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 (PIC16)
1 (8 bits)	0 to 255	char
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 int* 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

```
Address:  0x20 0x21 0x22 0x23 0x24 0x25
contents: 0x57 0xA4 0xF2 0x3D 0xB8 0x38    little endian
contents: 0xA4 0x57 0x38 0xB8 0x3D 0xF2    big endian
          i         j
```


Which is better?

- No inherent advantage to either byte ordering
- On 8-bit processors, it is the choice of the programmer or compiler writer
  - Be consistent!
- On processors that have 16-bit and 32-bit operations, the µP architects choose the byte ordering
  - Intel µPs use little endian, Motorola µPs uses big endian
  - It is a religious argument....
- In these examples, will use little endian

Multi-byte values in MPLAB

C code

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

PIC16 assembly

```
CBLOCK 0x040
 i_low, i_high
ENDC

;; i = 0xC428

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

Explicity named each byte of *i*.

Arranged in little endian order.

Multi-byte values in MPLAB

C code

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

PIC16 assembly

```
CBLOCK 0x040
 i:2
ENDC
;; i = 0xC428
movlw 0x28
movwf i        ; LSB = 0x28
movlw 0xC4
movwf i+1      ;MSB = 0xC4
```

Reserve two bytes of space for *i*.

i refers to loc 0x20

i+1 refers to loc 0x21

16-bit Addition using Carry Flag

```
         +C flag
   0x 34 F0

+  0x 22 40
-----------
   0x 57 30
```

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;
```

PIC16 assembly

```
CBLOCK 0x040
i:2,j:2
ENDC
;; i = i + j
```

LSByte adddition →

```
movf  j,w   ; w ← j (LSB)
addwf i,f   ;i LSB ← w + i LSB
```

if C=1, add 1 to MSByte →

```
btfsc STATUS, C
incf  i+1,f   ;i MSB ← i MSB + 1
```

MSByte adddition →

```
movf j+1,w   ; w ← j (MSB)
addwf i+1,f  ; i MSB← w +i MSB
```


16-bit Subtraction using Carry Flag

```
       - ~C flag
   0x 34 10

 - 0x 22 40
 ----------
   0x 11 D0
```

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;
```

PIC16 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
```

if C=0, subtract 1 from MSByte →

```
btfss STATUS, C
decf  i+1,f   ;i MSB ← i MSB - 1
```

MSByte subtraction →

```
movf j+1,w   ; w ← j (MSB)
subwf i+1,f  ; i MSB← i MSB - w
```


PIC18 Add/Sub with Carry

addwfc f, d ; d ← d + f + Cflag (add with carry)

subwfb f, d ; d ← f - d - ~Cflag (subtract with borrow)

```
int i,j;
i = i + j;
```

PIC16 assembly

```
movf  j,w   ; w ←j(LSB)
addwf i,f   ;LSB addition

btfsc STATUS, C
incf  i+1,f  ;MSB+1

movf j+1,w   ; w← j (MSB)
addwf i+1,f  ;MSB addition
```

PIC18 assembly

```
movf  j,w   ; w ←j(LSB)
addwf i,f   ; LSB addition

movf j+1,w    ; w← j (MSB)
addwfc i+1,f  ;MSB addition
```

MSByte, use add with carry

16-bit Increment/Decrement

On the PIC18, the increment/decrement instructions affect the Carry flag, so can do increment/decrement of LSByte followed by add-with-carry/subtract-with-borrow to MSByte.

On the PIC16, the increment/decrement instructions only affect the Z flag, so cannot use C flag approach if *incf/decf* used (can always just do add + 1). Use the procedure below:

```
   0x33 FF          0x34 (00)
 +       1        -        1
 ---------        ---------
   0x34 (00)        0x33 FF
```

If LSByte = 0 **after** increment, then increment MSbyte

If LSByte = 0 **before** increment, then decrement MSbyte

16-bit Increment

C code

```
int i;
i++;
```

PIC16 assembly

```
CBLOCK 0x040
i:2
ENDC
;; i++;
```

LSByte increment →

```
incf  i,f   ; i(LSB)+ 1
```

if Z=0, continue →

```
btfsc STATUS,Z   ;Z=0?
```

Z=1, so MSByte increment →

```
incf  i+1,f   ;i MSB ← i MSB + 1

skip
 ...continue...
```

16-bit Decrement

C code

```
int i;
 i--;
```

PIC16 assembly

```
CBLOCK 0x040
i:2
ENDC
;; i--;
```

MSByte decrement if LSByte==0

```
movf i,f;  ; test LSByte
btfsc STATUS, Z
decf i+1;  ; MSByte--
```

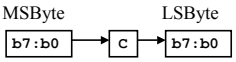
decrement LSByte

```
decf i;    ; LSByte--
```


16-bit Right Shift/ Left Shift

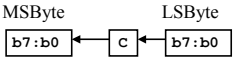
Unsigned Right Shift (i >> 1)

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



Left Shift (i << 1)

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



16-bit Left Shift

C code

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

PIC16 assembly

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

```
bcf STATUS,C   ;clear carry
rlf i          ;i LSB << 1
rlf i+1        ;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 >> 1;
```

PIC16 assembly

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

```
bcf STATUS,C   ;clear carry
rrf i+1        ;i MSB >> 1
rrf i          ;i LSB >> 1
```

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

16-bit Logical Operations

C code

```
int i,j;
i = i & j;
```

PIC16 assembly

```
movf  j,w  ; w ← j (LSB)
andwf i,f  ;i LSB ← w && i LSB
movf j+1,w  ; w ← j (MSB)
andwf i+1,f  ; i MSB← w  && i MSB
```

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.

Unsigned vs. Signed Operations

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

```
unsigned char i,j;
signed char i,j;
```

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 (<<)

Signed Right Shift (>>)

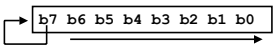
The value 0x80 = -128 in two's complement.

Recall that right shift is same as divide by two. Then

-128 >> 1 == -128 / 2 == -64 == 0xC0

If unsigned right shift is performed (0 shifted into MSB), then 0x80 >> 1 == 0x40 == +64 , the wrong answer!!

When doing a signed right shift, the MSB must be kept the same value (this is also known as an *arithmetic right shift*). Makes sense, dividing a negative number by 2 should not change the sign!!



Signed Right Shift in PIC16 Assembly

C code

```
signed char i;
i >> 1;
```

PIC16 assembly

```
;; i >> 1
```

Set carry to be same as sign bit before shift.

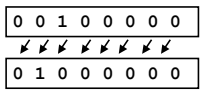
```
bcf STATUS,C    ;clear carry
btfsc i,7       ;sign bit =1?
bsf STATUS,C    ;set carry
rrf i           ;i >> 1
```


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

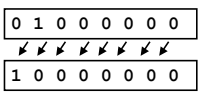
0x20 << 1 == 0x40



no overflow, +64 can be represented in 8 bits

+64 * 2 = +128

0x40 << 1 == 0x80 = -128



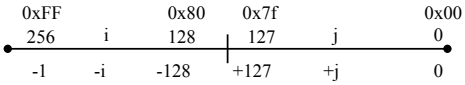
overflow!! +128 cannot be represented in 8 bits! Multiplied positive number by 2, got a negative number!

Signed Comparisons

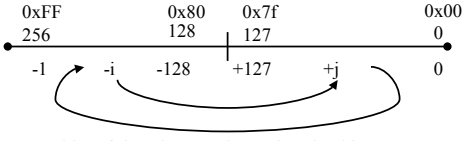
The table below shows what happens if unsigned comparisons are used for signed numbers in the case of '>'. If the numbers have different signs, the comparison gives the wrong result.

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	**True**	i = -128, j = +127	**False**
i = 0x01 j = 0xFF	i = 1, j = 255	**False**	i = 1, j = -1	**True**

PIC16: Signed Comparisons



Unsigned *i* > *j* true if *i* is to left of *j*, need for -*i* and +*j* to swap places if want to use unsigned comparison



Swap sides of -i, +*j* by complementing sign bits

PIC16: Signed Comparison Algorithm #1

Steps for *i* > *j* signed comparison

operand adjustment

1. Load *j* into *w* reg.
2. Complement *w* sign bit by XOR'ing with 0x80.
3. Store in temporary location (you pick), call this *temp*.
4. Load *i* into *w* reg.
5. Complement *w* sign bit by XOR'ing with 0x80.
6. Subtract *w* from *temp* (new *j* - new *i*).
7. If carry = 0, then *i* > *j*

Unsigned compare

Signed Compare (>, Alg #1)

C code

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

PIC16

```
  movf  j,w   ; w ← j
  xorlw 0x80  ; toggle j sign bit
  movwf temp  ; save new j
  movf  i,w   ; w ← i
  xorlw 0x80  ; toggle i sign bit
  subwf temp,w ; w← new j-new i
  btfsc STATUS,C
  goto  skip  ;C=1, j>= i
ifbody
  movf  j,w
  addwf i,f   ;i = i + j
skip
 ;; do stuff...
```

Must use temporary location because do not want to modify j!!!! Requires 8 instructions, not counting *if* body.

PIC16: Signed Comparison, Algorithm #2

Steps for $i > j$ signed comparison

check if signs equal

1. Load *j* into *w* reg.
2. XOR *w* with i, store in *w*
3. If bit 7 = 0, then both signs the same. Goto 5, do unsigned compare.
4. If bit 7 of j = 1, then j is negative, so i > j!

5. Load *i* into w reg
6. Subtract *w* from *j* (*j* - *i*).
7. If carry = 0, then $i > j$

Unsigned compare

Signed Compare (>, Alg. #2)

C code

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

PIC16

```
  movf  j,w   ; w ← j
  xorwf i,w   ; w ← j ^ i
  btfss WREG,7
  goto docmp  ; unsigned cmp
  btfss j,7   ; check j sign
  goto skip   ; j pos, so i < j
  goto ifbody  ; j neg, so i > j
docmp
  movf i,w    ; w ← i
  subwf j,w   ; w← j- i
  btfsc STATUS,C
  goto  skip  ;C=1, j>= i
ifbody
  movf  j,w
  addwf i,f   ;i = i + j
skip
 ;; do stuff...
```

Does not need temporary location, but requires more instructions (11 vs. 8)

PIC18 Signed Compare

The PIC18 added two additional flags
V (overflow flag), set on two's complement overflow
N (negative flag), set if MSB = 1 after operation

Also added 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*, will discuss exact operation later.

Using N,V flags for Signed Compare

To compare i > j, perform j – i

After *j-i*, if V = 0 (correct result, no overflow)
if N=1 (result negative) then i > j
else N=0 (answer positive) so j >= i

After *j-i*, if V = 1 (incorrect result)
if N=0 (result positive) then i > j
else N=1 (result negative) so j >= i

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 compare instructions (*cpfsgt, cpfslt)* but does have the V,N and branches based on these flags. The PIC16 only has Z,C flags and no dedicated compare instructions.

PIC18 Signed Compare (Assembly)

C code

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

Does not need a temporary location, requires 6 instructions outside of *if* body.

PIC18

```
  movf  i,w   ; w ← i
  subwf j,w   ; w ← j - i
  bvs   v_1
  bnn   skip ; V=0,N=0 j>=i
  bra   ifbody; V=0,N=1
v_1
  bn    skip ; V=1, N=1 j>=i
ifbody       ;V=1,N=0
  movf  j,w
  addwf i,f   ;i = i + j
skip
 ;; do stuff...
```

What do you need to know?

- PIC16 extended precision operations for logical, addition/subtraction, increment/decrement, shift left, shift right
- PIC18 add with carry, subtract with borrow
- PIC16 methods for signed comparison
- How to use N,V flags of PIC18 for signed comparison