Course Theme

- Abstraction is good, but don't forget reality!
- Many CS Courses emphasize abstraction
  - Abstract data types
  - Asymptotic analysis
- These abstractions have limits
  - Especially in the presence of bugs
  - Need to understand underlying implementations
- Useful outcomes
  - Become more effective programmers
  - Able to find and eliminate bugs efficiently
  - Able to tune program performance
  - Prepare for later "systems" classes in CS
    - Compilers, Operating Systems, File Systems, Computer Architecture, Robotics, etc.

Textbooks

- Randal E. Bryant and David R. O'Hallaron,
- Brian Kernighan and Dennis Ritchie,
- Larry Miller and Alex Quilici
Syllabus

- Syllabus on Web: https://www.cs.hmc.edu/~geoff/cs105
- Calendar defines due dates
  - Also has links to slides and labs
- Labs: cs105submit for some, others have specific directions

Notes:

Work groups
- You must work in pairs on all labs
- Honor-code violation to work without your partner!
- Corollary: showing up late doesn't harm only you

Handins
- Check calendar for due dates
- Electronic submissions only

Grading Characteristics
- Lab scores tend to be high
  - Serious handicap if you don't hand a lab in
- Tests & quizzes typically have a wider range of scores
  - I.e., they're have major effect on your grade
  - ...but not the ONLY one
- Do your share of lab work and reading, or bomb tests
- Do practice problems in book

Facilities

Assignments will use Intel computer systems
- Not all machines are created alike
  - Performance varies (and matters sometimes in 105)
  - Security settings vary and can matter
- Wilkes: x86/Linux specifically set up for this class
- Log in on a Mac, then ssh to Wilkes
  - If you want fancy programs, start X11 first
  - Directories are cross-mounted, so you can edit on Knuth or your Mac, and Wilkes will see your files
  - ...or ssh into Wilkes from wherever you are
- All programs must run on Wilkes: we grade there
- Have lecture slides (and textbook) available when working on labs!

CS 105

"Tour of the Black Holes of Computing"

Bits, Bytes, Integers

Topics
- 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
Everything is bits

Each bit is 0 or 1

By encoding/interpreting sets of bits in various ways

- Computers determine what to do (instructions)
- ... and represent and manipulate numbers, sets, strings, etc...

Why bits? Electronic implementation

- Easy to store with bistable elements
- Reliably transmitted on noisy and inaccurate wires

Encoding Byte Values

Byte = 8 bits

- Binary: 00000000 to 11111111
- Decimal: 0 to 255
- Hexadecimal: 0x00 to 0xFF

Use characters '0' to '9' and 'A' to 'F'

Write FA1D37B in C as

```c
0xFA1D37B
0xfa1d37b
```

Example Data Sizes

<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>8</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>long double</td>
<td>-</td>
<td>-</td>
<td>10/16</td>
</tr>
<tr>
<td>pointer</td>
<td>4</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

Boolean Algebra

Developed by George Boole in 19th century

- Algebraic representation of logic
  - Encode “True” as 1 and “False” as 0

<table>
<thead>
<tr>
<th>A</th>
<th>~A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

And

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

Or

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Not

<table>
<thead>
<tr>
<th>A</th>
<th>~A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Exclusive-Or (Xor)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A^B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
General Boolean Algebras

Operate on bit vectors
- Operations applied bitwise

01101001 & 01010101 = 01000001
01101001 | 01010101 = 01111101
01101001 ^ 01010101 = 00111100
~ 01010101 = 10101010

01101001 01000001 01111101 00111100 10101010

All of the properties of Boolean algebra apply

Example: Representing & Manipulating Sets

Representation
- Width w bit vector represents subsets of {0, …, w–1}
  - \( a_j = 1 \) if \( j \in A \)

\[
\begin{align*}
01101001 & \quad \{ 0, 3, 5, 6 \} \\
01010101 & \quad \{ 0, 2, 4, 6 \}
\end{align*}
\]

Operations
- & Intersection 01000001  \( \{ 0, 6 \} \)
- | Union 01111101  \( \{ 0, 2, 3, 4, 5, 6 \} \)
- ^ Symmetric difference 00111100  \( \{ 2, 3, 4, 5 \} \)
- ~ Complement 10101010  \( \{ 1, 3, 5, 7 \} \)

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
- Operations applied bit-wise

Examples (char data type)
- "0x41 \rightarrow 0x41"
- "0x00 \rightarrow 0xFF"
- "0x00 \rightarrow 0x00"
- "0x00 \rightarrow 0x00"
- 0x69 & 0x55 \rightarrow 0x41
- 01101001 & 01010101 \rightarrow 01000001
- 0x69 | 0x55 \rightarrow 0x7D
- 0x69 | 0x55 \rightarrow 0x7D
- 01101001, 01010101 \rightarrow 01111101,

Contrast: Logic Operations in C

Contrast to Logical Operators
- &.| |. !
  - View 0 as "False"
  - Anything nonzero seen as "True"
  - Always return 0 or 1
  - Early termination

Examples (char data type)
- 10x41 \rightarrow 0x00
- 10x00 \rightarrow 0x01
- 10x41 \rightarrow 0x00
- 0x69 & 0x55 \rightarrow 0x01
- 0x69 | 0x55 \rightarrow 0x01
- 0x69 | 0x55 \rightarrow 0x01
- p = 0 & p (unreadably avoids null pointer access)
Shift Operations

Left Shift: \[ x << y \]
- Shift bit-vector \( x \) left \( y \) positions
- Throw away extra bits on left
- Fill with 0's on right

Right Shift: \[ x >> y \]
- Shift bit-vector \( x \) right \( y \) positions
- Throw away extra bits on right
- Logical shift
  - Fill with 0's on left
- Arithmetic shift
  - Replicate most significant bit on left

Undefined Behavior
- Shift amount < 0 or word size

C Puzzles
- Taken from old exams
- Assume machine with 32-bit word size, two's complement integers
- For each of the following C expressions, either:
  - Argue that it is true for all argument values, or
  - Give example where it is not true

- \( x < 0 \) \[ ((x*2) < 0) \]
- \( ux >= 0 \) \[ (x<<30) < 0 \]
- \( x & 7 == 7 \) \[ (x<<2) < 0 \]
- \( ux > -1 \) \[ -x < -y \]
- \( x * x >= 0 \) \[ -x <= 0 \]
- \( x <= 0 \) \[ -x >= 0 \]

Initialization
- \( x = \text{foo}(); \)
- \( y = \text{bar}(); \)
- \( \text{unsigned} \ ux = x; \)
- \( \text{unsigned} \ uy = y; \)

Encoding Integers

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x ) = 15213</td>
<td>00111011 01101101</td>
<td></td>
</tr>
<tr>
<td>( y ) = -15213</td>
<td>11000100 10010011</td>
<td></td>
</tr>
</tbody>
</table>

Sign Bit
- For 2's complement, most-significant bit indicates sign
  - 0 for nonnegative
  - 1 for negative

Encoding Integers (Cont.)

<table>
<thead>
<tr>
<th>Weight</th>
<th>15213</th>
<th>-15213</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>64</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>128</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>256</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>512</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1024</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2048</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4096</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>8192</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>16384</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>-32768</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Sum 15213 -15213
### Numeric Ranges

#### Unsigned Values
- UMin = 0
- UMax = \(2^w - 1\)

#### Two’s-Complement Values
- TMin = \(-2^{w-1}\)
- TMax = \(2^{w-1} - 1\)

#### Other Values
- Minus 1

<table>
<thead>
<tr>
<th>Values for W = 16</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Decimal</strong></td>
</tr>
<tr>
<td>UMax</td>
</tr>
<tr>
<td>TMax</td>
</tr>
<tr>
<td>Tmin</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>0</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>18,446,744,073,709,551,615</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

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

### An Important Detail

No self-identifying data
- Looking at a bunch of bits doesn’t tell you what they mean
- Could be signed, unsigned integer
- Could be floating-point number
- Could be part of a string

Only the program (instructions) knows for sure!
- (To be fair, experienced humans make good guesses—see Lab 2)

### Unsigned & Signed Numeric Values

<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>

#### Equivalence
- Same encodings for nonnegative values

#### Uniqueness
- Every bit pattern represents unique integer value
- Each representable integer has unique bit encoding
Mapping Between Signed & Unsigned

Mappings between unsigned and two’s complement numbers:
Keep bit representations and reinterpret

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

Casting Signed to Unsigned

C Allows Conversions from Signed to Unsigned

short int          x = 15213;
unsigned short int ux = (unsigned short) x;
short int          y = -15213;
unsigned short int uy = (unsigned short) y;

Resulting Value

- No change in bit representation
- Nonnegative values unchanged
  - ux = 15213
- Negative values change into (large) positive values!
  - uy = 50323
### Relation Between Signed & Unsigned

<table>
<thead>
<tr>
<th>Two's Complement</th>
<th>x</th>
<th>T2U</th>
<th>B2U</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maintain Same Bit Pattern</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Conversion Visualized

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

#### Signed vs. Unsigned in C

**Integer Constants**
- By default are considered to be signed integers
- Exception: unsigned, if too big to be signed but fit in unsigned
- Unsigned if have "U" as suffix
- 0u, 4294967295u

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

#### Casting Surprises

**Expression Evaluation**
- If you mix unsigned and signed in single expression, signed values are implicitly cast to unsigned
- Including comparison operations `<`, `>`, `==`, `<=`, `>=`
- Examples for $W = 32$

<table>
<thead>
<tr>
<th>Constant&lt;sub&gt;1&lt;/sub&gt;</th>
<th>Constant&lt;sub&gt;2&lt;/sub&gt;</th>
<th>Relation Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0u</td>
<td></td>
</tr>
<tr>
<td>−1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td>−2147483648</td>
<td></td>
</tr>
<tr>
<td>2147483647u</td>
<td>−2147483648u</td>
<td></td>
</tr>
<tr>
<td>−1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td>−2147483648</td>
<td></td>
</tr>
<tr>
<td>(unsigned)−1</td>
<td>−2</td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td>2147483648u</td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td>(int)2147483648u</td>
<td></td>
</tr>
</tbody>
</table>
Summary: Casting Signed $\leftrightarrow$ Unsigned: Basic Rules
Bit pattern is maintained—but reinterpreted
Can have unexpected effects: adding or subtracting $2^w$

In expression containing signed and unsigned int:
- int is cast to unsigned!!

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

Sign Extension Example

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>01234</td>
<td>01010000 00111011 01101101</td>
</tr>
<tr>
<td>$lx$</td>
<td>00001</td>
<td>00000000 00111011 01101101</td>
</tr>
<tr>
<td>$y$</td>
<td>01234</td>
<td>11000010 10010011</td>
</tr>
<tr>
<td>$ly$</td>
<td>FF FF</td>
<td>1111111 1111111 1111111 1111111</td>
</tr>
</tbody>
</table>

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

Negating with Complement & Increment

Claim: Following holds for 2’s complement

$-x + 1 = -x$

Complement
- Observation: $-x + x = 1111\ldots11 = -1$
  - $x + (-x + 1) = -1$
  - $x + 1 = -x$ (associativity and commutativity hold)

Warning: Be cautious treating int’s as integers
- OK here (associativity and commutativity hold)
Unsigned Addition

Operands: $w$ bits

True Sum: $w+1$ bits

Discard Carry: $w$ bits

Standard Addition Function

- Ignores carry output
- Implements Modular Arithmetic

$s = \text{UAdd}_w(u,v) = u + v \mod 2^w$

Two’s-Complement Addition

Operands: $w$ bits

True Sum: $w+1$ bits

Discard Carry: $w$ bits

TAdd and UAdd have identical bit-level behavior

- Signed vs. unsigned addition in C:
  ```c
  int s, t, u, v;
  s = (int) ((unsigned)u + (unsigned)v);
  t = u + v;
  ```

  Will give $s == t$

Detecting 2’s-Complement Overflow

Task

- Given $s = \text{TAdd}_w(u,v)$
- Determine if $s = \text{Add}_w(u,v)$
- Example
  ```c
  int s, u, v;
  s = u + v;
  ```

Claim

- Overflow if either:
  - $u, v < 0, s \geq 0 \text{ (NegOver) }$
  - $u, v \geq 0, s < 0 \text{ (PosOver) }$

A Fun Fact

Official C standard says overflow is “undefined”

- Intention was to let machine define what happens

Recently compiler writers have decided “undefined” means “we get to choose”

- We can generate 0, biggest integer, or anything else
- Or if we’re sure it’ll overflow, we can optimize out completely
- This can introduce some lovely bugs (e.g., it’s tricky to check for overflow)

Fight between compiler community and security community over this issue
Multiplication

Computing exact product of w-bit numbers \( x, y \)
- Either signed or unsigned

Ranges
- **Unsigned**: \( 0 \leq x \cdot y \leq (2^w - 1)^2 = 2^{2w} - 2^w + 1 \)
  - Up to 2w bits
- **Two’s complement min**: \( x \cdot y = -(2^w - 1)^2 = -2^{2w} - 2^w + 1 \)
  - Up to 2w-1 bits (including 1 for sign)
- **Two’s complement max**: \( x \cdot y = (2^w - 1)^2 = 2^{2w} - 2^w + 1 \)
  - Up to 2w bits, but only for \( T_{\text{Min}} \)

Maintaining exact results
- Would need to keep expanding word size with each product computed
- Done in software by “arbitrary-precision” arithmetical packages

Power-of-2 Multiply by Shifting

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

<table>
<thead>
<tr>
<th>Operands: w bits</th>
<th>( u \ll k )</th>
<th>( u \cdot 2^k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Division: ( u \ll k )</td>
<td>( u \cdot 2^k )</td>
<td>( u \cdot 2^k )</td>
</tr>
<tr>
<td>Result: ( u \ll k )</td>
<td>( u \cdot 2^k )</td>
<td>( u \cdot 2^k )</td>
</tr>
</tbody>
</table>

Examples
- \( u \ll 1 = u \cdot 2 \)
- \( u \ll 5 - u \ll 3 = u \cdot 24 \)
- Most machines shift and add much faster than multiply
  - Compiler generates this code automatically

Unsigned Power-of-2 Divide by Shifting

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

<table>
<thead>
<tr>
<th>Operands: ( u \gg k )</th>
<th>( u \gg k ) ( / 2^k )</th>
<th>Binary Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Division: ( u \gg k )</td>
<td>( u / 2^k )</td>
<td>( u / 2^k )</td>
</tr>
<tr>
<td>Result: ( u / 2^k )</td>
<td>( u / 2^k )</td>
<td>( u / 2^k )</td>
</tr>
</tbody>
</table>

Arithmetic: Basic Rules

Addition:
- Signed/unsigned: Normal addition followed by truncate; same operation on bit level
- Signed: 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:
- Signed/unsigned: Normal multiplication followed by truncate; same operation on bit level
- Signed: multiplication mod 2^w
- Signed: modified multiplication mod 2^w (result in range -2^{w-1} to 2^{w-1}-1)
Why Should I Use Unsigned?

Don’t use without understanding implications
- Easy to make mistakes
  ```c
  unsigned i;
  for (i = cnt-2; i >= 0; i--)
    a[i] += a[i+1];
  ```
- Can be very subtle
  ```c
  #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
- Use with care
  ```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 unsigned addition will behave like modular arithmetic
  - For unsigned i, 0 - 1 \(\equiv\) UMax
- Even better
  ```c
  size_t i;
  for (i = cnt-2; i < cnt; i--)
    a[i] += a[i+1];
  ```
  - Data type size_t is unsigned value with length = word size
  - Code will work even if cnt \(\equiv\) 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

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
Machine Words

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 PB (petabytes) of addressable memory
    - That’s 18.4 X 10^{15}
- 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)

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

Big Endian

<table>
<thead>
<tr>
<th>Addr</th>
<th>Bytes</th>
<th>Addr</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>01</td>
<td>0001</td>
<td>23</td>
</tr>
<tr>
<td>0002</td>
<td>45</td>
<td>0003</td>
<td>67</td>
</tr>
<tr>
<td>0004</td>
<td></td>
<td>0005</td>
<td></td>
</tr>
<tr>
<td>0006</td>
<td></td>
<td>0007</td>
<td></td>
</tr>
<tr>
<td>0008</td>
<td></td>
<td>0009</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td></td>
<td>0011</td>
<td></td>
</tr>
<tr>
<td>0012</td>
<td></td>
<td>0013</td>
<td></td>
</tr>
<tr>
<td>0014</td>
<td></td>
<td>0015</td>
<td></td>
</tr>
</tbody>
</table>

Little Endian

<table>
<thead>
<tr>
<th>Addr</th>
<th>Bytes</th>
<th>Addr</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0012</td>
<td>67</td>
<td>0013</td>
<td>45</td>
</tr>
<tr>
<td>0014</td>
<td>23</td>
<td>0015</td>
<td>01</td>
</tr>
<tr>
<td>0010</td>
<td></td>
<td>0011</td>
<td></td>
</tr>
<tr>
<td>0012</td>
<td></td>
<td>0013</td>
<td></td>
</tr>
<tr>
<td>0014</td>
<td></td>
<td>0015</td>
<td></td>
</tr>
</tbody>
</table>

This is what we use in 105

And it will drive you nuts!
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 has code 0x30
  - String should be null-terminated
  - Final character = 0

Compatibility

- Byte ordering not an issue

```
char S[6] = "15213";
```