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 $0$ ... $2^W-1$
  - Signed values are $-2^{W-1}$ ... $2^{W-1}-1$
Unsigned Integers

• Unsigned values are just what you expect
  \[ b_7b_6b_5b_4b_3b_2b_1b_0 = b_72^7 + b_62^6 + b_52^5 + \ldots + b_12^1 + b_02^0 \]

  Interesting aside: \[ 1+2+4+8+\ldots+2^{N-1} = 2^N - 1 \]

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

• unsigned integers in C are not the same thing as pointers
  • Similar: There are no negative memory addresses
  • Similar: Years ago sizeof(int) = sizeof(int *)
  • Not Similar: Today and in well written code for all time, sizeof(int) != sizeof(int *)
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 something like 'negative'
  - Historically, there have been 3 flavors in use... but today there is only 1 (and for good reason).
    - Bad ideas (but were commonly used in the past!)
      - sign/magnitude
      - one’s complement
    - Good idea:
      - Two's complement
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
Sign-and-Magnitude Negatives

How should we represent -1 in binary?

- **Possibility 1**: \(10000001\)
  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: 11111110

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></td>
</tr>
<tr>
<td>+ 3</td>
<td>+ 0011</td>
<td></td>
</tr>
<tr>
<td>= 7</td>
<td>= 0111</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>(-4) 1011</td>
</tr>
<tr>
<td>- 3</td>
<td>+ 1100</td>
<td>(+3) + 0011</td>
</tr>
<tr>
<td>= 1</td>
<td>1 0000</td>
<td>(-1) 1110</td>
</tr>
<tr>
<td>add carry:</td>
<td>+1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>= 0001</td>
<td></td>
</tr>
</tbody>
</table>

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 $10000_2 = 2^4 - 1$
Ones’ Complement Negatives

- How should we represent -1 in binary?
  - **Possibility 2:** \(11111110_2\)
  
  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:** 11111111<sub>2</sub>
    Bitwise complement plus one
    (Only one zero)
  - **Simplifies arithmetic**
    Use the same hardware adder to add signed integers as unsigned (simple addition; discard

<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 1 0001</td>
<td>– 1 1111</td>
</tr>
<tr>
<td></td>
<td>drop carry = 0001</td>
<td></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, \(-y\) is \(2^b - y\) (or \(-x == \sim x + 1\))

  • \(-y\) 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?

\[
\begin{align*}
00000001 & +?????????? \quad \text{(want whichever bit string gives right result)} \\
00000000 &
\end{align*}
\]

\[
\begin{align*}
00000010 & \quad 00000011 \\
+?????????? & \quad +?????????? \\
00000000 & \quad 00000000
\end{align*}
\]
Unsigned & Signed Numeric Values

- Both signed and unsigned integers have limits
- If you compute a number that is too big, you wrap: \(6 + 4 = ? \quad 15U + 2U = ?\)
- If you compute a number that is too small, you wrap: \(-7 - 3 = ? \quad 0U - 2U = ?\)
- 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 $\leftrightarrow$ 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>
Numeric Ranges

- **Unsigned Values**
  - \( U_{\text{Min}} = 0 \)
    - 000...0
  - \( U_{\text{Max}} = 2^w - 1 \)
    - 111...1

Two’s Complement Values

- \( T_{\text{Min}} = -2^{w-1} \)
  - 100...0
- \( T_{\text{Max}} = 2^{w-1} - 1 \)
  - 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>UMax</td>
<td>65535</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>TMax</td>
<td>32767</td>
<td>7F FF</td>
<td>01111111 11111111</td>
</tr>
<tr>
<td>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></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td></td>
</tr>
<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 * 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

TMax

0

-1

-2

TMin

UMax

UMax – 1

TMax + 1

TMax

Unsigned Range
Signed vs. Unsigned in C

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

- Size can be typed too 1234567890123456ULL

- **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 mix unsigned and signed in single expression, signed values implicitly cast to unsigned

Including comparison operations <, >, ==, <=, >=

Examples for W = 32: TMIN = -2,147,483,648  TMAX = 2,147,483,647

<table>
<thead>
<tr>
<th>Constant\textsubscript{1}</th>
<th>Constant\textsubscript{2}</th>
<th>Relation</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0U</td>
<td>(\leq)</td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>&lt;</td>
<td>signed</td>
</tr>
<tr>
<td>-1</td>
<td>0U</td>
<td>&gt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>2,147,483,647</td>
<td>-2,147,483,647-1</td>
<td>&lt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>2,147,483,647U</td>
<td>-2,147,483,647-1</td>
<td>&gt;</td>
<td>signed</td>
</tr>
<tr>
<td>-1</td>
<td>0U</td>
<td>&gt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>(unsigned)-1</td>
<td>-2</td>
<td>&lt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>2,147,483,647</td>
<td>2,147,483,648U</td>
<td>&gt;</td>
<td>signed</td>
</tr>
<tr>
<td>2,147,483,647</td>
<td>(int) 2,147,483,648U</td>
<td>&gt;</td>
<td>signed</td>
</tr>
</tbody>
</table>
General advice on types

- Be as explicit as possible
  
  typedef unsigned int uint32_t;
  
  uint32_t i; for(i = 0; i < n; i++) { ... }

- Use modern C dialect features / use the type system to catch errors at compile time:
  
  // fast and loose
  
  #define my_constant 1234
  
  // better
  
  #define my_constant 1234U
  
  // generally (but not always) best
  
  const unsigned int my_constant = 1234;

- Use opaque types as much as possible
  
  struct my_type; struct my_type *allocate_object_of_my_type();

- C compilers have a lot of legacy cruft in this area. Much can go wrong...
  e.g. is unsigned long long x:4; a 4 bit field of a 64 bit type? or a 32 bit one?
Shift Operations

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

**Right shift:** \( \text{x} \gg \text{y} \)
- Shift bit-vector \( \text{x} \) right by \( \text{y} \) 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 \( \text{x} \)
- Divide by \( 2^{\text{y}} \)
- 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>( \ll 3 )</td>
<td>00010000</td>
</tr>
<tr>
<td>Logical ( \gg 2 )</td>
<td>00011000</td>
</tr>
<tr>
<td>Arithmetic ( \gg 2 )</td>
<td>00011000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Argument x</th>
<th>10100010</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \ll 3 )</td>
<td>00010000</td>
</tr>
<tr>
<td>Logical ( \gg 2 )</td>
<td>00101000</td>
</tr>
<tr>
<td>Arithmetic ( \gg 2 )</td>
<td>11101000</td>
</tr>
</tbody>
</table>

What if \( \text{y} < 0 \) or \( \text{y} \geq \text{word_size} \)?
Using Shifts and Masks

Extract 2\textsuperscript{nd} most significant byte of an integer

First shift: \( x \gg (2 \times 8) \)
Then mask: \((x \gg 16) \& 0xFF\)

<table>
<thead>
<tr>
<th>x</th>
<th>01100001 0110010 01100011 01100100</th>
</tr>
</thead>
<tbody>
<tr>
<td>x &gt;&gt; 16</td>
<td>00000000 00000000 01100001 0110010</td>
</tr>
<tr>
<td>(x &gt;&gt; 16) &amp; 0xFF</td>
<td>00000000 00000000 00000000 11111111</td>
</tr>
<tr>
<td></td>
<td>00000000 00000000 00000000 0110010</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 here)

\[
\text{if } (x) \text{ a=y else a=z; \ which is the same as } a = x ? y : z;
\]
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
Sign Extension Example

```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>

Converting from smaller to larger integer data type

C automatically performs sign extension

You might have to if converting a bizarre data type to a native one (e.g. PMC counters are sometimes 48 bits)