

# Extended Precision Operations

To represent larger numbers, more bits are needed.

N bits can represent the unsigned range 0 to  $2^N-1$ .

Bytes 1 Byte = 8 bits	Unsigned Range	C Data Type (PIC18 HITECH C compiler)
1 (8 bits)	0 to 255	char
2 (16 bits)	0 to 65,535	short
2 (16 bits)	0 to 65,535	int
4 (32 bits)	0 to 4,294,967,295	long

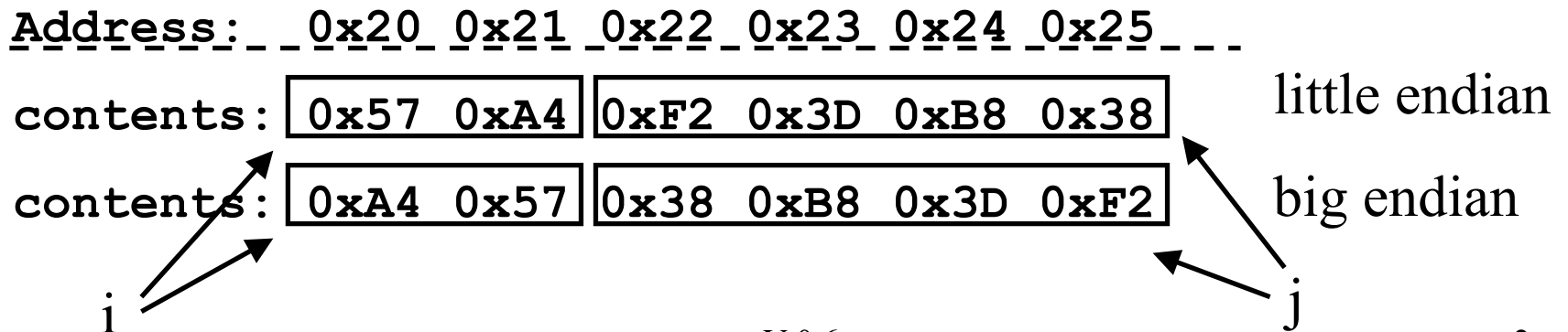
The size of *int*, *long* depends on the C implementation; on some machines both *int* and *long* are 4 bytes, with a *short* being 2 bytes. On some machines a *long* is 8 bytes (64 bits).

# Little Endian vs Big Endian

Byte-ordering for multi-byte integers can be stored  
least significant to most significant (**little endian**)  
or most significant to least significant (**big endian**)

```
int i; long j;  
  
i = 0xA457;  
j = 0x38B83DF2;
```

Assume i @ 0x20, and j @ 0x22



# Which is better?

- No inherent advantage to either byte ordering
- If a processor only contains 8-bit registers, it is the choice of the programmer or compiler writer
  - Be consistent!
- On processors that have 16-bit and 32-bit operations, the  $\mu$ P architects choose the byte ordering
  - Intel  $\mu$ Ps use little endian, Motorola  $\mu$ Ps uses big endian
  - It is a religious argument....
- The PIC18 has some multi-byte registers that are arranged in little-endian order in the file registers, so little endian ordering is used in all examples.

# Multi-byte values in the MPLAB<sup>®</sup> IDE

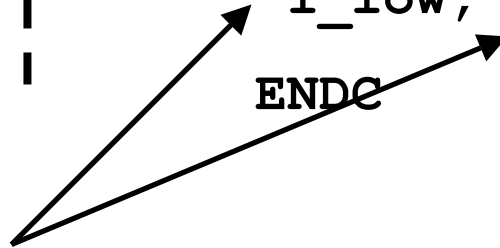
C code

```
int i;  
i = 0xC428;
```



PIC18 assembly

```
CBLOCK 0x040  
    i_low, i_high  
ENDC
```

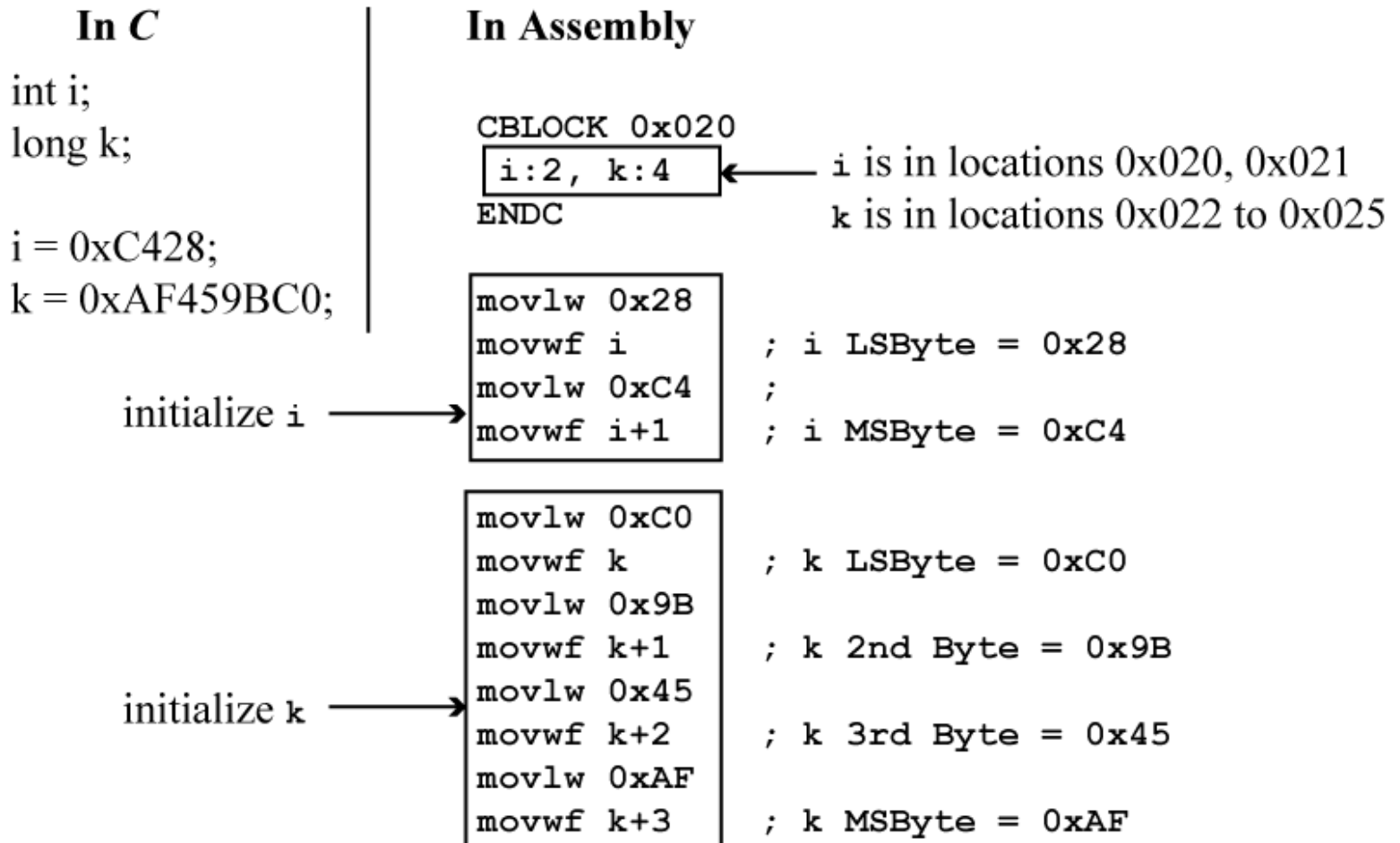


Explicitly named each byte of *i*.

Arranged in little endian order.

```
;; i = 0xC428  
movlw 0x28  
movwf i_low ; LSB = 0x28  
movlw 0xC4  
movwf i_high ;MSB = 0xC4
```

# Multi-byte values in the MPLAB<sup>®</sup> IDE (cont).



This method is preferred; i+1 refers to the memory address computed as  $0x020 + 1 = 0x021$ , which contains the MSB of i.

# 16-bit Addition using Carry Flag

$$\begin{array}{r} \text{+C flag} \\ \phantom{0x} \curvearrowright \\ 0x \ 34 \ F0 \\ + \ 0x \ 22 \ 40 \\ \hline 0x \ 57 \ 30 \end{array}$$

Add two LSBytes,

if Cflag =1 after addition, then increment (+1) MSByte before MSByte addition

# 16-bit Addition

C code

```
int i, j;
```

```
i = i + j;
```

PIC18 assembly

```
CBLOCK 0x040
```

```
i:2, j:2
```

```
ENDC
```

```
;; i = i + j
```

LSByte addition



```
movf j, w ; w ← j (LSB)
```

```
addwf i, f ; i LSB ← w + i LSB
```

MSByte addition



```
movf j+1, w ; w ← j (MSB)
```

```
addwfc i+1, f ; i MSB ← w + i MSB + C
```




*addwfc* is “add W to F with carry”

*addwfc* instruction does  $d \leftarrow f + w + C$  flag. Nifty, eh?

# 16-bit Subtraction using Carry Flag

$$\begin{array}{r} \phantom{-} \phantom{0x} \phantom{34} \phantom{10} \\ \phantom{-} \phantom{0x} \phantom{34} \phantom{10} \\ - \phantom{0x} \phantom{22} \phantom{40} \\ \hline \phantom{0x} \phantom{11} \phantom{D0} \end{array}$$

- ~C flag



Subtract two LSBytes,

if Cflag =0 after subtraction (a borrow), then decrement (-1) MSByte before MSByte subtraction

# 16-bit Subtraction

C code

```
int i, j;
```

```
i = i - j;
```

PIC18 assembly

```
CBLOCK 0x040
```

```
i:2, j:2
```

```
ENDC
```

```
;; i = i - j
```

LSByte subtraction 

```
movf j, w ; w ← j (LSB)  
subwf i, f ; i LSB ← i LSB - w
```

MSByte subtraction 

```
movf j+1, w ; w ← j (MSB)  
subwfb i+1, f ; i MSB ← i MSB - w - ~C
```

*subwfb* is “subtract W from F with borrow”

*subwfb* instruction does  $d \leftarrow f - w - \sim C$  flag. ‘tis Handy.

There is also a *subfbw* instruction ( $d \leftarrow w - f - \sim C$  flag) which has been included in the PIC18 instruction set for confusion purposes.

# 16-bit Increment/Decrement

## (a) Increment

```
int i;  
i++;  
  
movlw 0x0  
incf i,f  
addwfc i+1,f
```

Clear W so it is zero  
for `addwfc` to MSByte.

## (b) Decrement


```
int i;  
i--;  
  
movlw 0x0  
decf i,f  
subwfb i+1,f
```

Clear W so it is zero  
for `subwfb` from MSByte

## (c) Increment

```
int i;  
i++;  
  
infsnz i,f;  
incf i+1,f;
```

Increment LSByte, skip  
MSBbyte++ if LSByte is  
nonzero.

 only works for 16-bit values,  
does not extend to 32-bit values.

The *addwfc/subwfc* methods are more general; remember those.

# 16-bit Logical Operations

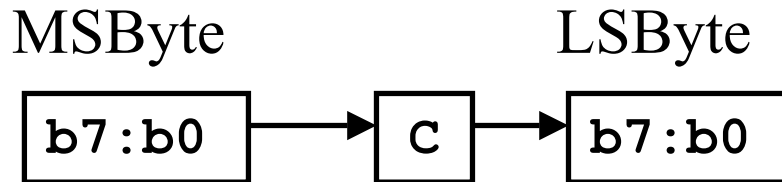
<i>C</i> code	PIC18 assembly
<code>int i, j;</code>	<code>movf j, w ; w ← j (LSB)</code>
<code>i = i &amp; j;</code>	<code>andwf i, f ; i LSB ← w &amp; i LSB</code>
	<code>movf j+1, w ; w ← j (MSB)</code>
	<code>andwf i+1, f ; i MSB ← w &amp; i MSB</code>

Bitwise logical operations on multi-byte values are easy; just perform the same operation on each byte. The order in which the bytes are computed does not matter.

# 16-bit Right Shift/ Left Shift

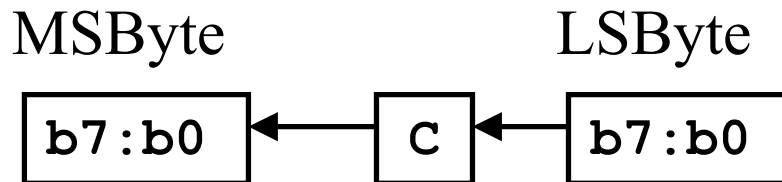
Unsigned Right Shift ( $i \gg 1$ )

Shift MSByte first, then LSByte. Use Carry flag to propagate bit between bytes.



Left Shift ( $i \ll 1$ )

Shift LSByte first, then MSByte. Use Carry flag to propagate bit between bytes.



# 16-bit Left Shift

C code

```
int i;  
i = i << 1;
```

PIC18 assembly

```
CBLOCK 0x040  
i:2  
ENDC  
;; i = i << 1
```

```
bcf STATUS,C ;clear carry
```

```
rlcf i,f ;i LSB << 1
```

```
rlcf i+1,f ;i MSB << 1
```

Clear carry for first shift, use carry to propagate bit for second shift.

# 16-bit Unsigned Right Shift

C code

```
unsigned int i;  
i = i >> 1;
```

PIC18 assembly

```
CBLOCK 0x040  
i:2  
ENDC  
  
;; i = i >> 1
```

```
bcf STATUS,C ;clear carry
```

```
rrcf i+1,f ;i MSB >> 1
```

```
rrcf i,f ;i LSB >> 1
```

Clear carry for first shift, use carry to propagate bit for second shift.

# 16-bit Non-Zero Test

## In C

```
unsigned int i, j;

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

## In Assembly

```
movf    i,w      ; w = i LSB
iorwf   i+1,w    ; w = i LSB | i MSB
bz      end_if   ; skip if Z=1, i is 0
movf    i,w
addwf   j,f
movf    i+1,w
addwfc  j+1,f
end_if
...rest of code
```

Test i for zero/non-zero by bitwise OR'ing of MSB/LSB

// if body  
j = j + i;

Bitwise OR the LSByte and MSByte; if result is non-zero, the 16-bit value is zero. You CANNOT just do:

```
movf i,w
movf i+1,w
bz   end_if
```

in this case, Z flag is only based on MSByte!!!!

# Another Example of 16-bit Testing

**In C**

```
unsigned int i, j;

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

**In Assembly**

```
movf    i,w          ; w = i LSB
iorwf   i+1,w        ; w = i LSB | i MSB
bnz     if_body      ; do if_body if i is non-zero
movf    j,w          ; w = j LSB
iorwf   j+1,w        ; w = j LSB | j MSB
bnz     end_if       ; skip if i==0 and j!=0
; do if_body if i!=0 or j==0
if_body
movf    i,w
addwf   j,f
movf    i+1,w
addwfc  j+1,f
end_if
...rest of code
```

Test i for zero/non-zero by bitwise OR'ing of MSB/LSB

Test j for zero/non-zero by bitwise OR'ing of MSB/LSB

// if body → j = j + i;

Note that the logical OR operation (`||`) has nothing to do with the fact that Bitwise OR is used for testing each 16-bit value for zero or non-zero!

If the code is `if (i && !j)`, bitwise OR is still used for zero/non-zero test.

# 16-bit Equality

## In C

```
int i, j;  
  
if (i == j) {  
    // if body  
    i = i + j;  
}  
// ... rest of code
```

## In Assembly

```
movf    j,w  
subwf   i,w    ; i - j, LSByte  
bnz     end_if ; if not equal, skip  
movf    j+1,w  
subwf   i+1,w  ; i - j, MSByte  
bnz     end_if  
[movwf  j,w  
 addwf  i,f  
 movwf  j+1,w  
 addwfc i+1,f  
] ← // if body  
      i = i + j;  
end_if  
...rest of code
```

If the result byte of either the LSB or MSB subtractions is non-zero, then values are not equal, so skip.

# 16-bit Inequality

## In C

```
int i, j;  
  
if (i != j) {  
    // if body  
    i = i + j;  
}  
// ... rest of code
```

## In Assembly

```
movf    j,w  
subwf   i,w      ; i - j, LSByte  
bnz     if_body  ; if not equal, do body  
movf    j+1,w  
subwf   i+1,w    ; i - j, MSByte  
bz      end_if  
if_body  
movwf   j,w  
addwf   i,f      ←  
movwf   j+1,w    // if body  
addwfc  i+1,f    i = i + j;  
end_if ←  
...rest of code
```

if both subtractions are zero, then skip

If the result bytes of both the LSB or MSB subtractions are zero, then skip.

# 16-bit Greater-than (>) Test

## In C

```
unsigned int i, j;

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

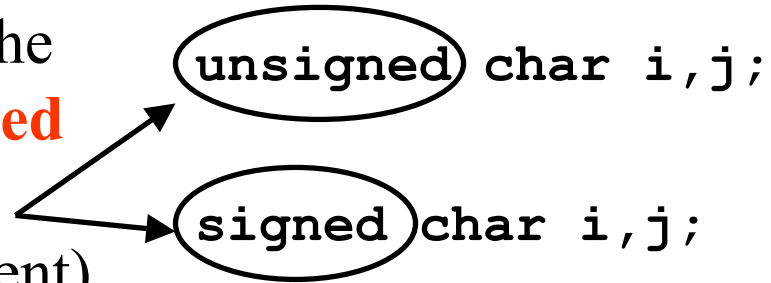
## In Assembly

```
    movf    i,w        ;
    subwf   j,w        ; j - i LSByte
    movf    i+1,w     ;
    subwfb  j+1,w     ; j - i MSByte
    bc     end_if     ; skip if C=1, i <= j
    movf    i,w        ;
    addwf   j,f        ; j + i LSByte
    movf    i+1,w     ;
    addwf   j+1,f     ; j + i MSByte
    → end_if
    ..rest of code..
```

Simply do a 16-bit subtraction, and test the C flag in the same way as was done for an 8-bit comparison.

# Unsigned vs. Signed Data

These modifiers determine if the variables are treated as **unsigned** or **signed** values (signed is assumed if no modifier is present). Signed values use 2's complement representation.



The following slides discuss how signed integers are represented in binary.

# Signed Integer Representation

We have been ignoring large sets of numbers so far; ie. the sets of signed integers, fractional numbers, and floating point numbers.

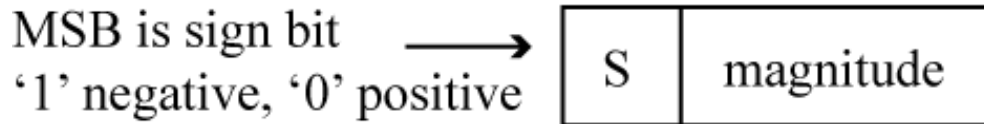
We will not talk about fractional number representation (10.3456) or floating point representation (i.e.  $9.23 * 10^{13}$ ).

We **WILL** talk about signed integer representation.

The *PROBLEM* with signed integers ( - 45, + 27, -99) is the SIGN! How do we encode the sign?

The sign is an extra piece of information that has to be encoded in addition to the magnitude. Hmmmmm, what can we do??

# Signed Magnitude Representation

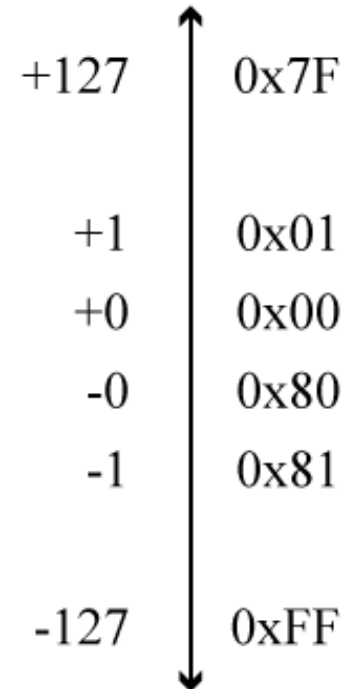


+5 =	0   0000101	= 0x05
-5 =	1   0000101	= 0x85
+0 =	0   0000000	= 0x00
-0 =	1   0000000	= 0x80

Number Line  
8 bits, range is

-127 to +127

For  $n$  bits:  
 $-2^{n-1} - 1$  to  $+2^{n-1} - 1$



Problems: two zeros (+, - zero). Also  $+K + (-K)$  does not equal zero.

$$-5 + 5 = 0x85 + 0x05 = 0x8A = -10 !!!$$

# 1's Complement Representation

One's Complement

$$-N = \sim(+N)$$

$$+5 = 0b00000101 = 0x05$$

$$-5 = 0b11111010 = 0xFA$$

$$+0 = 0b00000000 = 0x00$$

$$-0 = 0b11111111 = 0xFF$$

Number Line  
8 bits, range is

-127 to +127

For  $n$  bits:

$-2^{n-1} - 1$  to  $+2^{n-1} - 1$

+127

+1

+0

-0

-1

-127

0x7F

0x01

0x00

0xFF

0xFE

0x80

Problems: two zeros (+, - zero), and addition is off by 1 LSb if two negative numbers are added or a negative and a positive are added.

# 2's Complement Representation

Two's Complement

$$-N = \sim(+N) + 1$$

$$= 0 - (N)$$

$$+5 = 0b00000101 = 0x05$$

$$-5 = 0x00 - 0x05 = 0xFB$$

$$0 = 0b00000000 = 0x00$$

$$-128 = 0 - 128$$

$$= 0x00 - 0x80 = 0x80$$

Number Line

8 bits, range is

-128 to +127

For  $n$  bits:

$-2^{n-1}$  to  $+2^{n-1} - 1$

+127

+1

0

-1

-2

-128

0x7F

0x01

0x00

0xFF

0xFE

0x80

Fixes all problems of previous methods, this is the standard way of representing signed integers. From this point on, any signed integer is assumed to be in two's complement format unless explicitly stated otherwise.

# A common Question from Students

A question I get asked by students all the time is :

Given a hex number, how do I know if it is in 2's complement or 1's complement; is it already in 2's complement or do I have put it in 2's complement, etc, yadda,yadda, yadda,confusion,....

If I write a HEX number, I will ask for a decimal representation if you INTERPRET the encoding as a particular method (i.e, either 2's complement, 1's complement, signed magnitude).

A Hex or binary number BY ITSELF can represent ANYTHING (unsigned number, signed number, character code, colored llamas, etc). You MUST HAVE additional information that tells you what the encoding of the bits mean.

# Example Conversions

0xFE as an 8 bit unsigned integer = 254

0xFE as an 8 bit 2's Complement integer = -2

0x7F as an 8 bit unsigned integer = 127

0x7f as an 8 bit 2's Complement integer = +127

To do hex to signed decimal conversion, we need to determine sign (Step 1), determine Magnitude (step 2), combine sign and magnitude (Step 3)

# Signed Decimal to 2's complement

Convert +34, -20 to 8-bit 2's complement.

Step 1: Ignore the sign, convert the magnitude of the number to hex.

$$34 = 2 * 16 + 2 = 0x22$$

$$20 = 1 * 16 + 4 = 0x14$$

Step 2 (for positive decimal number): If the decimal number was positive, then you are finished!

+34 as an 8 bit 2s complement number is 0x22

# Signed Decimal to Hex conversion (cont)

Step 2 (for negative decimal number): Need to do more work if decimal number was negative. To get the final representation, we will use the fact that:

$$0x00 - (+N) = -N$$

$$\begin{array}{r} 0x00 \\ - 0x14 \\ \hline \end{array}$$

0xEC This is the final result.

-20 as an 8-bit 2s complement number is 0xEC

# Signed Decimal to 2's complement Summary

If $n$ is positive	Convert $n$ to hex	$+60 = 0x3C$
If $n$ is negative	Ignore sign, convert $n$ to hex.  Then subtract from zero.	$-60 = ??$ $60 = 0x3C$  $-60 = 0x00 - 0x3C$ $= 0xC4$

# Hex to Signed Decimal Conversion Rules

Given a Hex number, and you are told to convert to a signed integer  
(Hex number uses 2s complement encoding)

STEP 1: Determine the sign! If the Most Significant Bit is zero, the sign is positive. If the MSB is one, the sign is negative.

0xF0 = 0b 11110000 (MSB is '1'), so sign of result is '-'  
0x64 = 0b 01100100 (MSB is '0'), so sign of result is '+'.

If the Most Significant Hex Digit is > 7, then MSB = '1' !!!  
(eg, 0x8,9,A,B,C,D,E,F => MSB = '1' !!!)

# Hex to Signed Decimal (cont)

STEP 2 (positive sign): If the sign is POSITIVE, then just convert the hex value to decimal.

0x64 is a positive number, decimal value is  
 $6 * 16 + 4 = 100.$

Final answer is **+100**.

0x64, a 8-bit 2's Complement integer, in decimal is +100

# Hex to Signed Decimal (cont)

STEP 2 (negative sign): If the sign is Negative, then need to compute the magnitude of the number.

We will use the fact that  $0 - (-N) = +N$ , which is the magnitude!

$$\begin{array}{r} 0x00 \\ - 0xF0 \\ \hline 0x10 = 16 \end{array}$$

STEP 3 : Just combine the sign and magnitude to get the result.

0xF0, an 8-bit 2's Complement integer, is decimal -16

0x64, an 8-bit 2's Complement integer, is decimal +100

# 2's complement Hex to Decimal Summary

If MSb is 0 (hex digit < 8)	Number is positive, convert to decimal	$0x4D = +77$
If MSb is 1 (hex digit > 7)	Number is negative, subtract from zero, convert to decimal to find magnitude.  Combine sign and magnitude	$0xB3 = ??$ $0x00 - 0xB3 = 0x4D$ $0x4D = 77$  $0xB3 = -77$

# Two's Complement Overflow

Consider two 8-bit 2's complement numbers. I can represent the signed integers -128 to +127 using this representation.

Consider  $(+1) + (+127) = +128$ . The number +128 is OUT of the RANGE that I can represent with 8 bits. What happens when I do the binary addition?

$$+127 = 0x7F$$

$$+ \quad +1 = 0x01$$

-----

128  $\neq$  0x80 (this is actually -128 as a two's complement number!!! - the wrong answer!!!)

How do I know if overflowed occurred? Added two POSITIVE numbers, and got a NEGATIVE result.

# Two's Complement Overflow, Addition

Two's complement overflow for addition occurs if:

$+N + +M = -R$  (add two positive, get a negative)

$(-N) + (-M) = +R$  (add two negative, get a positive)

CANNOT get two's complement overflow when adding numbers of different signs.

The Carry out of the MSB means **nothing** if the numbers are two's complement numbers.

In hardware, overflow is detected by the boolean equation:

$$V = C_{\text{MSB}} \text{ xor } C_{\text{MSB}-1}$$

For N bits,  $C_{\text{MSB}} = \text{Carry out of bit}[N-1]$

$C_{\text{MSB}-1} = \text{Carry out of bit}[N-2]$  v 0.6

# Two's Complement Overflow, Subtraction

Two's complement overflow for subtraction occurs if:

$$\begin{aligned} -N - +M = +R & \quad (\text{this is just } -N + (-M) = +R, \text{ stated differently}) \\ (+N) - (-M) = -R & \quad (\text{this is just } +N + (+M) = -R, \text{ stated differently}) \end{aligned}$$

CANNOT get two's complement overflow when subtracting numbers of the same signs.

# Some Examples

adder logic	unsigned	signed	adder logic	unsigned	signed
$\begin{array}{r} 0x01 \\ + 0xFF \\ \hline 0x00 \end{array}$	$\begin{array}{r} 1 \\ + 255 \\ \hline 0 \end{array}$	$\begin{array}{r} +1 \\ + -1 \\ \hline 0 \end{array}$	$\begin{array}{r} 0xFF \\ + 0x80 \\ \hline 0x7F \end{array}$	$\begin{array}{r} 255 \\ + 128 \\ \hline 127 \end{array}$	$\begin{array}{r} -1 \\ + -128 \\ \hline +127 \end{array}$
<p>C=1, Z=1, V=0, N=0</p>			<p>C=1, Z=0, V=1, N=0</p>		

---

adder logic	unsigned	signed	adder logic	unsigned	signed
$\begin{array}{r} 0x7F \\ + 0x01 \\ \hline 0x80 \end{array}$	$\begin{array}{r} 127 \\ + 1 \\ \hline 128 \end{array}$	$\begin{array}{r} +127 \\ + +1 \\ \hline -128 \end{array}$	$\begin{array}{r} 0x80 \\ + 0x20 \\ \hline 0xA0 \end{array}$	$\begin{array}{r} 128 \\ + 32 \\ \hline 160 \end{array}$	$\begin{array}{r} -128 \\ + +32 \\ \hline -96 \end{array}$
<p>C=0, Z=0, V=1, N=1</p>			<p>C=0, Z=0, V=0, N=1</p>		

# Adding Precision (unsigned)

What if we want to take an unsigned number and add more bits to it?

Just add zeros to the left.

$$\begin{aligned} 128 &= 0x80 && (8 \text{ bits}) \\ &= 0x0080 && (16 \text{ bits}) \\ &= 0x00000080 && (32 \text{ bits}) \end{aligned}$$

# Adding Precision (two's complement)

What if we want to take a 2's Complement number and add more bits to it?

Take whatever the SIGN BIT is, and extend it to the left.

$$\begin{aligned} -128 &= 0x80 &= 0b \mathbf{1}0000000 & (8 \text{ bits}) \\ &= 0xFF80 &= 0b \mathbf{11111111}10000000 & (16 \text{ bits}) \\ &= 0xFFFFFFFF80 && (32 \text{ bits}) \end{aligned}$$

$$\begin{aligned} +127 &= 0x7F &= 0b \mathbf{0}1111111 & (8 \text{ bits}) \\ &= 0x007F &= 0b \mathbf{00000000}1111111 & (16 \text{ bits}) \\ &= 0x0000007F && (32 \text{ bits}) \end{aligned}$$

This is called SIGN EXTENSION. Extending the MSB to the left works for two's complement numbers and unsigned numbers.

# Unsigned vs. Signed Arith. Implementation

2's complement is nice in that the binary adder logic circuit used for unsigned numbers can also be used for 2's complement numbers.

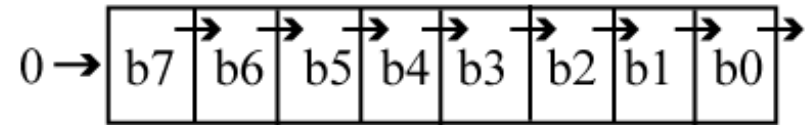
This is NOT TRUE for some operations.

Operations that work differently for signed, unsigned	Operations that work the same for unsigned, unsigned
comparison(>, >=, <, <=), right shift (>>), multiplication, division	Bitwise logical, addition, subtraction, left shift (<<), equality, inequality.

If these operations are implemented in logic gates, must use different logic networks. If implemented in assembly code, must use different sequences of instructions.

# Signed Right Shift (>>)

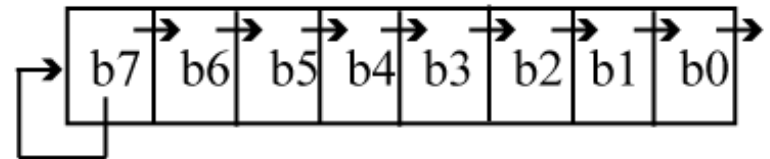
Shift in  
0 1 0 0 0 0 0 0 0 = 0x80 = -128  
↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓  
0 1 0 0 0 0 0 0 = 0x40 = +64



(a) Logical Shift Right 0x40 = 0x80 >> 1

---

Shift in 1 0 0 0 0 0 0 0 0 = 0x80 = -128  
↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓  
1 1 0 0 0 0 0 0 = 0xC0 = -64



(b) Arithmetic Shift Right 0xC0 = 0x80 >> 1

# Signed Right Shift in PIC18 Assembly

In C	In Assembly
signed int i;	bcf STATUS,C ; clear carry
	btfsc i+1, 7 ; sign bit=0?
i = i >> 1;	bsf STATUS,C ; set carry
	rrcf i+1,f ; i >> 1, MSByte
	rrcf i,f ; i >> 1, LSByte

Set carry to be same as sign bit before shift.

Note used of *signed* qualifier on *int* declaration; if this is left off the default assumption is *signed*.

Use the *unsigned* qualifier if you want unsigned data types.

Note: The right shift operation (>>) in ANSI C is *compiler dependent*. The HI-TECH PICC18 compiler preserves the sign bit, while the Microchip MCC18 compiler always shifts in a '0', regardless of signed or unsigned. In this class, we will assume the sign bit is preserved for signed data types.

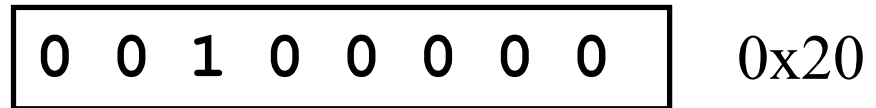
# Signed Left Shift (<<)?

There is no need for signed left shift. If the sign bit changes due to the shift operation, then overflow occurs!

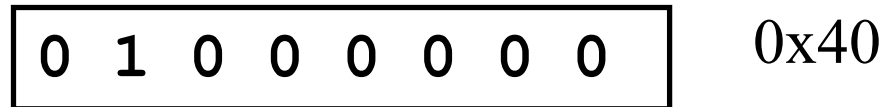
$$+32 * 2 = +64$$



$$0x20 \ll 1 == 0x40$$



no overflow, +64 can be represented in 8 bits



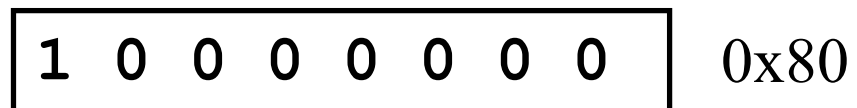
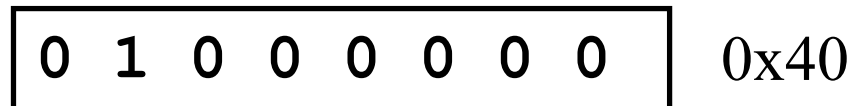
$$+64 * 2 = +128$$



$$0x40 \ll 1 == 0x80 = -128$$

overflow!! +128 cannot be represented in 8 bits!

Multiplied positive number by 2, got a negative number!



# Signed Comparisons

The table below shows what happens if unsigned comparisons are used for signed numbers in the case of '>'. If the numbers have different signs, the comparison gives the wrong result.

Numbers	As unsigned	$i > j$ ?	As signed	$i > j$ ?
$i = 0x7f,$ $j = 0x01$	$i = 127,$ $j = 01$	True	$i = +127,$ $j = +01$	True
$i = 0x80,$ $j = 0xFF$	$i = 128,$ $j = 256$	False	$i = -128,$ $j = -1$	False
$i = 0x80,$ $j = 0x7F$	$i = 128,$ $j = 127$	<b>True</b>	$i = -128,$ $j = +127$	<b>False</b>
$i = 0x01$ $j = 0xFF$	$i = 1,$ $j = 255$	<b>False</b>	$i = 1,$ $j = -1$	<b>True</b>

# PIC18 Signed Compare

The PIC18 has two flags that are useful for signed compare:

V (overflow flag), set on two's complement overflow

N (negative flag), set if MSB = 1 after operation

Also need branches based on single flag conditions:

*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)

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

# Using N, V flags for Signed Compare

To compare  $i > j$ , perform  $j - i$  (answer *should be* negative)

After  $j-i$ , if  $V = 0$  (correct result, no overflow)

if  $N=1$  (result negative) then  $i > j$  ( $V=0, N=1$ )

else  $N=0$  (answer positive) so  $j \geq i$  ( $V=0, N=0$ )

After  $j-i$ , if  $V = 1$  (incorrect result)

if  $N=0$  (result positive) then  $i > j$  ( $V=1, N=0$ )

else  $N=1$  (result negative) so  $j \geq i$  ( $V=1, N=1$ )

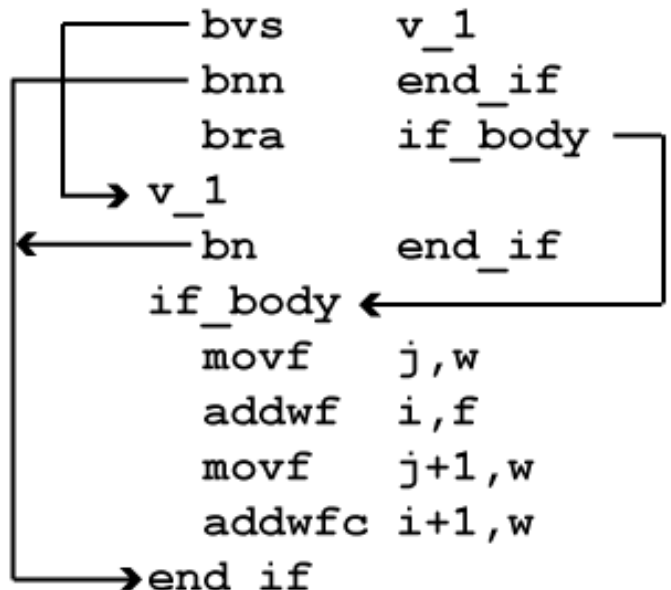
---

Most processors have *unsigned compare* instructions (operate from Z, C flags) and *signed compare* instructions (operate from Z, N, V flags). The PIC18 only has unsigned 8-bit compare instructions (*cpfsgt*, *cpfslt*). Signed comparison must be implemented using subtraction and flag tests of N, V.

# PIC18 Signed Compare (16-bit)

```
In C  
signed int i,j;  
  
if (i > j) {  
    // if_body  
    i = i + j;  
}  
// rest of code
```

```
In Assembly  
movf    i,w          ;  
subwfb  j,w          ; j-i  LSByte  
movf    i+1,w       ;  
subwfb  j+w,w       ; j-i  MSByte  
bvs     v_1  
bnn     end_if      ; skip if V=0,N=0  
bra     if_body     ; V=0, N=1  
bn      end_if      ; skip if V=1,N=1  
if_body ←  
movf    j,w  
addwfb  i,f          ; i=i+j, LSByte  
movf    j+1,w  
addwfb  i+1,w        ; i=i+j, MSByte  
end_if  
...rest of code...
```



typo: replace “bvs” with “bov”

# Signed Comparison Summary

Comparison	Operation	True Test
if (i > k) {}	$k - i$	V=0, N= 1 OR V=1, N=0
if (i >= k) {}	$i - k$	V=0, N=0 OR V=1 , N=1

Please remember this by understanding how the flag tests are derived – don't try to memorize it.

Memorization is what Doctors and Lawyers do.

Understanding, Derivation is what Engineers and Scientists do.

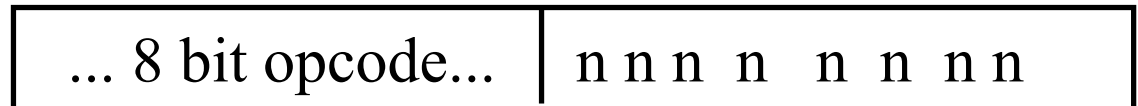
# *branch versus goto*

Recall that a *goto* used two instruction words which encoded a 20-bit value that is loaded directly into the PC.

The general format of the machine code for a conditional flag branch is:

B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	
1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	
5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0

*brz, brnz, brnc, bc,*  
etc.



where  $nnnnnnnn = N$  is a 2's complement, 8-bit value. The target address of the branch is computed as:

$$\text{branch target} = \text{PC} + 2 + 2*N$$

PC is the location of the branch instruction. The  $2*N$  is needed because  $N$  is word offset, need a byte offset.

# *branch* Machine Code examples

Assume a branch is located as location 0x220, and the target address is location 0x230. The branch offset is computed as:

$$\text{branch target} = \text{PC} + 2 + 2 * \text{N}$$

$$\text{N} = [\text{branch\_target} - (\text{PC} + 2)] / 2$$

$$= [0\text{x}230 - (0\text{x}220 + 2)] / 2 = 0\text{x}0\text{E} / 2 = 0\text{x}07$$

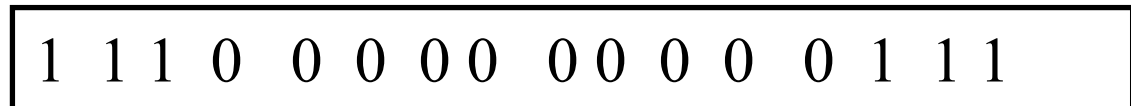


PC+2 because the PC is already incremented to next instruction when branch executes.

If the instruction is a 'bz', then the machine code is:

<b>B</b>	<b>B</b>	<b>B</b>	<b>B</b>	<b>B</b>	<b>B</b>	<b>B</b>	<b>B</b>	<b>B</b>	<b>B</b>	<b>B</b>	<b>B</b>	<b>B</b>	<b>B</b>	<b>B</b>	<b>B</b>
1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0

*bz* +7



*bz* +7



0xE007

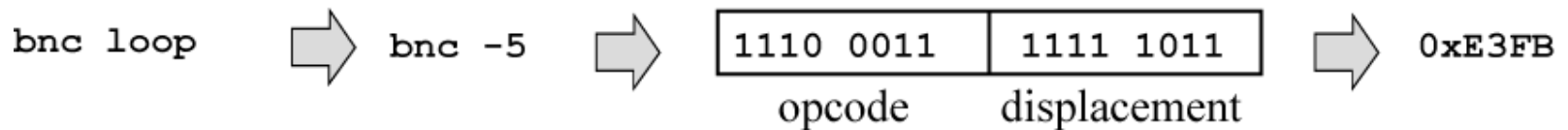
# *branch* Machine Code examples (cont)

What is the machine code for the *bnc* below?

Solution: just pick an arbitrary starting location, like 0x0100

location (hex)	machine code (hex)	instruction
0100	5000	loop movf i,w
0102	2601	addwf j,f ; j = j+1
0104	5000	movf i,w
0106	5C01	subwf j,w ; j-i
0108	E3FB	bnc loop ; branch to top

$$\begin{aligned}
 \text{branch\_target} &= \text{PC} + 2 + 2 * n && 0x0100 \quad (\text{branch\_target}) \\
 n &= [\text{branch\_target} - (\text{PC} + 2)] / 2 && - \frac{0x010A}{0xFFFF6} \quad (\text{PC} + 2) \\
 &= 0x0100 - (0x0108 + 2) / 2 \\
 &= 0xFFFF6 / 2 = 0xFFFF6 \gg 1 \\
 &= 0xFFFFB = 0xFB \text{ (8 bits)} = -5 \text{ (displacement)}
 \end{aligned}$$



# *branch, goto* Pros/Cons

The conditional flag branches (*bz, bnz, bn, bnn, bc, bnc, bov, bnov*) use an 8-bit offset. This means the target address must be within  $-128$  to  $+127$  instruction words ( $-256$  to  $+255$  bytes) from the branch.

A *branch* has **limited range**. This is ok, most loops are not that large.

The advantage of a *goto* is that it **can jump anyway in program memory**.

The advantage of a *branch* is it **only takes one instruction word**.

A *bra* (branch always) has an 11 bit offset. The target address must be within  $-1024$  to  $+1023$  instruction words ( $-2048$  to  $+2047$  bytes) from the branch.

# What do you need to know?

- Extended precision operations (16-bit) for logical, addition/subtraction, increment/decrement, shift left, shift right, equality
- 16-bit unsigned comparisons
- 16-bit signed comparisons using N, V flags.
- How to determine the machine code for a branch instruction.