i – k	bra Z	bra NZ
i != k	i – k	bra NZ	bra Z
i > k	i – k	bra GTU	bra LEU
i >= k	i – k	bra GEU	bra LTU
i < k	i – k	bra LTU	bra GEU
i <= k	i – k	bra LEU	bra GTU

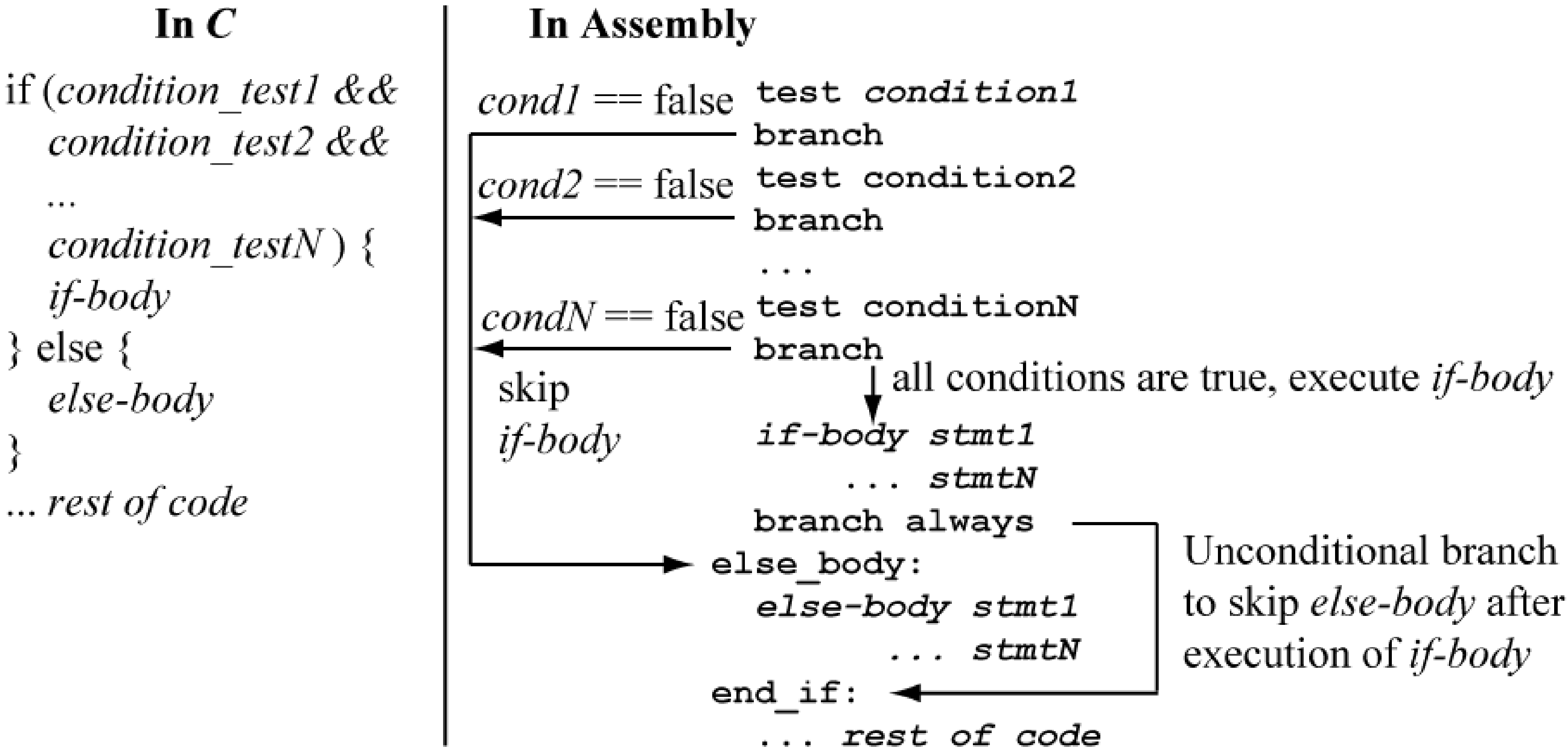
Other PIC24 Comparison Instructions

The PIC24 has various other comparison instructions

```
CPSEQ  Wb,Wn       ; if Wb == Wn, skip next instruction
CPSNE  Wb,Wn       ; if Wb != Wn, skip next instruction
CPSGT  Wb,Wn       ; if Wb == Wn, skip next instruction
CPSLT  Wb,Wn       ; if Wb < Wn, skip next instruction
```

These are provided as upward compatibility with previous PICmicro families, and may save an instruction or two in certain situations. However, we will not use them since their functionality can be duplicated by previously covered compare/branch instructions.

Complex Conditions (&&)



The *else-body* is branched to on the first condition that is false.
The *if-body* is executed if all conditions are true.

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

Complex Condition Example (&&)

In *C*

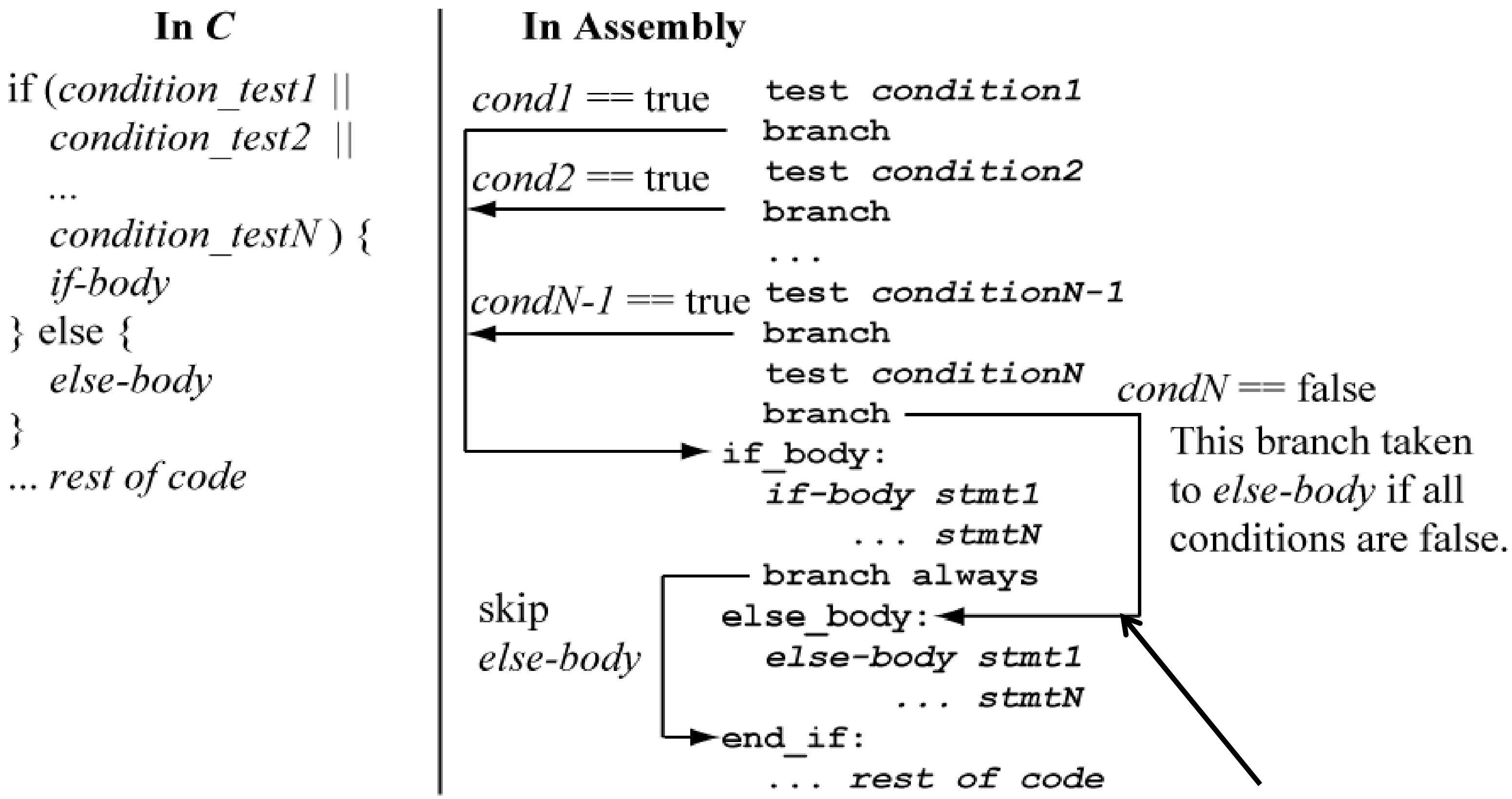
```
uint16 i, j, k;

if ((i < k) &&
   (j != 20)) {
   if-body
} else {
   else-body
}
... rest of code
```

In Assembly

```
  mov k,W0              ;W0 = k
  cp  i                 ;i - WREG
  bra GEU, else_body    ;skip if-body when i >= k
  mov #20,W0            ;W0 = 20
  cp  j                 ;j - WREG
  bra Z, else_body      ;skip if-body when j == 20
  if-body stmt1
      ... stmtN
  bra  end_if           ;skip else-body
else_body:
  else-body stmt1
        ... stmtN
end_if:
  ... rest of code
```

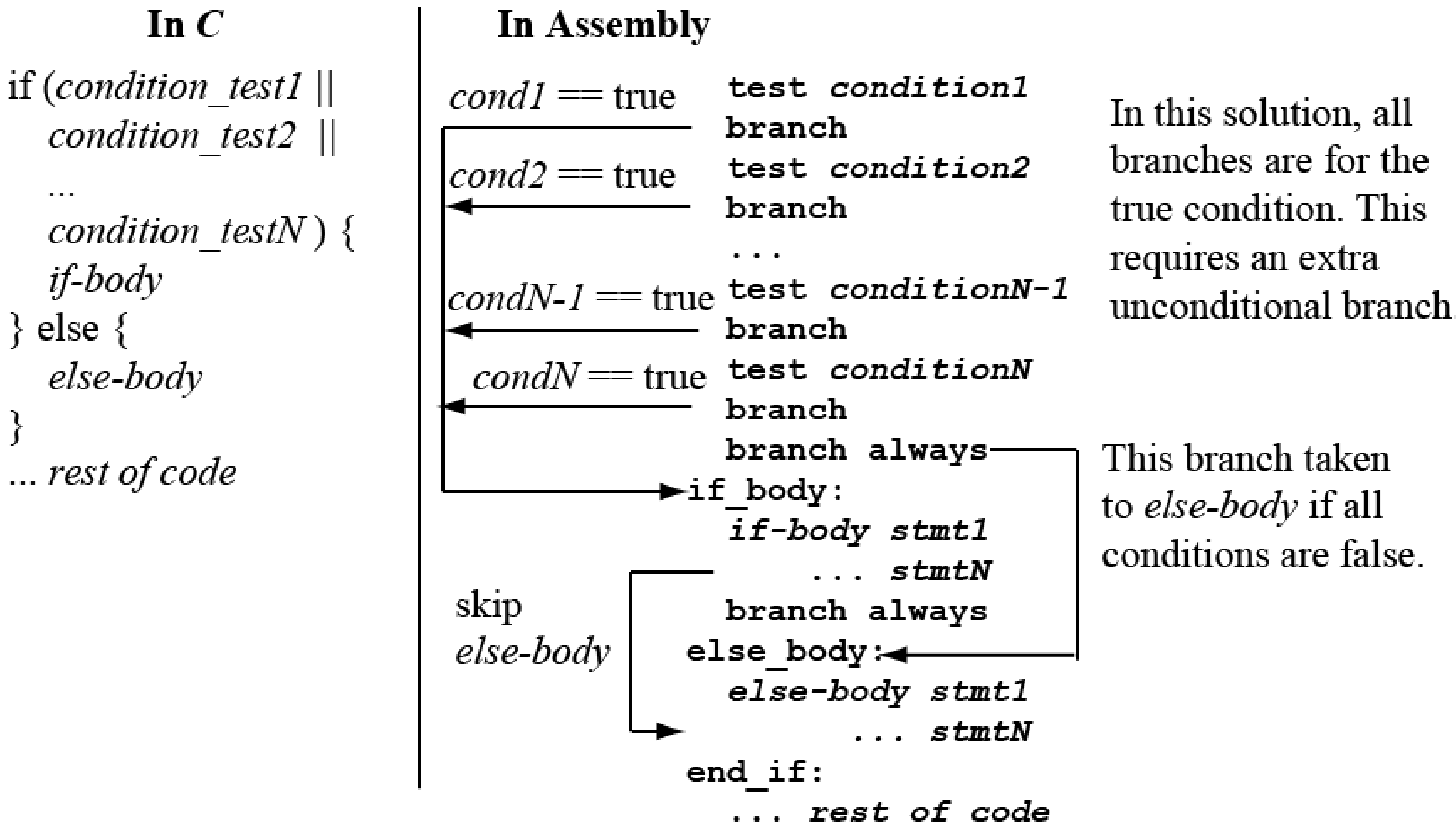
Complex Conditions (||)



The *if-body* is branched to on the first condition that is true.
The *else-body* is executed if all conditions are false.

Careful of last branch!
Different from others!

Complex Conditions (||), alternate method



The *if-body* is branched to on the first condition that is true.
The *else-body* is executed if all conditions are false.

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

Complex Condition Example (||)

In *C*

```
uint16 i, j, k;
uint16 p,q;

if ((i < k) ||
   (j == p) ||
   (q != 0)) {
   if-body
} else {
   else-body
}
... rest of code
```

In Assembly

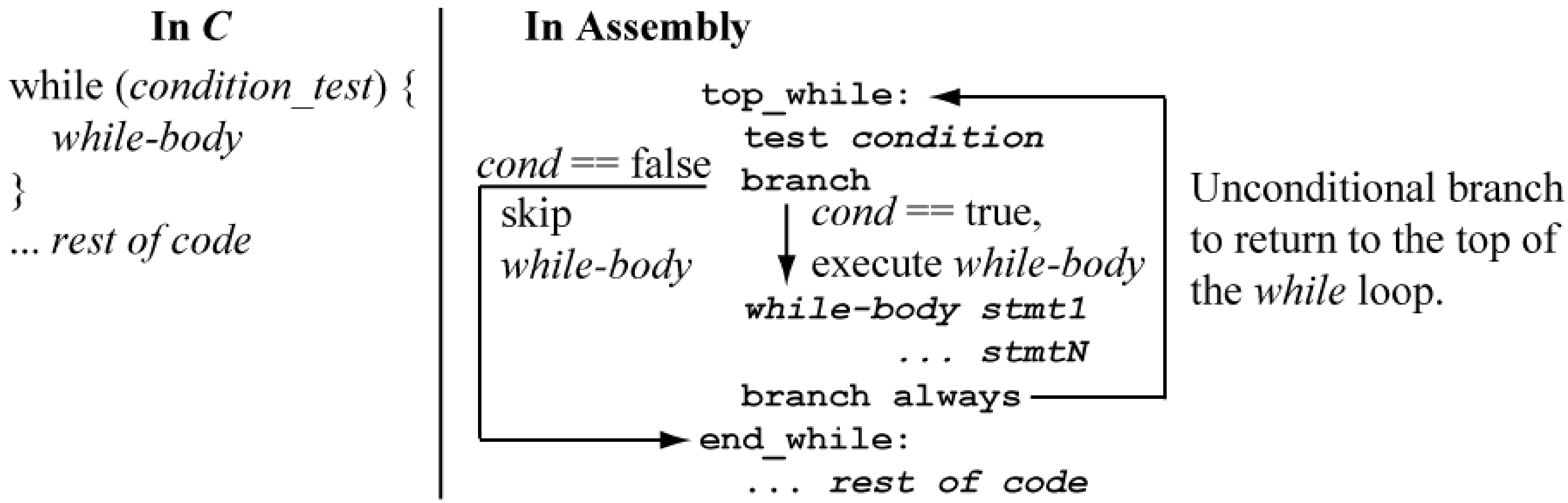
```
   mov k,W0               ;W0 = j
   cp  i                  ;i - WREG
   bra LTU, if_body       ;execute if-body when i < k
   mov p,W0               ;W0 = p
   cp  j                  ;j - WREG
   bra Z, if_body         ;execute if-body when j == p
   cp0 q                  ; q - 0
   bra Z, else_body       ;skip if-body when q == 0
if_body:
   if-body stmt1
       ... stmtN
   bra  end_if
else_body:
   else-body stmt1
       ... stmtN
end_if:
   ... rest of code
```

Can be replaced with:

```
bra NZ, if_body ;true cond!
bra else_body
```


From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

while loop Structure



The *while-body* is not executed if the condition test is initially false.

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

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

while loop Example

In *C*

```
uint16  k, j;

while (k > j) {
  while-body
}
... rest of code
```

In Assembly

```
top_while:
  mov j,W0              ;W0 = j
  cp  k                 ;k - WREG
  bra LEU, end_while
  while-body stmt1
         ... stmtN
  bra top_while
end_while:
  ... rest of code
```

skip *while-body* if k <= j

do-while loop Structure

In *C*

```
do {
   do-while-body
} while (condition_test)
... rest of code
```

In Assembly

```
top_do_while:
  do-while-body stmt1
            ... stmtN
  test condition
  branch
... rest of code
```

cond == true: On true condition, return to the top of the *do-while* loop.

cond == false, exit loop

The *do-while-body* is always executed at least once.

do-while Example

In *C*

```
uint16  k, j;

do {
 while-body
}while (k > j);
... rest of code
```

In Assembly

```
top_do_while:
  while-body stmt1
         ... stmtN
  mov j,W0        ;W0 = j
  cp  k           ;k - WREG
  bra GTU, top_do_while
  ... rest of code
```

return to top of *do{}while* loop if k > j

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

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).

```
uint16 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.

Bit Test Instruction

The ‘bit test’ instruction: **`btst f, #bit4`** is useful for testing a single bit in an operand and branching on that bit. The complement of the bit is copied to the Z flag (if bit is 0, then Z=1; if bit is 1, then Z=0).

In *C*

```
uint16  k, j;

if (k & 0x0080) {
 if-body
}
... rest of code
```

This is testing bit #7 of k

In Assembly

```
   btst k,#7          ;same as k & 0x0080
   bra Z, end_if      ;skip if-body when bit #7 is 0
   if-body stmt1
       ... stmtN
end_if:
   ... rest of code
```

Other forms of ‘bit test’ are available; they will not be discussed.

What instructions do you use?

You will discover that there are many ways for accomplishing the same thing using different instruction sequences.

Which method do you use?

The method that you **understand**......(and have not MEMORIZED), since memorization of code fragments will fail if faced with a situation different from what is memorized.

Your grade will not be penalized for 'inefficient code' in this course since this is your first look at assembly language programming.

Your grade will **always** be penalized for **incorrect** code – "close" does not count.

What do you need to know?

- Bitwise logical operations (and,or,xor, complement)
  – Clearing/setting/complementing groups of bits
- Bit set/clear/toggle instructions
- Shift left (<<), shift right (>>)
- Status register (C, Z flags)
- ==, !=, >, <, >=, <= tests on 8-bit, 16-bit unsigned variables
  – Conditional execution
- Loop structures