Today’s Topics

- Strings
- Boolean algebra
- Representation of integers: unsigned and signed
- Casting
- Arithmetic and shifting
- Sign extension
Quick review…

**x at location 0x04, y at 0x18**

```c
int * x; int y;
    x = &y + 3; // get address of y add 12

int * x; int y;
    *x = y; // value of y to location x point
```

```
+-----------------+----------+----------+----------+----------+
|                  | 0000     | 0004     | 0008     | 000C     |
|                  | 0010     | 0014     | 0018     | 001C     |
|                  | 0020     | 0024     |          |          |
|                  | AA       | BB       | CC       | DD       |
+-----------------+----------+----------+----------+----------+
```
Representing strings?
Representing strings

A C-style string is represented by an array of bytes.

- Elements are one-byte ASCII codes for each character.
- A 0 value marks the end of the array.
Null-terminated Strings

For example, “Harry Potter” can be stored as a 13-byte array.

<table>
<thead>
<tr>
<th>72</th>
<th>97</th>
<th>114</th>
<th>114</th>
<th>121</th>
<th>32</th>
<th>80</th>
<th>111</th>
<th>116</th>
<th>116</th>
<th>101</th>
<th>114</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>a</td>
<td>r</td>
<td>y</td>
<td>P</td>
<td>o</td>
<td>t</td>
<td>t</td>
<td>e</td>
<td>r</td>
<td>\0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Why do we put a a 0, or null, at the end of the string?

Computing string length?
Compatibility

char S[6] = "12345";

<table>
<thead>
<tr>
<th>Linux/Alpha S</th>
<th>Sun S</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>33</td>
<td>33</td>
</tr>
<tr>
<td>34</td>
<td>34</td>
</tr>
<tr>
<td>35</td>
<td>35</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>

Byte ordering not an issue

Unicode characters – up to 4 bytes/character

ASCII codes still work (leading 0 bit) but can support the many characters in all languages in the world

Java and C have libraries for Unicode (Java commonly uses 2 bytes/char)
Boolean Algebra

Developed by George Boole in 19th Century

Algebraic representation of logic

Encode “True” as 1 and “False” as 0

AND: \( A \& B = 1 \) when both \( A \) is 1 and \( B \) is 1

OR: \( A \| B = 1 \) when either \( A \) is 1 or \( B \) is 1

XOR: \( A^{\wedge}B = 1 \) when either \( A \) is 1 or \( B \) is 1, but not both

NOT: \( \sim A = 1 \) when \( A \) is 0 and vice-versa

DeMorgan’s Law: \( \sim(A \| B) = \sim A \& \sim B \)

<table>
<thead>
<tr>
<th>&amp;</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

| | 0 | 1 | 1 |
|---|---|---|
| 0 | 0 | 1 | 1 |

<table>
<thead>
<tr>
<th>^</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>~</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
General Boolean Algebras

**Operate on bit vectors**

Operations applied bitwise

\[
\begin{align*}
01101001 & \quad 01101001 & \quad 01101001 \\
\& 01010101 & \quad \upharpoonright 01010101 & \quad ^\wedge 01010101 & \quad ^\sim 01010101
\end{align*}
\]

**All of the properties of Boolean algebra apply**

\[
\begin{align*}
01010101 \\
^\wedge 01010101
\end{align*}
\]

**How does this relate to set operations?**
Representing & Manipulating Sets

Representation

Width $w$ bit vector represents subsets of $\{0, \ldots, w-1\}$

$a_j = 1$ if $j \in A$

01101001 \{ 0, 3, 5, 6 \}

76543210

01010101 \{ 0, 2, 4, 6 \}

76543210

Operations

& \quad \text{Intersection} \quad 01000001 \quad \{ 0, 6 \}

| \quad \text{Union} \quad 01111101 \quad \{ 0, 2, 3, 4, 5, 6 \}

^ \quad \text{Symmetric difference} \quad 00111100 \quad \{ 2, 3, 4, 5 \}

~ \quad \text{Complement} \quad 10101010 \quad \{ 1, 3, 5, 7 \}
Bit-Level Operations in C

**Operations &, |, ^, ~ are available in C**

Apply to any “integral” data type

- long, int, short, char, unsigned

View arguments as bit vectors

Arguments applied bit-wise

**Examples (char data type)**

~0x41 --> 0xBE

~01000001₂ → 10111110₂

~0x00 --> 0xFF

~00000000₂ → 11111111₂

0x69 & 0x55 --> 0x41

01101001₂ & 01010110₂ → 0100001₁₀₂

0x69 | 0x55 --> 0x7D

01101001₂ | 01010110₂ → 01111101₂
Contrast: Logic Operations in C

Contrast to logical operators

&&, ||, !

View 0 as "False"
Anything nonzero as "True"
Always return 0 or 1
Early termination

Examples (char data type)

!0x41  -->  0x00
!0x00  -->  0x01
!!0x41 -->  0x01

0x69 && 0x55  -->  0x01
0x69 || 0x55  -->  0x01
p && *p++      (avoids null pointer access, null pointer = 0x00000000)
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+...+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: $10000001_2$
    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: \(10000001_2\)
    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_2
    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>= 1 10000</td>
</tr>
<tr>
<td></td>
<td>add carry: +1</td>
<td>-1 1110</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 \(10000_2 = 2^4 - 1\)
  - \(-y\) is represented by \((2^4 - 1) - y\)
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: $11111111_2$
    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</td>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>+ 3</td>
<td>+ 0011</td>
<td>-3</td>
</tr>
<tr>
<td>= 7</td>
<td>= 0111</td>
<td>= 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>drop carry</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 $2^b - y$
  - $-y$ 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
+????????
00000000
```

* (want whichever bit string gives right result)
Unsigned & Signed Numeric Values

- Both signed and unsigned integers have limits
  - If you compute a number that is too big, you wrap: \(6 + 4 = ?\) \(15_{U} + 2_{U} = ?\)
  - If you compute a number that is too small, you wrap: \(-7 - 3 = ?\) \(0_{U} - 2_{U} = ?\)
  - 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...

<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>
# 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>
Numeric Ranges

Unsigned Values

- $UMin = 0$
  
  000...0

- $UMax = 2^w - 1$
  
  111...1

Two’s Complement Values

- $TMin = -2^{w-1}$
  
  100...0

- $TMax = 2^{w-1} - 1$
  
  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>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>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>18,446,744,073,709,551,619</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>
</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>
</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
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 mix unsigned and signed in single expression, 

*signed values implicitly cast to unsigned*

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

Examples for $W = 32$:  

<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>&lt;</td>
<td>signed</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>&gt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>0U</td>
<td>&gt;</td>
<td>signed</td>
</tr>
<tr>
<td>2147483647</td>
<td>-2147483647-1</td>
<td>&lt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647U</td>
<td>-2147483647-1</td>
<td>&gt;</td>
<td>signed</td>
</tr>
<tr>
<td>-1</td>
<td>-2</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>2147483647</td>
<td>2147483648U</td>
<td>&gt;</td>
<td>signed</td>
</tr>
<tr>
<td>2147483647</td>
<td>(int) 2147483648U</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TMIN = -2,147,483,648  
TMAX = 2,147,483,647
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 \( 2^y \)

**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 \( 2^y \)
- *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>

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

Extract 2\textsuperscript{nd} 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</th>
<th>01100010</th>
<th>01100011</th>
<th>01100100</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x &gt;&gt; 16 )</td>
<td>00000000</td>
<td>00000000</td>
<td>01100001</td>
<td>01100010</td>
</tr>
<tr>
<td>(( x &gt;&gt; 16 ) &amp; 0xFF)</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>11111111</td>
</tr>
<tr>
<td>( 00000000 00000000 00000000 01100010 )</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>01100010</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)

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:

$$X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0$$

$k$ 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