Computer Systems
CSE 410 Spring 2012
3 - Integers
Today’s Topics

- Representation of integers: unsigned and signed
- Casting
- Arithmetic and shifting
- Sign extension

- Reading: Bryant/O’Hallaron sec. 2.2-2.3
Encoding Integers

- The hardware (and C) supports two flavors of integers:
  - unsigned – only the non-negatives
  - signed – both negatives and non-negatives

- There are only $2^W$ distinct bit patterns of $W$ bits, so...
  - Can't represent all the integers
  - Unsigned values are $0 \ldots 2^{W-1}$
  - Signed values are $-2^{W-1} \ldots 2^{W-1}-1$
Unsigned Integers

- Unsigned values are just what you expect
  - $b_7b_6b_5b_4b_3b_2b_1b_0 = b_72^7 + b_62^6 + b_52^5 + ... + b_12^1 + b_02^0$
  - Interesting aside: $1+2+4+8+...+2^{N-1} = 2^N - 1$

- You add/subtract them using the normal “carry/borrow” rules, just in binary

- An important use of unsigned integers in C is pointers
  - There are no negative memory addresses
Signed Integers

- Let's do the natural thing for the positives
  - They correspond to the unsigned integers of the same value
    - Example (8 bits): 0x00 = 0, 0x01 = 1, ..., 0x7F = 127

- But, we need to let about half of them be negative
  - Use the high order bit to indicate 'negative'
  - Call it “the sign bit”
  - Examples (8 bits):
    - 0x00 = 00000000₂ is non-negative, because the sign bit is 0
    - 0x7F = 01111111₂ is non-negative
    - 0x80 = 10000000₂ is negative
Sign-and-Magnitude Negatives

How should we represent -1 in binary?

- Possibility 1: 10000001₂
  Use the MSB for “+ or -”, and the other bits to give magnitude
Sign-and-Magnitude Negatives

How should we represent -1 in binary?

- Possibility 1: 10000001₂
  Use the MSB for “+ or -”, and the other bits to give magnitude
  (Unfortunate side effect: there are two representations of 0!)
Sign-and-Magnitude Negatives

- How should we represent -1 in binary?
  - Possibility 1: \(10000001_2\)
    Use the MSB for “+ or -”, and the other bits to give magnitude
  - Another problem: math is cumbersome
    - \(4 - 3 \neq 4 + (-3)\)
Ones’ Complement Negatives

How should we represent -1 in binary?
- Possibility 2: 11111110₂

Negative numbers: bitwise complements of positive numbers
It would be handy if we could use the same hardware adder to add signed integers as unsigned.
Ones’ Complement Negatives

- How should we represent -1 in binary?
  - Possibility 2: $11111110_2$
    - Negative numbers: bitwise complements of positive numbers
    - Solves the arithmetic problem

<table>
<thead>
<tr>
<th>Add</th>
<th>Invert, add, add carry</th>
<th>Invert and add</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0100</td>
<td>-4 1011</td>
</tr>
<tr>
<td>+3</td>
<td>+0011</td>
<td>+3 +0011</td>
</tr>
<tr>
<td>=7</td>
<td>=0111</td>
<td></td>
</tr>
</tbody>
</table>

- 1 10000
- add carry: +1
- = 0001

end-around carry
Ones’ Complement Negatives

How should we represent -1 in binary?

- Possibility 2: 11111110_2
  Negative numbers: bitwise complements of positive numbers
  Use the same hardware adder to add signed integers as unsigned (but we have to keep track of the end-around carry bit)

- Why does it work?
- The ones’ complement of a 4-bit positive number y is 1111_2 – y
  - 0111 \equiv 7_{10}
  - 1111_2 – 0111_2 = 1000_2 \equiv -7_{10}
  - 1111_2 is 1 less than 10000_2 = 2^4 – 1
    - –y is represented by (2^4 – 1) – y
Ones’ Complement Negatives

How should we represent -1 in binary?

- Possibility 2: 11111110₂
  Negative numbers: bitwise complements of positive numbers
  (But there are still two representations of 0!)
Two's Complement Negatives

- How should we represent -1 in binary?
  - Possibility 3: $11111111_2$
    Bitwise complement plus one
    (Only one zero)
Two's Complement Negatives

How should we represent -1 in binary?

- Possibility 3: $11111112$
  Bitwise complement plus one
  (Only one zero)

- Simplifies arithmetic
  Use the same hardware adder to add signed integers as unsigned
  (simple addition; discard the highest carry bit)

<table>
<thead>
<tr>
<th>Add</th>
<th>Invert and add</th>
<th>Invert and add</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 0100</td>
<td>4 0100</td>
<td>-4 1100</td>
</tr>
<tr>
<td>+ 3 + 0011</td>
<td>-3 + 1101</td>
<td>+ 3 + 0011</td>
</tr>
<tr>
<td>= 7 0111</td>
<td>= 1 0001</td>
<td>= 1 0001</td>
</tr>
<tr>
<td>drop carry</td>
<td></td>
<td>-1 1111</td>
</tr>
</tbody>
</table>
Two's Complement Negatives

How should we represent -1 in binary?
- Two’s complement: Bitwise complement plus one

Why does it work?
- Recall: The ones’ complement of a b-bit positive number y is \((2^b - 1) - y\)
- Two’s complement adds one to the bitwise complement, thus, \(-y = 2^b - y\)
  - \(-y\) and \(2^b - y\) are equal mod \(2^b\)
    - (have the same remainder when divided by \(2^b\))
  - Ignoring carries is equivalent to doing arithmetic mod \(2^b\)
Two's Complement Negatives

How should we represent -1 in binary?
- Two’s complement: Bitwise complement plus one

What should the 8-bit representation of -1 be?

00000001
+???????? (want whichever bit string gives right result)
00000000

00000010  00000011
+????????  +????????
00000000  00000000
Unsigned & Signed Numeric Values

<table>
<thead>
<tr>
<th>X</th>
<th>Unsigned</th>
<th>Signed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>-8</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
<td>-7</td>
</tr>
<tr>
<td>1010</td>
<td>10</td>
<td>-6</td>
</tr>
<tr>
<td>1011</td>
<td>11</td>
<td>-5</td>
</tr>
<tr>
<td>1100</td>
<td>12</td>
<td>-4</td>
</tr>
<tr>
<td>1101</td>
<td>13</td>
<td>-3</td>
</tr>
<tr>
<td>1110</td>
<td>14</td>
<td>-2</td>
</tr>
<tr>
<td>1111</td>
<td>15</td>
<td>-1</td>
</tr>
</tbody>
</table>

- Both signed and unsigned integers have limits
  - If you compute a number that is too big, you wrap: \( 6 + 4 = ? \) \( 15U + 2U = ? \)
  - If you compute a number that is too small, you wrap: \(-7 - 3 = ?\) \( 0U - 2U = ? \)
  - Answers are only correct mod \( 2^b \)
- The CPU may be capable of “throwing an exception” for overflow on signed values
  - It won't for unsigned
- But C and Java just cruise along silently when overflow occurs...
# Mapping Signed ↔ Unsigned

<table>
<thead>
<tr>
<th>Bits</th>
<th>Signed</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>-8</td>
<td>8</td>
</tr>
<tr>
<td>1001</td>
<td>-7</td>
<td>9</td>
</tr>
<tr>
<td>1010</td>
<td>-6</td>
<td>10</td>
</tr>
<tr>
<td>1011</td>
<td>-5</td>
<td>11</td>
</tr>
<tr>
<td>1100</td>
<td>-4</td>
<td>12</td>
</tr>
<tr>
<td>1101</td>
<td>-3</td>
<td>13</td>
</tr>
<tr>
<td>1110</td>
<td>-2</td>
<td>14</td>
</tr>
<tr>
<td>1111</td>
<td>-1</td>
<td>15</td>
</tr>
</tbody>
</table>

Note: For each bit pattern, the signed value is mapped to its corresponding unsigned value. The mapping is bilateral, with each signed value having a corresponding unsigned value and vice versa.
Numeric Ranges

- **Unsigned Values**
  - $\text{UMin} = 0$
    - 000...0
  - $\text{UMax} = 2^w - 1$
    - 111...1

- **Two’s Complement Values**
  - $\text{TMin} = -2^{w-1}$
    - 100...0
  - $\text{TMax} = 2^{w-1} - 1$
    - 011...1

- **Other Values**
  - Minus 1
    - 111...1 $0xFFFFFFFF$ (32 bits)

**Values for $W = 16$**

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{UMax}$</td>
<td>65535</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>$\text{TMax}$</td>
<td>32767</td>
<td>7F FF</td>
<td>01111111 11111111</td>
</tr>
<tr>
<td>$\text{TMin}$</td>
<td>-32768</td>
<td>80 00</td>
<td>10000000 00000000</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>
# Values for Different Word Sizes

<table>
<thead>
<tr>
<th></th>
<th>W</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>255</td>
<td>65,535</td>
<td>4,294,967,295</td>
<td>18,446,744,073,709,551,615</td>
<td></td>
</tr>
<tr>
<td>Tmax</td>
<td>127</td>
<td>32,767</td>
<td>2,147,483,647</td>
<td>9,223,372,036,854,775,807</td>
<td></td>
</tr>
<tr>
<td>Tmin</td>
<td>-128</td>
<td>-32,768</td>
<td>-2,147,483,648</td>
<td>-9,223,372,036,854,775,808</td>
<td></td>
</tr>
</tbody>
</table>

## Observations
- \(|TMin| = TMax + 1\)
  - Asymmetric range
- \(UMax = 2 \times TMax + 1\)

## C Programming
- `#include <limits.h>`
- Declares constants, e.g.,
  - `ULONG_MAX`
  - `LONG_MAX`
  - `LONG_MIN`
- Values platform specific
Conversion Visualized

- **2’s Comp. → Unsigned**
  - Ordering Inversion
  - Negative → Big Positive

2’s Complement Range

Unsigned Range

- $T_{Max}$
- $T_{Max} + 1$
- $T_{Max}$
- $UMax$
- $UMax - 1$
- 0
- -1
- -2
- $T_{Min}$
Signed vs. Unsigned in C

- **Constants**
  - By default are considered to be signed integers
  - Unsigned if have “U” as suffix
    - 0U, 4294967259U

- **Casting**
  - int tx, ty;
    - unsigned ux, uy;
  - Explicit casting between signed & unsigned same as U2T and T2U
    - tx = (int) ux;
    - uy = (unsigned) ty;
  - Implicit casting also occurs via assignments and procedure calls
    - tx = ux;
    - uy = ty;
Casting Surprises

Expression Evaluation

- If you mix unsigned and signed in a single expression, then **signed values implicitly cast to unsigned**
- Including comparison operations `<`, `>`, `==`, `<=`, `>=`
- Examples for $W = 32$: $TMIN = -2,147,483,648$ \hspace{1cm} $TMAX = 2,147,483,647$

<table>
<thead>
<tr>
<th>Constant$_1$</th>
<th>Constant$_2$</th>
<th>Relation</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0U</td>
<td><code>==</code></td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td><code>&lt;</code></td>
<td>signed</td>
</tr>
<tr>
<td>-1</td>
<td>0U</td>
<td><code>&gt;</code></td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>-2147483647-1</td>
<td><code>&gt;</code></td>
<td>signed</td>
</tr>
<tr>
<td>2147483647U</td>
<td>-2147483647-1</td>
<td><code>&lt;</code></td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>-2</td>
<td><code>&gt;</code></td>
<td>signed</td>
</tr>
<tr>
<td>(unsigned)-1</td>
<td>-2</td>
<td><code>&gt;</code></td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>2147483648U</td>
<td><code>&lt;</code></td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>(int) 2147483648U</td>
<td><code>&gt;</code></td>
<td>signed</td>
</tr>
</tbody>
</table>
Shift Operations

- **Left shift:** \( x << y \)
  - Shift bit-vector \( x \) left by \( y \) positions
    - Throw away extra bits on left
    - Fill with 0s on right
  - Multiply by \( 2^{**y} \)

- **Right shift:** \( x >> y \)
  - Shift bit-vector \( x \) right by \( y \) positions
    - Throw away extra bits on right
  - Logical shift (for unsigned)
    - Fill with 0s on left
  - Arithmetic shift (for signed)
    - Replicate most significant bit on right
    - Maintain sign of \( x \)
  - Divide by \( 2^{**y} \)
  - Correct truncation (towards 0) requires some care with signed numbers

<table>
<thead>
<tr>
<th>Argument x</th>
<th>01100010</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;&lt; 3)</td>
<td>00010000</td>
</tr>
<tr>
<td>Logical (&gt;&gt; 2)</td>
<td>00011000</td>
</tr>
<tr>
<td>Arithmetic (&gt;&gt; 2)</td>
<td>00011000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Argument x</th>
<th>10100010</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;&lt; 3)</td>
<td>00010000</td>
</tr>
<tr>
<td>Logical (&gt;&gt; 2)</td>
<td>00101000</td>
</tr>
<tr>
<td>Arithmetic (&gt;&gt; 2)</td>
<td>11101000</td>
</tr>
</tbody>
</table>

*Undefined behavior when \( y < 0 \) or \( y \geq \text{word\_size} \)*
Using Shifts and Masks

- **Extract 2nd most significant byte of an integer**
  - First shift: \( x >> (2 \times 8) \)
  - Then mask: \( ( x >> 16 ) & 0xFF \)

<table>
<thead>
<tr>
<th>( x )</th>
<th>01100001 01100010 01100011 01100100</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x &gt;&gt; 16 )</td>
<td>00000000 00000000 01100001 01100010</td>
</tr>
<tr>
<td>( ( x &gt;&gt; 16 ) &amp; 0xFF )</td>
<td>00000000 00000000 00000000 11111111</td>
</tr>
</tbody>
</table>

- **Extracting the sign bit**
  - \( ( x >> 31 ) & 1 \) - need the “& 1” to clear out all other bits except LSB

- **Conditionals as Boolean expressions (assuming \( x \) is 0 or 1)**
  - if \((x)\) \(a=y\) else \(a=z\); which is the same as \( a = x ? y : z; \)
  - Can be re-written as: \( a = ( (x << 31) >> 31) & y + (!x << 31 ) >> 31 ) & z; \)
Sign Extension

- **Task:**
  - Given $w$-bit signed integer $x$
  - Convert it to $w+k$-bit integer with same value

- **Rule:**
  - Make $k$ copies of sign bit:
  - $X' = x_{w-1}, ..., x_{w-1}, x_{w-1}, x_{w-2}, ..., x_0$

$k$ copies of MSB

---

\[ \overbrace{x_{w-1}, ..., x_{w-1}, x_{w-1}, x_{w-2}, ..., x_0}^{k \text{ copies of MSB}} \]
Sign Extension Example

- Converting from smaller to larger integer data type
- C automatically performs sign extension

```c
short int x = 12345;
int     ix = (int) x;
short int y = -12345;
int     iy = (int) y;
```

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>12345</td>
<td>30 39</td>
<td>00110000 01101101</td>
</tr>
<tr>
<td>ix</td>
<td>12345</td>
<td>00 00 30 39</td>
<td>00000000 00000000 00110000 01101101</td>
</tr>
<tr>
<td>y</td>
<td>-12345</td>
<td>CF C7</td>
<td>11001111 11000111</td>
</tr>
<tr>
<td>iy</td>
<td>-12345</td>
<td>FF FF CF C7</td>
<td>11111111 11111111 11001111 11000111</td>
</tr>
</tbody>
</table>