Bits, Bytes, and Integers – Part 2

15-213: Introduction to Computer Systems
3rd Lecture, Jan. 19, 2016

Instructors:
Franz Franchetti, Seth Copen Goldstein, Ralf Brown, and Brian Railing
Autolab accounts

- You should have an autolab account by now
- You must be enrolled to get an account
  - Autolab is not tied in to the Hub’s rosters
  - If you do NOT have an Autolab account for 213/513 this semester, please add your name to the following Google form. The link is available from the course web page.
    
    https://docs.google.com/forms/d/1M3dHRvEraM8eCpk9jq46rkqDqeEho_ffhdce7F25rqY/viewform?usp=send_form
    
    We will update the autolab accounts once a day, so check back in 24 hours.
First Assignment: Data Lab

- Due: Thursday, Jan 28th 2016, 11:59:00 pm

- Last Possible Time to Turn in: Fri, Jan 29, 11:59PM

- Read the instructions carefully

- You should have started

- Seek help (office hours started on Sunday)

- Based on Lecture 2, 3, and 4

- After today’s lecture you know everything for the integer problems, float problems covered on Thursday
Summary From Last Lecture

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
- Representations in memory, pointers, strings
- Summary
Bit-Level Operations in C

- **Operations &,, |,, ~,, ^ 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)**
  - \(~0x41 \rightarrow 0xBE\)
    - \(~0100 \ 0001_2 \rightarrow 1011 \ 1110_2\)
  - \(~0x00 \rightarrow 0xFF\)
    - \(~0000 \ 0000_2 \rightarrow 1111 \ 1111_2\)
  - \(0x69 \& \ 0x55 \rightarrow 0x41\)
    - \(0110 \ 1001_2 \& \ 0101 \ 0101_2 \rightarrow 0100 \ 0001_2\)
  - \(0x69 \ | \ 0x55 \rightarrow 0x7D\)
    - \(0110 \ 1001_2 \ | \ 0101 \ 0101_2 \rightarrow 0111 \ 1101_2\)
Logic Operations in C

- Logic Operations: &&, ||, !
  - 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)
Unsigned & Signed Numeric Values

- **Equivalence**
  - Same encodings for nonnegative values

- **Uniqueness**
  - Every bit pattern represents unique integer value
  - Each representable integer has unique bit encoding

- **Expression containing signed and unsigned int:**
  - `int` is cast to `unsigned`!!

<table>
<thead>
<tr>
<th>X</th>
<th>B2U(X)</th>
<th>B2T(X)</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>
Sign Extension and Truncation

- **Sign Extension**

- **Truncation**
Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations

Integers
- Representation: unsigned and signed
- Conversion, casting
- Expanding, truncating
- Addition, negation, multiplication, shifting

Representations in memory, pointers, strings

Summary
Unsigned Addition

Operands: $w$ bits

True Sum: $w+1$ bits

Discard Carry: $w$ bits

- **Standard Addition Function**
  - Ignores carry output

- **Implements Modular Arithmetic**
  \[
  s = \text{UAdd}_w(u, v) = u + v \mod 2^w
  \]

<table>
<thead>
<tr>
<th>Hex</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>A</td>
<td>10</td>
</tr>
<tr>
<td>B</td>
<td>11</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
</tr>
<tr>
<td>D</td>
<td>13</td>
</tr>
<tr>
<td>E</td>
<td>14</td>
</tr>
<tr>
<td>F</td>
<td>15</td>
</tr>
</tbody>
</table>

unsigned char

\[
\begin{array}{c}
  \text{1110 1001} \\
  + \text{1101 0101} \\
  \hline
  \text{1 1011 1110}
\end{array}
\]

\[
\begin{array}{c}
  \text{E9} \\
  \hline
  \text{1BE}
\end{array}
\]

\[
\begin{array}{c}
  \text{223} \\
  \hline
  \text{213}
\end{array}
\]

\[
\begin{array}{c}
  \text{1 1011 1110} \\
  + \text{D5} \\
  + \text{213} \\
  \hline
  \text{BE}
\end{array}
\]

\[
\begin{array}{c}
  \text{446} \\
  \hline
  \text{190}
\end{array}
\]
Visualizing (Mathematical) Integer Addition

- **Integer Addition**
  - 4-bit integers $u, v$
  - Compute true sum $\text{Add}_4(u, v)$
  - Values increase linearly with $u$ and $v$
  - Forms planar surface
Visualizing Unsigned Addition

- Wraps Around
  - If true sum $\geq 2^w$
  - At most once

True Sum

$2^{w+1}$

$2^w$

0

Modular Sum

Overflow

UAdd$_4(u,v)$

Overflow
Two's Complement Addition

Operands: $w$ bits

$$u \quad \cdot \cdot \cdot \quad \cdot \cdot \cdot \quad \cdot \cdot \cdot$$

$$+ \quad v \quad \cdot \cdot \cdot \quad \cdot \cdot \cdot \quad \cdot \cdot \cdot$$

True Sum: $w+1$ bits

$$u + v \quad \cdot \cdot \cdot \quad \cdot \cdot \cdot \quad \cdot \cdot \cdot$$

Discard Carry: $w$ bits

$$\text{TAdd}_w(u, v) \quad \cdot \cdot \cdot \quad \cdot \cdot \cdot \quad \cdot \cdot \cdot$$

- **TAdd and UAdd have Identical Bit-Level Behavior**
  - Signed vs. unsigned addition in C:
    ```
    int s, t, u, v;
    s = (int) ((unsigned) u + (unsigned) v);
    t = u + v
    ```
  - Will give $s == t$

<table>
<thead>
<tr>
<th></th>
<th>1110 1001</th>
<th>E9</th>
<th>-23</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$v$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$u + v$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{TAdd}_w(u, v)$</td>
<td>1011 1110</td>
<td>BE</td>
<td>-66</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1101 0101</th>
<th>D5</th>
<th>-43</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$v$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$u + v$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{TAdd}_w(u, v)$</td>
<td>1 1011 1110</td>
<td>1BE</td>
<td>446</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1101 1001</th>
<th>D5</th>
<th>-66</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$v$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$u + v$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{TAdd}_w(u, v)$</td>
<td>1011 1110</td>
<td>BE</td>
<td>-66</td>
</tr>
</tbody>
</table>
TAdd Overflow

**Functionality**

- True sum requires $w+1$ bits
- Drop off MSB
- Treat remaining bits as 2’s comp. integer

<table>
<thead>
<tr>
<th>True Sum</th>
<th>TAdd Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$011...1$</td>
<td>$011...1$</td>
</tr>
<tr>
<td>$010...0$</td>
<td>$000...0$</td>
</tr>
<tr>
<td>$000...0$</td>
<td>$000...0$</td>
</tr>
<tr>
<td>$1011...1$</td>
<td>$100...0$</td>
</tr>
<tr>
<td>$1000...0$</td>
<td>$100...0$</td>
</tr>
<tr>
<td>$-2^{w-1}$</td>
<td>$-2^{w-1}$</td>
</tr>
<tr>
<td>$2^{w-1}-1$</td>
<td>$2^{w-1}-1$</td>
</tr>
</tbody>
</table>
Visualizing 2’s Complement Addition

- **Values**
  - 4-bit two’s comp.
  - Range from -8 to +7

- **Wraps Around**
  - If sum $\geq 2^{w-1}$
    - Becomes negative
    - At most once
  - If sum $< -2^{w-1}$
    - Becomes positive
    - At most once
Multiplication

- **Goal:** Computing Product of $w$-bit numbers $x, y$
  - Either signed or unsigned

- **But, exact results can be bigger than $w$ bits**
  - Unsigned: up to $2w$ bits
    - Result range: $0 \leq x \times y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
  - Two’s complement min (negative): Up to $2w$-1 bits
    - Result range: $x \times y \geq (-2^{w-1})*(2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$
  - Two’s complement max (positive): Up to $2w$ bits, but only for $(TMin_w)^2$
    - Result range: $x \times y \leq (-2^{w-1})^2 = 2^{2w-2}$

- **So, maintaining exact results...**
  - would need to keep expanding word size with each product computed
  - is done in software, if needed
    - e.g., by “arbitrary precision” arithmetic packages
Unsigned Multiplication in C

Operands: \( w \) bits

True Product: \( 2w \) bits

Discard \( w \) bits: \( w \) bits

- **Standard Multiplication Function**
  - Ignores high order \( w \) bits

- **Implements Modular Arithmetic**

\[
\text{UMult}_w(u, v) = u \cdot v \mod 2^w
\]

\[
\begin{array}{c}
1110 \ 1001 \\
* \\
1101 \ 0101 \\
\hline
1100 \ 0001 \ 1101 \ 0010 \ \\
\end{array}
\]

\[
\begin{array}{c}
\text{E9} \\
* \\
\text{D5} \\
* \\
\hline
\text{C1DD} \\
\end{array}
\]

\[
\begin{array}{c}
1101 \ 1101 \\
\hline
\text{DD} \\
\end{array}
\]

\[
\begin{array}{c}
221 \\
\hline
47499 \\
\end{array}
\]
Signed Multiplication in C

Operands: $w$ bits

True Product: $2w$ bits

Discard $w$ bits: $w$ bits

### Standard Multiplication Function
- Ignores high order $w$ bits
- Some of which are different for signed vs. unsigned multiplication
- Lower bits are the same

<table>
<thead>
<tr>
<th>$u$</th>
<th>$v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1110\ 1001$</td>
<td>$1101\ 0110$</td>
</tr>
<tr>
<td>$E9$</td>
<td>$D5$</td>
</tr>
<tr>
<td>$-23$</td>
<td>$-43$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$u \cdot v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1100\ 0001\ 1101\ 0010$</td>
</tr>
<tr>
<td>$1100\ 0001\ 1101\ 0010$</td>
</tr>
<tr>
<td>$C1DD$</td>
</tr>
<tr>
<td>$16896$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$TMult_w(u, v)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1101\ 1101$</td>
</tr>
<tr>
<td>$DD$</td>
</tr>
<tr>
<td>$-35$</td>
</tr>
</tbody>
</table>
Power-of-2 Multiply with Shift

Operation

- \( u \ll k \) gives \( u \times 2^k \)
- Both signed and unsigned

Operands: \( w \) bits

True Product: \( w+k \) bits

Discard \( k \) bits: \( w \) bits

Examples

- \( u \ll 3 \)  \(==\)  \( u \times 8 \)
- \((u \ll 5) - (u \ll 3) \)  \(==\)  \( u \times 24 \)
- Most machines shift and add faster than multiply
  - Compiler generates this code automatically
Unsigned Power-of-2 Divide with Shift

- Quotient of Unsigned by Power of 2
  - \( u \gg k \) gives \( \lfloor u / 2^k \rfloor \)
  - Uses logical shift

### Operands:

#### Division:

- \( u / 2^k \)

### Result:

- \( \lfloor u / 2^k \rfloor \)

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>15213</td>
<td>3B 6D</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>( x \gg 1)</td>
<td>7606.5</td>
<td>1D B6</td>
<td>00011101 10110110</td>
</tr>
<tr>
<td>( x \gg 4)</td>
<td>950.8125</td>
<td>03 B6</td>
<td>00000011 10110110</td>
</tr>
<tr>
<td>( x \gg 8)</td>
<td>59.4257813</td>
<td>00 3B</td>
<td>00000000 00111011</td>
</tr>
</tbody>
</table>
Signed Power-of-2 Divide with Shift

- Quotient of Signed by Power of 2
  - $x \gg k$ gives $\lfloor x / 2^k \rfloor$
  - Uses arithmetic shift
  - Rounds wrong direction when $u < 0$

Operands:

<table>
<thead>
<tr>
<th>$x$</th>
<th>$2^k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>$0 \cdots 010 \cdots 00$</td>
</tr>
</tbody>
</table>

Division:

<table>
<thead>
<tr>
<th>$x$</th>
<th>$2^k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>$\cdots \cdots \cdots \cdots$</td>
</tr>
</tbody>
</table>

Result: $\text{RoundDown}(x / 2^k)$

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y$</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>$y \gg 1$</td>
<td>-7606.5</td>
<td>E2 49</td>
<td>11100010 01001001</td>
</tr>
<tr>
<td>$y \gg 4$</td>
<td>-950.8125</td>
<td>FC 49</td>
<td>11111100 01001001</td>
</tr>
<tr>
<td>$y \gg 8$</td>
<td>-59.4257813</td>
<td>FF C4</td>
<td>11111111 11000100</td>
</tr>
</tbody>
</table>
Correct Power-of-2 Divide

**Quotient of Negative Number by Power of 2**

- Want \( \left\lfloor \frac{x}{2^k} \right\rfloor \) (Round Toward 0)
- Compute as \( \left\lfloor \frac{x+2^k-1}{2^k} \right\rfloor \)
  - In C: \((x + (1<<k)-1) >> k\)
  - Biases dividend toward 0

**Case 1: No rounding**

<table>
<thead>
<tr>
<th>Dividend:</th>
<th>Divisor:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( u )</td>
<td>( 2^k )</td>
</tr>
<tr>
<td>( +2^k - 1 )</td>
<td></td>
</tr>
<tr>
<td>( u / 2^k )</td>
<td>( u / 2^k )</td>
</tr>
</tbody>
</table>

**Biasing has no effect**
Correct Power-of-2 Divide (Cont.)

Case 2: Rounding

Dividend:

\[
x \quad + \quad 2^k - 1
\]

Divisor:

\[
x \quad / \quad 2^k
\]

Biasing adds 1 to final result
Negation: Complement & Increment

- Negate through complement and increase
  \[ \sim x + 1 = -x \]

- Example
  - Observation: \( \sim x + x = 1111...111 = -1 \)

\[
\begin{array}{c}
\text{x} & 10011101 \\
+ \sim x & 01100010 \\
\hline
\text{-1} & 11111111
\end{array}
\]

\[x = 15213\]

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x)</td>
<td>15213</td>
<td>3B 6D</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>(\sim x)</td>
<td>-15214</td>
<td>C4 92</td>
<td>11000100 10010010</td>
</tr>
<tr>
<td>(\sim x+1)</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>(y)</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
</tbody>
</table>
## Complement & Increment Examples

### $x = 0$

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0</td>
<td>00 00 00000000000000</td>
</tr>
<tr>
<td>~0</td>
<td></td>
<td>-1</td>
<td>FF FF 111111111111111</td>
</tr>
<tr>
<td>~0+1</td>
<td></td>
<td>0</td>
<td>00 00 00000000000000</td>
</tr>
</tbody>
</table>

### $x = \text{TMin}$

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>-32768</td>
<td>80</td>
<td>00 00 10000000 00000000</td>
</tr>
<tr>
<td>~x</td>
<td>32767</td>
<td>7F</td>
<td>FF 01111111 11111111</td>
</tr>
<tr>
<td>~x+1</td>
<td>-32768</td>
<td>80</td>
<td>00 00 10000000 00000000</td>
</tr>
</tbody>
</table>

**Canonical counter example**
Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary
- Representations in memory, pointers, strings
Arithmetic: Basic Rules

- **Addition:**
  - Unsigned/signed: Normal addition followed by truncate, same operation on bit level
  - Unsigned: addition mod $2^w$
    - Mathematical addition + possible subtraction of $2^w$
  - Signed: modified addition mod $2^w$ (result in proper range)
    - Mathematical addition + possible addition or subtraction of $2^w$

- **Multiplication:**
  - Unsigned/signed: Normal multiplication followed by truncate, same operation on bit level
  - Unsigned: multiplication mod $2^w$
  - Signed: modified multiplication mod $2^w$ (result in proper range)
Why Should I Use Unsigned?

- *Don’t use without understanding implications*
  - Easy to make mistakes
    ```
    unsigned i;
    for (i = cnt-2; i >= 0; i--)
    a[i] += a[i+1];
    ```
  - Can be very subtle
    ```
    #define DELTA sizeof(int)
    int i;
    for (i = CNT; i-DELTA >= 0; i-= DELTA)
    . . .
    ```
Counting Down with Unsigned

- **Proper way to use unsigned as loop index**
  
  ```c
  unsigned i;
  for (i = cnt-2; i < cnt; i--)
    a[i] += a[i+1];
  ```

- **See Robert Seacord, *Secure Coding in C and C++***
  - C Standard guarantees that unsigned addition will behave like modular arithmetic
    - \( 0 - 1 \rightarrow UMax \)

- **Even better**
  
  ```c
  size_t i;
  for (i = cnt-2; i < cnt; i--)
    a[i] += a[i+1];
  ```
  - Data type `size_t` defined as unsigned value with length = word size
  - Code will work even if `cnt = UMax`
  - What if `cnt` is signed and < 0?
Why Should I Use Unsigned? (cont.)

- **Do Use When Performing Modular Arithmetic**
  - Multiprecision arithmetic

- **Do Use When Using Bits to Represent Sets**
  - Logical right shift, no sign extension

- **Do Use In System Programming**
  - Bit masks, device commands,...
### Integer Arithmetic Example

**Unsigned char**

<table>
<thead>
<tr>
<th>Hex</th>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>0111</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>A</td>
<td>10</td>
<td>1010</td>
</tr>
<tr>
<td>B</td>
<td>11</td>
<td>1011</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td>1100</td>
</tr>
<tr>
<td>D</td>
<td>13</td>
<td>1101</td>
</tr>
<tr>
<td>E</td>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>F</td>
<td>15</td>
<td>1111</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>unsigned char</th>
<th>F3</th>
<th>243</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111 0011</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ 0101 0010</td>
<td>+ 52</td>
<td>+ 82</td>
</tr>
<tr>
<td>1 0100 0101</td>
<td>145</td>
<td>325</td>
</tr>
<tr>
<td>0101 0101</td>
<td>45</td>
<td>69</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>unsigned char</th>
<th>19</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001 1001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>* 0000 0010</td>
<td>* 02</td>
<td>* 2</td>
</tr>
<tr>
<td>0 0011 0010</td>
<td>032</td>
<td>50</td>
</tr>
<tr>
<td>0011 0010</td>
<td>32</td>
<td>50</td>
</tr>
</tbody>
</table>
Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations

Integers

- Representation: unsigned and signed
- Conversion, casting
- Expanding, truncating
- Addition, negation, multiplication, shifting
- Summary

- Representations in memory, pointers, strings
Byte-Oriented Memory Organization

- Programs refer to data by address
  - Conceptually, envision it as a very large array of bytes
    - In reality, it’s not, but can think of it that way
  - An address is like an index into that array
    - and, a pointer variable stores an address

- Note: system provides private address spaces to each “process”
  - Think of a process as a program being executed
  - So, a program can clobber its own data, but not that of others
Any given computer has a “Word Size”
- Nominal size of integer-valued data
  - and of addresses

- Until recently, most machines used 32 bits (4 bytes) as word size
  - Limits addresses to 4GB ($2^{32}$ bytes)

- Increasingly, machines have 64-bit word size
  - Potentially, could have 18 EB (exabytes) of addressable memory
  - That’s $18.4 \times 10^{18}$

- Machines still support multiple data formats
  - Fractions or multiples of word size
  - Always integral number of bytes
### Word-Oriented Memory Organization

**Addresses Specify Byte Locations**

- Address of first byte in word
- Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)

<table>
<thead>
<tr>
<th>32-bit Words</th>
<th>64-bit Words</th>
<th>Bytes</th>
<th>Addr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr = 0000</td>
<td>Addr = 0000</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>Addr = 0004</td>
<td>Addr = 0008</td>
<td>0001</td>
<td></td>
</tr>
<tr>
<td>Addr = 0008</td>
<td>Addr = 0008</td>
<td>0002</td>
<td></td>
</tr>
<tr>
<td>Addr = 0012</td>
<td></td>
<td>0003</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Addr = 0008</td>
<td>0004</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Addr = 0008</td>
<td>0005</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Addr = 0008</td>
<td>0006</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Addr = 0008</td>
<td>0007</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Addr = 0008</td>
<td>0008</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Addr = 0008</td>
<td>0009</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Addr = 0008</td>
<td>0010</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Addr = 0008</td>
<td>0011</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Addr = 0008</td>
<td>0012</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Addr = 0008</td>
<td>0013</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Addr = 0008</td>
<td>0014</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Addr = 0008</td>
<td>0015</td>
<td></td>
</tr>
</tbody>
</table>
## Example Data Representations

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Typical 32-bit</th>
<th>Typical 64-bit</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>4</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>pointer</td>
<td>4</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>
Byte Ordering

- So, how are the bytes within a multi-byte word ordered in memory?

- Conventions
  - Big Endian: Sun, PPC Mac, Internet
    - Least significant byte has highest address
  - Little Endian: x86, ARM processors running Android, iOS, and Windows
    - Least significant byte has lowest address
Byte Ordering Example

Example
- Variable x has 4-byte value of 0x01234567
- Address given by &x is 0x100

<table>
<thead>
<tr>
<th>Big Endian</th>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>01</td>
<td>23</td>
<td>45</td>
<td>67</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Little Endian</th>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>67</td>
<td>45</td>
<td>23</td>
<td>01</td>
</tr>
</tbody>
</table>
Representing Integers

\[
\text{int } A = 15213; \\
\text{long int } C = 15213;
\]

\[
\begin{array}{|c|c|}
\hline
\text{IA32, x86-64} & \text{Sun} \\
\hline
6D & 00 \\
3B & 00 \\
00 & 6D \\
00 & 3B \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\text{IA32} & \text{x86-64} & \text{Sun} \\
\hline
6D & 6D & 00 \\
3B & 3B & 00 \\
00 & 00 & 6D \\
00 & 00 & 3B \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\text{IA32, x86-64} & \text{Sun} \\
\hline
93 & FF \\
C4 & FF \\
FF & C4 \\
FF & 93 \\
\hline
\end{array}
\]

Two’s complement representation.
Examining Data Representations

- Code to Print Byte Representation of Data
  - Casting pointer to unsigned char * allows treatment as a byte array

```c
typedef unsigned char *pointer;

void show_bytes(pointer start, size_t len){
    size_t i;
    for (i = 0; i < len; i++)
        printf("%p	0x%.2x\n", start+i, start[i]);
    printf("\n");
}
```

Printf directives:
- %p: Print pointer
- %x: Print Hexadecimal
show_bytes Execution Example

```c
int a = 15213;
printf("int a = 15213;\n");
show_bytes((pointer) &a, sizeof(int));
```

Result (Linux x86-64):

```c
int a = 15213;
0x7ffffffb7f71dbc 6d
0x7ffffffb7f71dbd 3b
0x7ffffffb7f71dbe 00
0x7ffffffb7f71dbf 00
```
# Representing Pointers

Here's an example demonstrating how different compilers & machines assign different locations to objects:

```c
int B = -15213;
int *P = &B;
```

### Different Compilers & Machines

**Sun**

- EF
- FF
- FB
- 2C

**IA32**

- AC
- 28
- F5
- FF

**x86-64**

- 3C
- 1B
- FE
- 82
- FD
- 7F
- 00
- 00

Even get different results each time run program.
Representing Strings

- **Strings in C**
  - Represented by array of characters
  - Each character encoded in ASCII format
    - Standard 7-bit encoding of character set
    - Character “0” has code 0x30
      - Digit $i$ has code 0x30+$i$
  - String should be null-terminated
    - Final character = 0

- **Compatibility**
  - Byte ordering not an issue

```c
char S[6] = "18213";
```
Reading Byte-Reversed Listings

- **Disassembly**
  - Text representation of binary machine code
  - Generated by program that reads the machine code

- **Example Fragment**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Code</th>
<th>Assembly Rendition</th>
</tr>
</thead>
<tbody>
<tr>
<td>8048365:</td>
<td>5b</td>
<td>pop %ebx</td>
</tr>
<tr>
<td>8048366:</td>
<td>81 c3 ab 12 00 00</td>
<td>add $0x12ab,%ebx</td>
</tr>
<tr>
<td>804836c:</td>
<td>83 bb 28 00 00 00 00</td>
<td>cmp $0x0,0x28(%ebx)</td>
</tr>
</tbody>
</table>

- **Deciphering Numbers**
  - Value: 0x12ab
  - Pad to 32 bits: 0x000012ab
  - Split into bytes: 00 00 12 ab
  - Reverse: ab 12 00 00
Integer C Puzzles

\[
x < 0 \quad \Rightarrow \quad ((x \times 2) < 0)
\]
\[
ux \geq 0
\]
\[
x \& 7 == 7 \quad \Rightarrow \quad (x << 30) < 0
\]
\[
ux > -1
\]
\[
x > y \quad \Rightarrow \quad -x < -y
\]
\[
x \times x \geq 0
\]
\[
x > 0 && y > 0 \quad \Rightarrow \quad x + y > 0
\]
\[
x \geq 0 \quad \Rightarrow \quad -x \leq 0
\]
\[
x \leq 0 \quad \Rightarrow \quad -x \geq 0
\]
\[
(x | -x) >> 31 == -1
\]
\[
ux >> 3 == ux / 8
\]
\[
x >> 3 == x / 8
\]
\[
x \& (x - 1) != 0
\]

Initialization

```c
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```
Summary

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
- Representations in memory, pointers, strings
- Summary