Csci 2021 Class Web Pages

Departmental Class Web Page:
- Public access without login
- Course syllabus, schedule, lecture notes, other useful info
  http://www-users.cselabs.umn.edu/classes/Spring-2016/csci2021/
- You should read course syllabus carefully for all course policies, personnel info, and course logistics.

Moodle 2.8 Web page:
- Use your university x500 account to access
- Class forums, assignment submission, exam info
  https://ay15.moodle.umn.edu/course/view.php?id=13068

Data Representation: 
Bits, Bytes, and Integers

CSci 2021 - Machine Architecture and Organization
Professor Pen-Chung Yew

With slides from Randy Bryant and Dave O'Hallaron

Bits, Bytes, and Integers

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

Binary Representations

- Each bit is 0 or 1
- Why bits? Electronic Implementation
  - Easy to store with bistable elements, e.g. switches, transistors...
  - Reliably transmitted on noisy wires
- By encoding/interpreting sets of bits in various ways
  - Computers determine what to do (instructions)... and represent and manipulate numbers, sets, strings, etc...

Base-2 number representation

- Decimal to Binary
  - Represent 152 as 10011000
  - Represent 1.20 as 1.0011[0011]... [binary repeat]
  - Represent 1.51 as 1.0010001

- Binary to Decimal
  - Represent 10110 as 22
  - Represent 10.1101 as 5.625
  - Represent 1.0111 X 2^3 as 1.38 X 10^3

Encoding Byte Values

- Byte = 8 bits
  - Binary: 00000000 to 11111111
  - Decimal: 0 to 255
  - Hexadecimal: 0x0 to 0xFF
  - Base 16 number representation
    - Use characters '0' to '9' and 'A' to 'F'
    - Write FA1D37BE in C as
      - 0xFA1D37BE
      - 0xa1d37be
      - 0xFa1D37Be

0.0V 0.2V 0.9V 1.1V

1.1V 0.9V 0.2V 0.0V
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>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>

Today: Bits, Bytes, and Integers

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

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 \( A=1 \) and \( B=1 \)

Or
- \( A \mid B = 1 \) when either \( A=1 \) or \( B=1 \)

Not
- \( \sim A = 1 \) when \( A=0 \)

Exclusive-Or (Xor)
- \( A \text{^} B = 1 \) when either \( A=1 \) or \( B=1 \), but not both

Application of Boolean Algebra

- Applied to Digital Systems by Claude Shannon
  - 1937 MIT Master's Thesis
  - Reason about networks of relay switches
  - Encode closed switch as 1, open switch as 0

\[ A \& \sim B \mid \sim A \& B = A \text{^} B \]

General Boolean Algebras

- Operate on Bit Vectors
  - Operations applied bitwise
    - 01101001 01101001 01101001 01101001 01101001 01101001
    - 01000001 01111101 00111100 10101010
  - All of the Properties of Boolean Algebra Apply

Representing & Manipulating Sets

- Representation
  - Width \( w \) bit vector represents subsets of \( \{0, \ldots, w-1\} \)
  - \( a_j = 1 \) if \( j \in A \)
    - 01101001 \( A = \{0, 3, 5, 6\} \)
    - 76543210
    - 01010101 \( A = \{0, 2, 4, 6\} \)
    - 76543210
- 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
  - Arguments applied bit-wise
- Examples (char data type) (1 byte)
  \[-0x69 = 0x69\]
  \[-0x00 = 0xFF\]
  \[-0x69 | 0x55 = 0x69\]
  \[-0x069 & 0x55 = 0x69\]
  \[-0x10101011 & 0x10101011 = 0x10101011\]

Contrast: Logic Operations in C

- Contrast to Logical Operators
  - & & | | !
  - View 0 as “False”
  - Anything nonzero as “True”
  - Always return 0 or 1
- Examples (char data type)
  \[-0x69 || 0x55 = 0x01\]
  \[-0x69 && 0x55 = 0x01\]
  \[-!!0x41 = 0x01\]
  \[-!0x00 = 0x01\]
  \[-!0x41 = 0x00\]

Shift Operations

- Left Shift: x << y
  - Shift bit-vector x left y positions
  - Fill with 0’s on right
- Right Shift: x >> y
  - Shift bit-vector x right y positions
  - Throw away extra bits on right
- Arith.
  - Arithmetic shift
  - Fill with 0’s on left
- Logical shift
  - Throw away extra bits on right
- Undefined Behavior
  - Shift amount < 0 or 2 word size

Contrast: Logic Operations in C

- Contrast to Logical Operators
  - & & | | !
  - View 0 as “False”
  - Anything nonzero as “True”
  - Always return 0 or 1
- Examples (char data type)
  \[-0x69 || 0x55 = 0x01\]
  \[-0x69 && 0x55 = 0x01\]
  \[-!!0x41 = 0x01\]
  \[-!0x00 = 0x01\]
  \[-!0x41 = 0x00\]

Today: Bits, Bytes, and Integers

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

Announcement 1/25/2016

- Programming Assignment #1 (Data Lab) will be issued today, due in 2 weeks on 2/8/2016 11:55pm Monday
  - Download Data Lab from class Moodle page
  - This lab asks you to manipulate bit strings using restricted instruction types and counts
  - Each problem in the assignment has different degrees of difficulty, start with easier ones first.
  - Feel free to use class Forum to ask questions and get information from TA or other students
  - TA will discuss Data Lab in Thur recitation sessions.
**EncodingUnsigned and Signed Integers**

- **Unsigned (B2U)**
  \[ B2U(x) = \sum_{i=0}^{w-1} x_i 2^i \]

- **Two’s Complement (B2T)**
  \[ B2T(x) = -x + 2^w \]

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

  - **In C, data type short is 2 bytes long**

**Binary Weight Table**

- \( 2^1 = 2 \)
- \( 2^2 = 4 \)
- \( 2^3 = 8 \)
- \( 2^4 = 16 \)
- \( 2^5 = 32 \)
- \( 2^6 = 64 \)
- \( 2^7 = 128 \)
- \( 2^8 = 256 \)
- \( 2^9 = 512 \)
- \( 2^{10} = 1024 \)

**Encoding Example (Binary to Unsigned – B2U)**

- **x = 15213**: \( x = \sum_{i=0}^{15} x_i 2^i \)

**Converting Unsigned to Binary (U2B)**

Solving \[ B2U(x) = \sum_{i=0}^{w-1} x_i 2^i \]

**Values for Different Word Sizes**

- **Observations**
  - \(|\text{TMin}| = TMax + 1\)
  - Asymmetric range

- **C Programming**
  - #include <limits.h>
  - Declares constants, e.g.,
    - ULONG_MAX
    - LONG_MAX
    - LONG_MIN
  - Values platform specific
  - Need to know -> to avoid overflow

**Numeric Ranges**

- **Unsigned Values**
  - **Unsigned Min**: \( 0 \)
  - **Unsigned Max**: \( 2^w - 1 \)

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

- **Other Values**
  - **Minus 1**: \( -2^{w-1} + 1 \)

**Ranges for \( w = 16 \), i.e. 2 bytes (short)**

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>00000000 00000000</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>00000000 00000000</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>00000000 00000000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>65535</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>65536</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>65537</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>65538</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
</tbody>
</table>

- **TMax**: 65535
- **TMin**: -32768

**Conversions**

- **TMax**: \( x = \sum_{i=0}^{w-1} x_i 2^i \)
- **TMin**: \( x = \sum_{i=0}^{w-1} x_i 2^i \)
Unsigned & Signed Integer 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/ negative integer values

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

Can Invert Mappings
- U2B(x) = B2U(x)
  - Bit pattern for signed integer
- T2B(x) = B2T(-x)
  - Bit pattern for two’s complement integer

Mapping Between Signed & Unsigned

Signed (Two’s Complement) → Unsigned
- Maintain Same Bit Pattern

Unsigned → Signed (Two’s Complement)
- Maintain Same Bit Pattern

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

Today: Bits, Bytes, and Integers

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

Mapping Signed ↔ Unsigned

Signed

<table>
<thead>
<tr>
<th>Bits</th>
<th>Signed</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>

Unsigned

<table>
<thead>
<tr>
<th>Bits</th>
<th>Signed</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>9</td>
</tr>
<tr>
<td>1010</td>
<td>10</td>
</tr>
<tr>
<td>1011</td>
<td>11</td>
</tr>
<tr>
<td>1100</td>
<td>12</td>
</tr>
<tr>
<td>1101</td>
<td>13</td>
</tr>
<tr>
<td>1110</td>
<td>14</td>
</tr>
<tr>
<td>1111</td>
<td>15</td>
</tr>
</tbody>
</table>

Relation between Signed & Unsigned

Two’s Complement

<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>-7</td>
</tr>
<tr>
<td>1010</td>
<td>-6</td>
<td>-6</td>
</tr>
<tr>
<td>1011</td>
<td>-5</td>
<td>-5</td>
</tr>
<tr>
<td>1100</td>
<td>-4</td>
<td>-4</td>
</tr>
<tr>
<td>1101</td>
<td>-3</td>
<td>-3</td>
</tr>
<tr>
<td>1110</td>
<td>-2</td>
<td>-2</td>
</tr>
<tr>
<td>1111</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

Large negative weight becomes Large positive weight

x = \begin{cases} 
  x - 2^n & x > 0 \\
  x + 2^n & x < 0 
\end{cases}
Conversion Visualized
- Signed → Unsigned
  - Ordering inversion
  - Negative → Big Positive
  - Warning: Can cause a lot of confusion and bugs in C !!!

2's Complement Range
- Tmax
- Tmin
- UMax
- UMin
- UMax - 1
- TMax
- TMax + 1

Unsigned Range

Review: Signed vs. Unsigned in C
- Constants
  - By default are considered to be signed integers
  - Unsigned if have "U" as suffix
    - 00, 4294967259
- Casting (i.e. conversion)
  - Explicit casting between signed & unsigned is same as U2T and T2U
    - int tx, ty;
      - unsigned ux, uy;
      - tx = (int) ux;
      - uy = (unsigned) ty;
  - Implicit casting also occurs via assignments and procedure calls
    - tx = ux;
    - uy = ty;

Review: Casting Surprises
- Expression Evaluation
  - If there is a mix of 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>Constant1</th>
<th>Constant2</th>
<th>Relation</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0U</td>
<td>=</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>2147483647</td>
<td>-2147483647-1</td>
<td>&lt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647U</td>
<td>-2147483647-1</td>
<td>&lt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>-2</td>
<td>&gt;</td>
<td>signed</td>
</tr>
<tr>
<td>(unsigned)-1</td>
<td>-2</td>
<td>&gt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>(int) 2147483648U</td>
<td>&gt;</td>
<td>signed</td>
</tr>
</tbody>
</table>

Summary
Casting Signed ↔ Unsigned: Basic Rules
- Bit pattern is maintained (i.e. kept the same)
- But reinterpreted
- Can have unexpected effects: adding or subtracting 2^w
- Expression containing unsigned and signed int
  - int is cast to unsigned!!

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

Announcement 1/27/2016
- This Thursday’s recitation sessions will cover both Data Lab assignment, and more C programming language features.
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:
    
    
    
    $X' = \underbrace{x, x, \ldots, x}_{k \text{ copies of MSB}}$

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

Example

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

**Summary:**

**Expanding**, **Truncating:** Basic Rules

- **Expanding** (e.g., short to int)
  - Unsigned: zeros added
  - Signed: sign extension
  - Both yield expected result

- **Truncating** (e.g., unsigned to unsigned short)
  - Unsigned: high-ordered bits are truncated
  - Result reinterpreted
  - Signed: similar to mod
  - For small numbers yields expected behavior

Negation: Complement & Increment

- **Claim:** Following holds for 2's Complement
  - $-x + 1 = x$

- **A Shortcut to Calculate 2's Complement**
  - Observation: $-x + x = 1111\ldots111 = -1$

- **What about $-T_{\text{min}}$?**
  - $T_{\text{min}} = -T_{\text{max}} - 1$
**Unsigned Addition**

- **Operands:** $w$ bits
- **True Sum:** $w+1$ bits
- **Discard Carry:** $w$ bits

- Standard Addition Function
  - Ignores carry output (overflow is not signaled as errors in C language)
- Implements Modular Arithmetic
  $$ s = \text{UAdd}_w(u, v) = (u + v) \mod 2^w $$

**Visualizing (Mathematical) Integer Addition**

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

**Visualizing Unsigned Addition**

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

**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
    ```

**TAdd Overflow**

- **Functionality**
  - True sum requires $w+1$ bits
  - Drop off Most Significant Bit (MSB)
  - Treat remaining bits as 2’s comp. integer

**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
  - If sum $< -2^{w-1}$
    - Becomes positive
Characterizing TAdd

- **Functionality**
  - True sum requires \( w+1 \) bits
  - Drop off MSB
  - Treat remaining bits as 2's comp. integer

\[
\text{TAdd}(u, v) = \begin{cases} 
  u + v + 2^{-w-1} & \text{if } u + v \geq TMax_u \\
  u + v - 2^{-w-1} & \text{otherwise}
\end{cases}
\]

Unsigned Multiplication in C

- **Operands**: \( w \) bits
- **True Product**: \( 2^w \) bits
- **Discard**: \( w \) bits

- **Standard Multiplication Function**
  - Ignores high order \( w \) bits
  - Some of which are different for signed vs. unsigned multiplication
  - Lower bits are the same as unsigned multiplication

- **Range**
  - \( 0 \leq u \cdot v \leq 2^{w-1} \)

Signed Multiplication in C

- **Operands**: \( w \) bits
- **True Product**: \( 2^w \) bits
- **Discard**: \( w \) bits

- **Standard Multiplication Function**
  - Ignores high order \( w \) bits
  - Some of which are different for signed vs. unsigned multiplication
  - Lower bits are the same as unsigned multiplication

- **Range**
  - \( -2^{w-1} \leq u \cdot v \leq 2^{w-1} - 2^{w-3} - 1 \)

Power-of-2 Multiply with Shift

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

\[
\begin{array}{c|c}
\text{Operands: } \text{bits} & \text{Truth Table} \\
\hline
\text{True Product: } 2^{w+k} \text{ bits} & \text{Truth Table} \\
\text{Discard } k \text{ bits: } \text{bits} & \text{Truth Table}
\end{array}
\]

- **Examples**
  - \( u \ll 3 \) gives \( u \cdot 8 \)
  - \( u \ll 5 - u \ll 3 \) gives \( (u \cdot 32) - (u \cdot 8) \)
  - \( u \ll 5 - u \ll 3 \) gives \( u \cdot 24 \)
  - Most machines shift and add faster than multiply
  - Compiler generates this code automatically

Multiplication

- **Computing Exact Product of \( w \)-bit numbers \( x, y \)**
  - Either signed or unsigned

- **Ranges**
  - **Unsigned**: Up to \( 2w \) bits
    - Result range: \( 0 \leq x \cdot y \leq (2^w - 1)^2 = 2^{2w} - 2^{w-1} \cdot 1 \)
  - **Two's complements**
    - \( \text{min (negative)} \): Up to \( 2w-1 \) bits
      - Result range: \( x \cdot y \geq (2^{w-3})^2 = 2^{2w-6} \)
    - \( \text{max (positive)} \): Up to \( 2w \) bits, but only for \( (TMin_u)^2 \)
      - Result range: \( x \cdot y \leq (2^{w-1} - 1)^2 = 2^{2w} \)

- **Maintaining Exact Results**
  - Would need to keep expanding word size with each product computed
  - Done in software by "arbitrary precision" arithmetic packages
### Bits, Bytes, and Integers

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

### Unsigned Power-of-2 Divide with Shift

**Quotient of Unsigned by Power of 2**
- \( x >> k \) gives \( \lceil x / 2^k \rceil \) ("floor" operations)
- Uses logical right shift

<table>
<thead>
<tr>
<th>Operands:</th>
<th>u</th>
<th>( \leq )</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>Division:</td>
<td>( u / 2^k )</td>
<td>( \leq )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>Result:</td>
<td>( \lfloor u / 2^k \rfloor )</td>
<td>( \leq )</td>
<td>( 1 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>u</td>
<td>16219</td>
<td>C4</td>
<td>00110001 00101001</td>
</tr>
<tr>
<td>( x &gt;&gt; k )</td>
<td>16219</td>
<td>C4</td>
<td>00110001 00101001</td>
</tr>
<tr>
<td>( y &gt;&gt; k )</td>
<td>950.8125</td>
<td>58</td>
<td>00 3B 03 B6</td>
</tr>
<tr>
<td>( z &gt;&gt; k )</td>
<td>950.4257811</td>
<td>58</td>
<td>00 3B</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x / 2^k</td>
<td>15219v3</td>
<td>C4</td>
<td>00110011 00101001</td>
</tr>
<tr>
<td>y / 2^k</td>
<td>950.8125</td>
<td>58</td>
<td>00 3B 03 B6</td>
</tr>
<tr>
<td>z / 2^k</td>
<td>950.4257811</td>
<td>58</td>
<td>00 3B</td>
</tr>
</tbody>
</table>

**Correct Power-of-2 Divide**

**Quotient of Negative Number by Power of 2**
- Want \( \lfloor x / 2^k \rfloor \) (Round toward 0, i.e. use "ceiling" operation)
- Compute as \( \lceil (x+2^k-1)/2^k \rceil \)
- In C: \( \lfloor x + (1<<k)-1 \rfloor >> k \)
- Bias dividend toward 0

**Case 1: No rounding**

<table>
<thead>
<tr>
<th>Dividend:</th>
<th>u</th>
<th>( \leq )</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>Division:</td>
<td>( u / 2^k )</td>
<td>( \leq )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>Result:</td>
<td>( \lfloor u / 2^k \rfloor )</td>
<td>( \leq )</td>
<td>( 1 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Divisor:</th>
<th>( 1 / 2^k )</th>
<th>( \leq )</th>
<th>( 1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x / 2^k )</td>
<td>( \leq )</td>
<td>( 1 )</td>
<td></td>
</tr>
</tbody>
</table>

**Biasing has no effect**

### Correct Power-of-2 Divide (Cont.)

**Case 2: Rounding**

<table>
<thead>
<tr>
<th>Dividend:</th>
<th>u</th>
<th>( \leq )</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incremented by 1</td>
<td>( \lfloor u / 2^k \rfloor )</td>
<td>( \leq )</td>
<td>( 1 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Divisor:</th>
<th>( 1 / 2^k )</th>
<th>( \leq )</th>
<th>( 1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x / 2^k )</td>
<td>( \leq )</td>
<td>( 1 )</td>
<td></td>
</tr>
</tbody>
</table>

**Biasing adds 1 to final result**

### Today: Bits, Bytes, and Integers

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

- **Addition:**
  - Normal addition followed by truncate, same operation on bit level
  - **Unsigned:** addition mod $2^n$
  - **Signed:** modified addition mod $2^n$ (result in proper range)
  - Mathematical addition & Possible subtraction of $2^n$
- **Multiplication:** (product has 2w bits)
  - Normal multiplication followed by truncate, same operation on bit level
  - **Unsigned:** multiplication mod $2^n$ (result in proper range)
  - **Signed:** modified multiplication mod $2^n$ (result in proper range)

---

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- **Integers**
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
- Addition, negation, multiplication, divide with shift
- **Summary**
- Data representation in memory, pointers, character strings

---

Byte-Oriented Memory Organization

- **System provides a private address space to each “process”**
  - Think of a process as a program being executed
  - A program can clobber its own data, but not that of others
- **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

---

Machine Words

- Any given machine has a “Word Size”
  - Nominal size of integer-valued data
    - Including addresses
  - Many current machines use 32 bits (4 bytes) words
    - Limits addresses to $2^{32} = 4 GB$
    - Becoming too small for memory-intensive applications
  - Most high-end systems use 64 bits (8 bytes) words
    - Potential address space = 18 X 10^15 bytes (exabytes)
  - Machines support multiple data formats (data types)
    - Fractions or multiples of word size
    - Always integral number of bytes

---

Integer C Puzzle

- TMin and TMax make a good counter example in many cases

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>x &lt; 0</td>
</tr>
<tr>
<td>2.</td>
<td>x &gt;= 0</td>
</tr>
<tr>
<td>3.</td>
<td>x &amp; 7 == 7</td>
</tr>
<tr>
<td>5.</td>
<td>x &gt; -1</td>
</tr>
<tr>
<td>6.</td>
<td>a + x &gt;= 0</td>
</tr>
<tr>
<td>7.</td>
<td>a &gt; 0 &amp; y &gt; 0</td>
</tr>
<tr>
<td>8.</td>
<td>a &gt;= 0</td>
</tr>
<tr>
<td>9.</td>
<td>a &lt; 0</td>
</tr>
</tbody>
</table>

Initialization:

```c
int a = 400;
int y = bar();
unsigned ux = a;
unsigned uy = y;
```
Word-Oriented Memory Organization

- Addresses specify byte locations
  - Address of first byte in a word
  - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)
- Need to distinguish the notion of address vs. data stored in that address

<table>
<thead>
<tr>
<th>Addresses</th>
<th>32-bit Words</th>
<th>64-bit Words</th>
<th>Bytes</th>
<th>Addr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0001</td>
<td>0002</td>
<td>0003</td>
<td></td>
</tr>
<tr>
<td>0004</td>
<td>0005</td>
<td>0006</td>
<td>0007</td>
<td></td>
</tr>
<tr>
<td>0008</td>
<td>0009</td>
<td>0100</td>
<td>0101</td>
<td></td>
</tr>
<tr>
<td>0102</td>
<td>0103</td>
<td>0110</td>
<td>0111</td>
<td></td>
</tr>
<tr>
<td>0112</td>
<td>0113</td>
<td>0114</td>
<td>0115</td>
<td></td>
</tr>
</tbody>
</table>

Byte Ordering

- How should bytes within a multi-byte word be ordered in memory?
- Conventions
  - Big Endian: Sun, PPC Mac, Internet
    - Most significant byte first
  - Little Endian: x86, ARM processors running Android, iOS and Windows
    - Least significant byte first
    - Least significant byte has lowest address, i.e. 0000

Announcement 2/1/2016

- Homework Assignment #1 has been issued today
  - Download from Moodle class web page
  - Due date before class Wednesday 2/10/2016 (check class schedule)
- Data Lab due next Monday 2/8/2016
  - Should start asap, if not yet
  - Check Data Lab Forum for common Q&As, or post your problem there.

Review: Byte Ordering Example

- Big Endian
  - Most significant byte first
  - i.e. Least significant byte has highest address
- Little Endian
  - Least significant byte first
  - i.e. Least significant byte has lowest address
- Example
  - Variable x has 4-byte representation 0x76543210
  - Address given by &x is 0x100

Representing Integers

<table>
<thead>
<tr>
<th>Decimal: 15213</th>
<th>Binary: 0011 1011 0110 1101</th>
<th>Hex: 7 B 6 D</th>
</tr>
</thead>
</table>

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\t0x%.2x\n", start+i, start[i]);
}
```

Print directives:
- %p: print pointer (i.e. address)
- %x: print Hexadecimal
- \%n: tab
- \%s: new line
show_bytes Execution Example

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

**Result (Linux x86-64):**

```plaintext
int a = 15213;
0x7fffb7f71dbc 6d
0x7fffb7f71dbd 3b
0x7fffb7f71dbe 00
0x7fffb7f71dbf 00
```

### Representing Pointers

- Different compilers & machines assign different locations to objects
- Even get different results each time run program

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

### 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 1 has code 0x31
    - String should be null-terminated
      - Final character = 0
      - Invisible
  - **Compatibility**
    - Byte ordering not an issue

| ASCII: American Standard Code for Information Interchange |
| Unicode Consortium |

### Move on to Floating Point Numbers