Extended Precision Operations

To represent larger numbers, more bits are needed.

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

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Unsigned Range</th>
<th>C Data Type (PIC 24 compiler)</th>
<th>Book’s Data type definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (8 bits)</td>
<td>0 to 255</td>
<td>unsigned char</td>
<td>uint8</td>
</tr>
<tr>
<td>2 (16 bits)</td>
<td>0 to 65,535</td>
<td>unsigned short</td>
<td>uint16</td>
</tr>
<tr>
<td>2 (16 bits)</td>
<td>0 to 65,535</td>
<td>unsigned int</td>
<td>uint16</td>
</tr>
<tr>
<td>4 (32 bits)</td>
<td>0 to 4,294,967,295</td>
<td>unsigned long</td>
<td>uint32</td>
</tr>
</tbody>
</table>

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).
# 32-bit Data Moves

<table>
<thead>
<tr>
<th>Name</th>
<th>Mnemonic</th>
<th>Operation</th>
</tr>
</thead>
</table>
| double Word Move from Source to \texttt{Wnd} | mov.d \texttt{Ws}, \texttt{Wnd}  
(recall that \texttt{Ws} includes indirect modes) | \((\texttt{Ws:Ws+1}) \rightarrow \texttt{Wnd:Wnd+1}\) |
| double Word Move from \texttt{Wns} to Destination | mov.d \texttt{Wns}, \texttt{Wd}  
(recall that \texttt{Wd} includes indirect modes) | \((\texttt{Wns:Wns+1}) \rightarrow \texttt{Wd:Wd+1}\) |

Note: When direct register addressing is used for either \texttt{Wns} or \texttt{Wnd}, then an even-numbered register (W0, W2, W4, \ldots W14) must be used.

The term ‘double word’ refers to 32-bit data.  
A double word mov instruction copies 32 bits of data, 16 bits at a time.  
Double words must start at an even memory address.
(a) Execute: `mov.d W0, W2`

<table>
<thead>
<tr>
<th>W0</th>
<th>0x1A23</th>
<th>W0</th>
<th>0x1A23</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1</td>
<td>0x0800</td>
<td>W1</td>
<td>0x0800</td>
</tr>
<tr>
<td>W2</td>
<td>0xC9BC</td>
<td>W2</td>
<td>0x1A23</td>
</tr>
<tr>
<td>W3</td>
<td>0xFB90</td>
<td>W3</td>
<td>0x0800</td>
</tr>
<tr>
<td>0x0800</td>
<td>0x90B3</td>
<td>0x0800</td>
<td>0x90B3</td>
</tr>
<tr>
<td>0x0802</td>
<td>0x4E81</td>
<td>0x0802</td>
<td>0x4E81</td>
</tr>
</tbody>
</table>

Before | After

(b) Execute: `mov.d [W1], W2`

Source Effective Address = (W1) = 0x0800
Operation is (0x0800:0x802) → W2:W3

<table>
<thead>
<tr>
<th>W0</th>
<th>0x1A23</th>
<th>W0</th>
<th>0x1A23</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1</td>
<td>0x0800</td>
<td>W1</td>
<td>0x0800</td>
</tr>
<tr>
<td>W2</td>
<td>0xC9BC</td>
<td>W2</td>
<td>0x90B3</td>
</tr>
<tr>
<td>W3</td>
<td>0xFB90</td>
<td>W3</td>
<td>0x4E81</td>
</tr>
<tr>
<td>0x0800</td>
<td>0x90B3</td>
<td>0x0800</td>
<td>0x90B3</td>
</tr>
<tr>
<td>0x0802</td>
<td>0x4E81</td>
<td>0x0802</td>
<td>0x4E81</td>
</tr>
</tbody>
</table>

Before | After

Double Word Move Example
32-bit Variable Initialization

In C
uint32 k,p;

In Assembly
k: .space 4
p: .space 4

;reserve 4 bytes
;reserve 4 bytes

k = 0x76543210;

mov #0x3210,W0
mov W0,k
mov #0x7654,W0
mov W0,k+2

;W0 = 0x3210
;k.LSW = 0x3210
;W0 = 0x7654
;k.MSW = 0x7654

p = k;

mov k,W0
mov W0,p
mov k+2,W0
mov W0,p+2

;W0 = k.LSW;
;p.LSW = W0 = k.LSW
;W0 = k.MSW
;p.MSW = W0 = k.MSW

Order in which the words (LSW or MSW) are initialized does not matter.
LSW == Least Significant Word (lower 16-bit word)
MSW == Most Significant Word (upper 16-bit word)
32-bit Bitwise Logical Operations

In C

```c
uint32 k, j;
k = k & j;
```

In Assembly

```assembly
k: .space 4 ;reserve 4 bytes
j: .space 4 ;reserve 4 bytes
mov j, W0 ;W0 = j.LSW
and k ;k.LSW = k.LSW & W0 = k.LSW & j.LSW
mov j+2, W0 ;W0 = j.MSW
and k+2 ;k.MSW = k.MSW & W0 = k.MSW & j.MSW
```

Order in which the words (LSW or MSW) are operated upon does not matter for bitwise logical operations.
32-Bit Addition, Subtraction

(a) Addition

<table>
<thead>
<tr>
<th>C flag</th>
<th>1 ↘</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x A734 F082</td>
<td></td>
</tr>
<tr>
<td>+ 0x 13C0 4370</td>
<td></td>
</tr>
<tr>
<td>0x BAF5 33F2</td>
<td></td>
</tr>
</tbody>
</table>

(b) Subtraction

<table>
<thead>
<tr>
<th>Borrow = ~C flag</th>
<th>-1 ↘</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x A734 1082</td>
<td></td>
</tr>
<tr>
<td>- 0x 13C0 4370</td>
<td></td>
</tr>
<tr>
<td>0x 9373 CD12</td>
<td></td>
</tr>
</tbody>
</table>

Operation done on least significant words first.
Addition of most significant words includes CARRY.
Subtraction of most significant words includes BORROW (~C).
32-bit Addition/Subtraction in Assembly

In C

```c
uint32 k, j;
uint32 p, q

k = k + j;
p = p - q;
```

In Assembly

```assembly
addc k+2
add j, w0
add k
mov j+2, w0

subb p+2
sub j, w0
sub p
mov q+2, w0
```

---

**addc** is add with carry

**subb** is subtract with borrow

---

Copyright Delmar Cengage Learning 2008. All Rights Reserved.

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.
## Other Forms of Add/Sub Extended Precision

<table>
<thead>
<tr>
<th>Name</th>
<th>Mnemonic</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Add with Carry</strong></td>
<td>addr{.b} (W_b, W_s, W_d)</td>
<td>((W_b)+(W_s)+(C) \rightarrow W_d)</td>
</tr>
<tr>
<td></td>
<td>addr{.b} (W_b, #lit5, W_d)</td>
<td>((W_b)+#lit5+(C) \rightarrow W_d)</td>
</tr>
<tr>
<td></td>
<td>addr{.b} #lit10, Wn</td>
<td>((W_n)+#lit10+(C) \rightarrow W_n)</td>
</tr>
<tr>
<td></td>
<td>addr{.b} (f)</td>
<td>((f)+\text{WREG}+(C) \rightarrow f)</td>
</tr>
<tr>
<td></td>
<td>addr{.b} (f),\text{WREG}</td>
<td>((f)+\text{WREG}+(C) \rightarrow \text{WREG})</td>
</tr>
<tr>
<td><strong>Subtract with Borrow</strong></td>
<td>subb{.b} (W_b, W_s, W_d)</td>
<td>((W_b)-(W_s)-(\neg C) \rightarrow W_d)</td>
</tr>
<tr>
<td></td>
<td>subb{.b} (W_b, #lit5, W_d)</td>
<td>((W_b)-#lit5-(\neg C) \rightarrow W_d)</td>
</tr>
<tr>
<td></td>
<td>subb{.b} #lit10, Wn</td>
<td>((W_n)-#lit10-(\neg C) \rightarrow W_n)</td>
</tr>
<tr>
<td></td>
<td>subb{.b} (f)</td>
<td>((f)-\text{WREG}-(\neg C) \rightarrow f)</td>
</tr>
<tr>
<td></td>
<td>subb{.b} (f),\text{WREG}</td>
<td>((f)-\text{WREG}-(\neg C) \rightarrow \text{WREG})</td>
</tr>
<tr>
<td><strong>Subtract with Borrow Reverse</strong></td>
<td>subbr{.b} (W_b, W_s, W_d)</td>
<td>((W_s)-(W_b)-(\neg C) \rightarrow W_d)</td>
</tr>
<tr>
<td></td>
<td>subbr{.b} (W_b, #lit5, W_d)</td>
<td>#lit5-(W_b)-(\neg C) \rightarrow W_d)</td>
</tr>
<tr>
<td></td>
<td>subbr{.b} (f)</td>
<td>\text{WREG}-(f)-(\neg C) \rightarrow f)</td>
</tr>
<tr>
<td></td>
<td>subbr{.b} (f),\text{WREG}</td>
<td>\text{WREG}-(f)-(\neg C) \rightarrow \text{WREG})</td>
</tr>
<tr>
<td><strong>Compare with Borrow</strong></td>
<td>cpb{.b} (W_b, W_s)</td>
<td>((W_b)-(W_s)-(\neg C))</td>
</tr>
<tr>
<td></td>
<td>cpb{.b} (W_b, #lit5)</td>
<td>((W_b)-#lit5-(\neg C))</td>
</tr>
<tr>
<td></td>
<td>cpb{.b} (f)</td>
<td>((f)-\text{WREG}-(\neg C))</td>
</tr>
</tbody>
</table>

Note: These instructions affect the DC, N, OV, and C flags. The Z flag is sticky in that it can only be cleared by these instructions.

There is no “Compare against Zero with borrow” instruction.
(a) Subtraction,  LSW result is non-zero, MSW result is zero.

\[
\text{subb} \leftarrow \begin{array}{c}
0x \\
0x
\end{array} \begin{array}{c}
0001 \\
0000
\end{array} \begin{array}{c}
-1 \\
1000
\end{array} \begin{array}{c}
\text{sub} \\
\text{instruction}
\end{array} \\
\hline
0x \begin{array}{c}
0000 \\
0000
\end{array} \begin{array}{c}
\text{F000}
\end{array} \rightarrow Z = 0
\]

\[Z = 0\]

MSW is 0, but \text{subb} can only clear the Z flag, it cannot set the Z flag! This means that the Z flag reflects the zero status of the 32-bit operation, which is non-zero.

(b) Subtraction,  LSW result is zero, MSW result is non-zero.

\[
\text{subb} \leftarrow \begin{array}{c}
0x \\
0x
\end{array} \begin{array}{c}
0010 \\
0000
\end{array} \begin{array}{c}
-0 \\
1000
\end{array} \begin{array}{c}
\text{sub} \\
\text{instruction}
\end{array} \\
\hline
0x \begin{array}{c}
0001 \\
0000
\end{array} \begin{array}{c}
0000
\end{array} \rightarrow Z = 1
\]

\[Z = 0\]

MSW is non-zero, so Z flag is cleared; 32-bit result is non-zero.

(c) Subtraction,  LSW result is zero, MSW result is zero.

\[
\text{subb} \leftarrow \begin{array}{c}
0x \\
0x
\end{array} \begin{array}{c}
0001 \\
0001
\end{array} \begin{array}{c}
-0 \\
1000
\end{array} \begin{array}{c}
\text{sub} \\
\text{instruction}
\end{array} \\
\hline
0x \begin{array}{c}
0000 \\
0000
\end{array} \begin{array}{c}
0000
\end{array} \rightarrow Z = 1
\]

\[Z = 1\]

MSW is zero, so Z flag remains set from previous instruction; 32-bit result is zero.
32-bit Increment/Decrement

(a) Increment

```c
uint32 k;
k++;  
clr W0 ;W0 = 0
inc k ;inc k.LSW
addc k+2 ;k.MSW = k.MSW + 0 + C
```

Clear W0 so it is zero for `addc` to MSW.

(b) Decrement

```c
uint32 k;
k--;  
clr W0 ;W0 = 0
dec k ;dec k.LSW
subb k+2 ;k.MSW = k.MSW - 0 - ~C
```

Clear W0 so it is zero for `subb` from MSW.

There are no “increment with carry” or “decrement with borrow” instructions.
32-bit Increment (again), add (again)

uint32 k, j;
k = j+1;

;2-operand approach
inc j, WREG  ;W0=j.lsw+1
mov W0, k    ;k.lsw = j.lsw+1, flags unaffected
clr W0       ;W0=0, flags unaffected
addc k+2     ;k.msw = k.msw + 0 + C

-----------------------------------------------

uint32 k, j, p;
k = j+p;

;uses 3-operand add
mov j, W0    ;W0=j.lsw
mov p, W1    ;W1=p.lsw
add W0, W1, W0 ;W0=j.lsw+p.lsw
mov W0, k    ;k.lsw=j.lsw+p.lsw
mov j+2, W0  ;W0=j.msw
mov p+2, W1  ;W1=p.msw
addc W0, W1, W0 ;W0=j.msw + p.msw + C
mov W0, k+2  ;k.msw = j.msw + p.msw + C
32-bit Shift Right/Shift Left

First, using `lsr`

In C

```c
uint32_t k;

k = k >> 1;
```

In Assembly

```assembly
lsr k+2 ; k.MSW >> 1
```

Second, using `rrc`

In C

```c
uint32_t k;

k = k >> 1;
```

In Assembly

```assembly
rrc k ; k.LSW >> 1
```

(b) shift left (<<)

In C

```c
uint32_t k;

k = k << 1;
```

In Assembly

```assembly
sl k ; k.LSW << 1
rlc k+2 ; k.MSW << 1
```

Repeat sequence for multiple-position 32-bit shifts
32-bit Non-Zero Test Using Compare

In C

```c
uint32 k;

if (k) {
    if-body
}
...
```

... rest of code

In Assembly

```
clr W0
cp k
cpb k+2
bra Z,end_if
if-body stmt1
... stmtN
end_if:
... rest of code
```

Test k for zero/non-zero by comparing against zero with `cp/cpb`

- `w0 = 0`
- `k.LSW == 0?`
- `&& k.MSW == 0? (Z is sticky)`
- `skip if-body when Z=1 (k is 0)`

Compare 32-bit K against 32-bit zero using compare, compare with borrow
32-bit Non-Zero Test

Test \( k \) for zero/non-zero by bitwise OR’ing of MSW and LSW.

In C

```c
uint32 k;
if (k) {
    if-body
}
... rest of code
```

In Assembly

```assembly
mov k, W0  ; W0 = k.LSW
ior k+2, WREG  ; W0 = k.LSW | k.MSW
bra Z, end_if  ; skip if-body when Z=1 (k is 0)
if-body stmt1
    ... stmtN
end_if:
    ... rest of code
```

A little more efficient than the previous slide.

Bitwise OR the LSW and MSW; if result is non-zero, the 32-bit value is non-zero. You CANNOT just do:

```assembly
mov k
mov k+2
bz end_if  ← in this case, Z flag is only based on MSW!!!!
```

Copyright Delmar Cengage Learning 2008. All Rights Reserved.

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

V 0.7 14
Another Example of 32-bit Testing

In C
uint32 k, j;

if (k || !j) {
  if-body
}

... rest of code

In Assembly

mov k, W0 ; W0 = k.LSW
ior k+2, WREG ; W0 = k.LSW | k.MSW
bra NZ, if_body ; execute if-body when k != 0
mov j, W0 ; W0 = j.LSW
ior j+2, WREG ; W0 = j.LSW | j.MSW
bra NZ, end_if ; skip if-body when j!=0

if_body:
  if-body stmt1
  ... stmtN
end_if:
  ... rest of code

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 (k && !j), bitwise OR is still used for zero/non-zero test.
32-bit Equality

In C

```c
uint32 k, j;
if (k == j) {
    if-body code
}
... rest of code
```

In Assembly

```assembly
mov j, W0
cp k
mov j+2, W0;
```

A 32-bit comparison with `cp` used for LSW and `cpb` (compare with borrow) used for MSW.

```assembly
bra NZ, end_if
if_body:
    if-body stmt1
    ... stmtN
end_if:
    ... rest of code
```

This mov instruction affects no flags, allowing the flags affected by the first `cp` instruction to be passed to `cpb` (compare with borrow).

Use same structure as used for 8/16-bit tests, except use a 32-bit comparison (`cp` for LSW comparison, `cpb` for MSW comparison). The sticky ‘Z’ flag behavior enables this comparison.
### 32-bit Greater-Than

#### In C
```c
uint32 k, j;

if (k > j) {
  if-body code
}

... rest of code
```

#### In Assembly
```
`mov j,W0`  ; W0 = j.LSW
`cp k` ; k.LSW - j.LSW
`mov j+2,W0`  ; W0 = j.MSW
`cpb k+2` ; k.MSW - j.MSW
`bra LEU,end_if` ; skip if-body when k<=j

_if_body:
  `if-body stmt1`
  ...
  `stmtN`

_end_if:
  .... rest of code
```

Has same structure as 8/16-bit code except that a 32-bit comparison is used.

All of the if/loop structures from Chapter 4 can be used, just replace 16-bit comparisons with 32-bit comparisons.
64-bit Addition

It is straightforward to extend the previously covered 32-bit operations to 64-bit operands. 16-bit microcontroller code rarely uses 64-bit operands, so we will not have any additional coverage of 64-bit operations.
Unsigned vs. Signed Data

The signed/unsigned modifiers in type declarations 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.

<table>
<thead>
<tr>
<th>C Type</th>
<th>Book Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned char</td>
<td>uint8</td>
<td>8-bit</td>
</tr>
<tr>
<td>unsigned int</td>
<td>uint16</td>
<td>16-bit</td>
</tr>
<tr>
<td>unsigned long</td>
<td>uint32</td>
<td>32-bit</td>
</tr>
<tr>
<td>signed char</td>
<td>int8</td>
<td>8-bit</td>
</tr>
<tr>
<td>signed int</td>
<td>int16</td>
<td>16-bit</td>
</tr>
<tr>
<td>signed long</td>
<td>int32</td>
<td>32-bit</td>
</tr>
</tbody>
</table>

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 defer discussion of fractional number representation (10.3456) and floating point representation (i.e. 9.23 * 10^{13}) until the very end of the semester.

At this point, we will cover 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. Hmmmmmm, what can we do??
Signed Magnitude Representation

MSB is sign bit “1” negative, “0” positive

Number range for $n$ bits:
$-2^{n-1} - 1$ to $+2^{n-1} - 1$

8-bit range:
-127 to +127

16-bit range:
-32767 to +32767

8-bit encoding examples:

<table>
<thead>
<tr>
<th>Number</th>
<th>8-bit Encoding</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>+5</td>
<td>0000101</td>
<td>0x05</td>
</tr>
<tr>
<td>-5</td>
<td>1000101</td>
<td>0x85</td>
</tr>
<tr>
<td>+0</td>
<td>0000000</td>
<td>0x00</td>
</tr>
<tr>
<td>-0</td>
<td>1000000</td>
<td>0x80</td>
</tr>
</tbody>
</table>

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

$-5 + 5 = 0x85 + 0x5 = 0x8A = -10$ !!!
1's Complement Representation

One’s Complement  
\[-N = \sim(+N)\]

Number range for \(n\) bits: \(-2^{n-1} - 1\) to \(+2^{n-1} - 1\)

<table>
<thead>
<tr>
<th>8-bit range:</th>
<th>16-bit range:</th>
</tr>
</thead>
<tbody>
<tr>
<td>-127 to +127</td>
<td>-32767 to +32767</td>
</tr>
</tbody>
</table>

8-bit encoding examples:

<table>
<thead>
<tr>
<th>8-bit</th>
<th>16-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>+5</td>
<td>000000101</td>
</tr>
<tr>
<td>-5</td>
<td>11111010</td>
</tr>
<tr>
<td>+0</td>
<td>0000000000</td>
</tr>
<tr>
<td>-0</td>
<td>11111111</td>
</tr>
</tbody>
</table>

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
\[-N = \neg(+N) + 1\]
\[= 0 - (N)\]

Number range for \(n\) bits: \(-2^{n-1}\) to \(+2^{n-1} - 1\)

8-bit range:
-128 to +127

16-bit range:
-32768 to +32767

8-bit encoding examples:

\[+5 = 0b00000101 = 0x05\]
\[-5 = 0x00 - 0x05 = 0xFB\]
\[0 = 0b00000000 = 0x00\]
\[-128 = 0 - 128\]
\[= 0x00 - 0x80 = 0x80\]

8-bit Number line

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.

\[ 34 = 2 \times 16 + 2 = \text{0x22} \]
\[ 20 = 1 \times 16 + 4 = \text{0x14} \]

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

+34 as an 8 bit 2’s 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:

\[
0x00 - (+N) = -N
\]

\[
\begin{align*}
0x00 & \quad 0x00 \\
- 0x14 & \\
\hline
0xEC & \quad \text{This is the final result.}
\end{align*}
\]

-20 as an 8-bit 2s complement number is 0xEC
# Signed Decimal to 2’s complement Summary

<table>
<thead>
<tr>
<th>If $n$ is positive</th>
<th>Convert $n$ to hex</th>
<th>+60 = 0x3C</th>
</tr>
</thead>
</table>
| If $n$ is negative | Ignore sign, convert $n$ to hex. Then subtract from zero. | -60 = ??  
 60 = 0x3C | -60 = 0x00 - 0x3C  
  = 0xC4 |

Copyright Thomson/Delmar Learning 2005. All Rights Reserved.
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.

\[
\begin{align*}
0xF0 &= 0b\ 11110000 \quad (\text{MSB is ‘1’}), \ \text{so sign of result is ‘-’} \\
0x64 &= 0b\ 01100100 \quad (\text{MSB is ‘0’}), \ \text{so sign of result is ‘+’}.
\end{align*}
\]

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

\[
\begin{align*}
0x00 \\
- 0xF0 \\
\hline
0x10 &= 16
\end{align*}
\]

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

<table>
<thead>
<tr>
<th>If MSb is 0 (hex digit &lt; 8)</th>
<th>Number is positive, convert to decimal</th>
<th>0x4D = +77</th>
</tr>
</thead>
<tbody>
<tr>
<td>If MSb is 1 (hex digit &gt; 7)</td>
<td>Number is negative, subtract from zero, convert to decimal to find magnitude.</td>
<td>0xB3 = -77</td>
</tr>
<tr>
<td></td>
<td>Combine sign and magnitude</td>
<td>0x00 - 0xB3 = 0x4D</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x4D = 77</td>
</tr>
</tbody>
</table>
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?

\[ +127 = 0x7F \]
\[ + +1 = 0x01 \]

\[ \begin{array}{c} \hline
\text{128} \\
\text{\neq} \\
\text{0x80} \\
\text{(this is actually -128 as a twos complement number!!! - the wrong answer!!!)} \\
\end{array} \]

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:

\[V = C_{\text{MSB}} \oplus C_{\text{MSB-1}}\]

For \(N\) bits, \(C_{\text{MSB}} = \text{Carry out of bit}[N-1]\)
\(C_{\text{MSB-1}} = \text{Carry out of bit}[N-2]\)
Two’s Complement Overflow, Subtraction

Two’s complement overflow for subtraction occurs if:

\[-N - +M = +R \quad (\text{this is just} \quad -N + (-M) = +R, \text{ stated differently})\]
\[ (+N) - (-M) = -R \quad (\text{this is just} \quad +N + (+M) = -R, \text{ stated differently})\]

CANNOT get two’s complement overflow when subtracting numbers of the same signs.
## Some Examples

<table>
<thead>
<tr>
<th>adder logic</th>
<th>unsigned</th>
<th>signed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01</td>
<td>1</td>
<td>+1</td>
</tr>
<tr>
<td>+ 0xFF</td>
<td>+ 255</td>
<td>+ -1</td>
</tr>
<tr>
<td>0x00</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ C = 1, \ Z = 1, \]
\[ V = 0, \ N = 0 \]

<table>
<thead>
<tr>
<th>adder logic</th>
<th>unsigned</th>
<th>signed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFF</td>
<td>255</td>
<td>-1</td>
</tr>
<tr>
<td>+ 0x80</td>
<td>+ 128</td>
<td>+ -128</td>
</tr>
<tr>
<td>0x7F</td>
<td>127</td>
<td>+127</td>
</tr>
</tbody>
</table>

\[ C = 1, \ Z = 0, \]
\[ V = 1, \ N = 0 \]

<table>
<thead>
<tr>
<th>adder logic</th>
<th>unsigned</th>
<th>signed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x80</td>
<td>128</td>
<td>-128</td>
</tr>
<tr>
<td>+ 0x20</td>
<td>+ 32</td>
<td>+ 32</td>
</tr>
<tr>
<td>0xA0</td>
<td>160</td>
<td>-96</td>
</tr>
</tbody>
</table>

\[ C = 0, \ Z = 0, \]
\[ V = 0, \ N = 1 \]
Range Extension (unsigned)

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

Just add zeros to the left.

\[
128 = \text{0x80} \quad \text{(8 bits)} \\
= \text{0x0080} \quad \text{(16 bits)} \\
= \text{0x00000080} \quad \text{(32 bits)}
\]

This is called zero extension; recall that we discussed an instruction called ‘ze’ that zero extended an 8-bit value to a 16-bit value.
Range Extension (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.

\[-128 = 0x80 = 0b 10000000 \text{ (8 bits)}\]
\[= 0xFF80 = 0b 1111111100000000 \text{ (16 bits)}\]
\[= 0xFFFFFFFF80 \text{ (32 bits)}\]

\[+127 = 0x7F = 0b 01111111 \text{ (8 bits)}\]
\[= 0x007F = 0b 0000000011111111 \text{ (16 bits)}\]
\[= 0x0000007F \text{ (32 bits)}\]

This is called SIGN EXTENSION, which simply copies the MSb to the left in the extended range number.
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.

<table>
<thead>
<tr>
<th>Operations that work differently for signed, unsigned</th>
<th>Operations that work the same for unsigned, unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>comparison(&gt;,&gt;=,&lt;,&lt;=), right shift (&gt;&gt;,), multiplication, division</td>
<td>Bitwise logical, addition, subtraction, left shift (&lt;&lt;), equality, inequality.</td>
</tr>
</tbody>
</table>

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 (>>)

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

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

The PIC24 uC has an arithmetic shift right (asr) instruction; it supports the same addressing modes as the logical shift right (lsr) instruction.
C Right Shift versus C Divide by 2

C division in GNU C and Visual Studio always truncates to zero. So:

```c
int8  i8_k;
i8_k = -5;    //this is 0xFB
i8_k = i8_k/2;  //returns –2, which is 0xFE
```

However, the right shift version is:

```c
i8_k = -5;    //this is 0xFB
i8_k = i8_k >> 1;  //returns 0xFD, which is –3!
```

This is not an issue with unsigned numbers. This is why the right shift operation (>>) behavior in ANSI C is compiler dependent; most compilers preserve the sign bit for signed numbers but some always shift in zero. But even preserving the sign bit does not guarantee a match to C division by 2. If you need division by 2, use division by 2. If you need right shift, then use right shift.
Signed Right Shift in PIC24 Assembly

In C
int16 i16_k;
int32 i32_j;

i16_k = i16_k >> 1;
i32_j = i32_j >> 1;

In Assembly

<table>
<thead>
<tr>
<th></th>
<th>In Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>asr i16_k</code></td>
<td><code>asr i32_j+2</code></td>
</tr>
<tr>
<td><code>;arithmetic shift right i16_k</code></td>
<td><code>;arithmetic shift right i32_j.MSW</code></td>
</tr>
<tr>
<td><code>rrc i32_j</code></td>
<td><code>;shift right i32_j.LSW</code></td>
</tr>
</tbody>
</table>

int8, int16, int32 are the C data types used in this book for signed 8-bit, 16-bit, 32-bit integers.

Use the **signed** data types of int16, int32.

Note: The right shift operation (>>>) in ANSI C is *compiler dependent*. The Microchip PIC24 compiler used in this book preserves the sign bit. 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 << 1 == 0x40

no overflow, +64 can be represented in 8 bits

0 0 1 0 0 0 0 0
0x20

+64 * 2 = +128  
0x40 << 1 == 0x80 = -128

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

Multiplied positive number by 2, got a negative number!

0 1 0 0 0 0 0 0
0x40

1 0 0 0 0 0 0 0
0x80
## Signed Comparisons

The table below shows what happens if unsigned comparisons are used for signed numbers in the case of ‘\( \geq \)’. If the numbers have different signs, the comparison gives the wrong result if the C flag is used as the test.

<table>
<thead>
<tr>
<th>Numbers</th>
<th>C flag (k−j)</th>
<th>As Unsigned</th>
<th>k ≥ j</th>
<th>As Signed</th>
<th>k ≥ j</th>
</tr>
</thead>
<tbody>
<tr>
<td>k = 0x7F, j = 0x01</td>
<td>C = 1</td>
<td>k = 127, j = 1</td>
<td>True</td>
<td>k = +127, j = +1</td>
<td>True</td>
</tr>
<tr>
<td>k = 0x80, j = 0xFF</td>
<td>C = 0</td>
<td>k = 128, j = 255</td>
<td>False</td>
<td>k = −128, j = −1</td>
<td>False</td>
</tr>
<tr>
<td>k = 0x80, j = 0x7F</td>
<td>C = 1</td>
<td>k = 128, j = 127</td>
<td>True</td>
<td>k = −128, j = +127</td>
<td>False</td>
</tr>
<tr>
<td>k = 0x01, j = 0xFF</td>
<td>C = 0</td>
<td>k = 1, j = 255</td>
<td>False</td>
<td>k = +1, j = −1</td>
<td>True</td>
</tr>
</tbody>
</table>
**PIC24 Signed Compare**

The PIC24 uC 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

**Flags are for** \( k-j \) **subtraction**

\[
\begin{align*}
&k \geq j \\
&\text{(} N = 0, V = 0 \text{)} \text{ OR } (N = 1, V = 1) \\
&k \leq j \\
&\text{(} N = 1, V = 0 \text{)} \text{ OR } (N = 0, V = 1) \text{ OR } Z=1
\end{align*}
\]
Using N,V flags for Signed Compare

To compare \( k \geq j \), perform \( k-j \) (answer \textit{should be} positive)

After \( k-j \), if \( V = 0 \) (correct result, no overflow)
   if \( N=0 \) (result positive) then \( k \geq j \) (\( V=0,N=0 \))
   else \( N=1 \) (answer negative) so \( k < j \) (\( V=0,N=1 \))

After \( k-j \), if \( V = 1 \) (incorrect result)
   if \( N=1 \) (result negative) then \( k \geq j \) (\( V=1,N=1 \))
   else \( N=0 \) (result positive) so \( k < j \) (\( V=1,N=0 \))

Other flag conditions for tests >, <, <= are seen from previous slide.

Fortunately, the PIC24 uC has some \textit{signed branches} that implement these flag tests in one instruction.
Signed Branches (cont)

Signed branches are used for Signed comparisons and test one or more flags, depending on the comparison

<table>
<thead>
<tr>
<th>Descr:</th>
<th>Syntax</th>
<th>Branch taken when</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch &gt;, signed</td>
<td>BRA GT, label</td>
<td>(~Z &amp; N &amp; OV)</td>
</tr>
<tr>
<td>Branch &gt;=, signed</td>
<td>BRA GE, label</td>
<td>(N &amp; OV)</td>
</tr>
<tr>
<td>Branch &lt;, signed</td>
<td>BRA LT, label</td>
<td>(N &amp; ~OV)</td>
</tr>
<tr>
<td>Branch &lt;=, unsigned</td>
<td>BRA LE, label</td>
<td>(N &amp; ~OV)</td>
</tr>
</tbody>
</table>

Use a Compare instruction to affect the flags before using a signed branch.
PIC24 Signed Compare

In C

```c
int32 k, j;

if (k > j) {
  if-body

... rest of code
```

In Assembly

```assembly
mov j, W0 ; W0 = j.LSW
cp k ; k.LSW - j.LSW
mov j+2, W0 ; W0 = j.MSW
cpb k+2 ; k.MSW - j.MSW
bra LE, end_if ; skip if-body when k<=j

if_body:
  if-body stmt1
  ... stmtN

end_if:
  ... rest of code
```

Used signed branch (LE) because k, j are signed integers.
Mixed Signed 8/16/32-bit Computations

(a) Mixed 8/16-bit operations with signed data

In C

```c
int16 i16_p;
int8 i8_i;
i16_p = i16_p + i8_i;
```

In Assembly

```assembly
mov.b i8_i,WREG ;WREG.lsb = i8_i
se W0,W0 ;sign-extend W0
add i16_p ;i16_p = i16_p + W0
```

Use se to sign-extend W0; W0.MSB will become 0x00 if W0.LSB is positive, else W0.MSB will become 0xFF.

(b) Mixed 16/32-bit operations with signed data

In C

```c
int32 i32_p;
int16 i16_i;
i32_p = i32_p + i16_i;
```

In Assembly

```assembly
mov.i16_i,W0 ;W0 = i16_i
clr W1 ;W1 = 0, assume i16_i is positive
btsc W0,#15 ;test MSbit of W0
setm W1 ;W1 = 0xFFFF, i16_i is negative
add i32_p ;i32_p.LSW = i32_p.LSW + i16_i
mov W1,W0 ;W0 = sign-extension of i16_i
addc i32_p+2 ;i32_p.MSW = i32_p.MSW + SE of i16_i
```

Load W1 with 0x0000 if i16_i is positive, else load W1 with 0xFFFF. Register W1 becomes the MSW of i16_i, and this is added to i32_p.MSW.
branch versus goto

Recall that a goto used two instruction words which encoded a 23-bit value that is loaded directly into the PC. A branch takes 1 instruction word, whose format is:

\[
\text{bra } \langle\text{cond}\rangle,\langle\text{BTA}\rangle
\]

BTA is branch target address

\[
\text{BTA} = (\text{PC}_{\text{old}} + 2) + (2 \times \#\text{Slit16}) \rightarrow \text{PC}_{\text{new}}
\]

where \(\text{PC}_{\text{old}}\) is branch instruction address. \#\text{Slit16} computed as:

\[
\#\text{Slit16} = [\text{BTA} - (\text{PC}_{\text{old}} + 2)]/2
\]

\((\text{PC}_{\text{old}} + 2)\) because PC is already incremented when the new PC value is computed.

<table>
<thead>
<tr>
<th>Branch</th>
<th>bbbb</th>
</tr>
</thead>
<tbody>
<tr>
<td>bra OV</td>
<td>0000</td>
</tr>
<tr>
<td>bra C/</td>
<td>0001</td>
</tr>
<tr>
<td>bra GEU</td>
<td>0010</td>
</tr>
<tr>
<td>bra Z</td>
<td>0011</td>
</tr>
<tr>
<td>bra N</td>
<td>0100</td>
</tr>
<tr>
<td>bra LE</td>
<td>0101</td>
</tr>
<tr>
<td>bra LT</td>
<td>0110</td>
</tr>
<tr>
<td>bra LEU</td>
<td>0111</td>
</tr>
<tr>
<td>bra unconditional</td>
<td>1110</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Branch</th>
<th>bbbb</th>
</tr>
</thead>
<tbody>
<tr>
<td>bra NOV</td>
<td>1000</td>
</tr>
<tr>
<td>bra NC/</td>
<td>1001</td>
</tr>
<tr>
<td>bra LTU</td>
<td>1010</td>
</tr>
<tr>
<td>bra NZ</td>
<td>1011</td>
</tr>
<tr>
<td>bra NN</td>
<td>1100</td>
</tr>
<tr>
<td>bra GT</td>
<td>1101</td>
</tr>
<tr>
<td>bra GE</td>
<td>1110</td>
</tr>
<tr>
<td>bra GTU</td>
<td>1110</td>
</tr>
</tbody>
</table>
### branch Machine Code example

<table>
<thead>
<tr>
<th>location</th>
<th>machine code</th>
<th>instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>(hex)</td>
<td>(hex)</td>
<td></td>
</tr>
<tr>
<td>0200</td>
<td>????</td>
<td>top_loop:</td>
</tr>
<tr>
<td>0202</td>
<td>????</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0210</td>
<td>????</td>
<td></td>
</tr>
<tr>
<td>0212</td>
<td>3DFFFF6</td>
<td>bra GE, top_loop ; branch to top</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{BTA} &= (\text{PC}_{\text{old}} + 2) + (2 \ast \#\text{Slit16}) \\
\#\text{Slit16} &= [\text{BTA} - (\text{PC}_{\text{old}} + 2)]/2 \\
&= (0x200 - (0x0212 + 2))/2 \\
&= 0xFFEC/2 = 0xFFEC >> 1 \\
&= 0xFFF6 = -10 \text{ (16-bit signed displacement encoded as \#Slit16)}
\end{align*}
\]

\[
<\text{cond}> = \text{GE} \downarrow \quad \#\text{Slit16}
\]

bra GE, top_loop \[\rightarrow\] bra -10 \[\rightarrow\] \[0011\ | \ 1101\ | \ 1111\ | \ 1111\ | \ 1111\ | \ 0110\] \[\rightarrow\] 0x3DFFFF6
branch, goto Pros/Cons

The branch instructions use a 16-bit offset. This means the target address must be within –32768 to +32767 instruction words from the branch.

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

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

The advantage of a branch is that it only takes one instruction word.
What do you need to know?

• 32-bit operands and usage in
  – bitwise logical
  – addition/subtraction
  – increment/decrement
  – shift left, shift right
  – Zero/non-zero testing
  – Operand comparison for branch operations

• Signed data
  – 2’s complement definition, conversion hex/decimal, decimal/hex
  – Signed right shift
  – N, V flags and their usage in signed branches
  – Sign extension, mixed 8/16/32 bit operations on signed variables