Bits, Bytes, and Integers

CENG331 - Computer Organization

Instructor:
Murat Manguoglu (Sections 1-2)

Unless otherwise noted adapted from slides of the textbook: http://csapp.cs.cmu.edu/
Assignment + Midterm Dates

- Bomb Lab: 24 October – 2 November (in Lab)
- Midterm I: week of November 4th (TBD)
- Attack Lab: 11-23 November (in Lab)
- Arch. Lab (take-home): 25 November – 8 December
- Midterm II: week of December 9th (TBD)
- Perf. Lab (take-home): 16-29 December
- Final: To be announced by the university
Hello World!

- What happens under the hood?
#include <stdio.h>

int main() {
    printf("Hello World");
}

Compilation of hello.c

- Source program (text)
  - Pre-processor (cpp)
    - hello.c
  - hello.i
    - Modified source program (text)
  - Compiler (ccl)
    - hello.i
  - hello.s
    - Assembly program (text)
    - printf.o
  - Assembler (as)
    - hello.s
    - hello.o
    - Relocatable object programs (binary)
  - Linker (ld)
    - hello.o
    - hello
      - Executable object program (binary)
Preprocessing

#include <stdio.h>

int main() {
    printf("Hello World");
}

cpp hello.c > hello.i
# 1 "hello.c"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "hello.c"
# 1 "/usr/include/stdio.h" 1 3 4

typedef unsigned char __u_char;
typedef unsigned short int __u_short;
typedef unsigned int __u_int;
typedef unsigned long int __u_long;

int main() {
    printf("Hello World");
}

gcc -Wall -S hello.i > hello.s
Assembler

```assembly
.main:
    pushl %ebp
    movl %esp, %ebp
    subl $8, %esp
    andl $-16, %esp
    movl $0, %eax
    addl $15, %eax
    addl $15, %eax
    shr $4, %eax
    sall $4, %eax
    subl %eax, %esp
    subl $12, %esp
    pushl $.LC0
    call printf
    addl $16, %esp
    leave
    ret

.size main, .-main
```

To compile the assembly code, use the command:

```
as hello.s -o hello.o
```

To verify the contents of the object file, use the command:

```
od -a hello.o
```
Finally...

$ gcc hello.o -o hello
$ ./hello
Hello World$
How do you say “Hello World”?
Typical Organization of System

- **CPU**
  - Register file
  - ALU
- **System bus**
- **Memory bus**
- **Main memory**
- **Bus interface**
- **I/O bridge**
- **I/O bus**
- **Expansion slots for other devices such as network adapters**
- **USB controller**
- **Mouse/Keyboard**
- **Graphics adapter**
- **Display**
- **Disk controller**
- **Disk**
- **Disk**
  - hello executable stored on disk
- **Main memory**
User types "hello"

Reading hello command from keyboard

System bus
Memory bus

Expansion slots for other devices such as network adapters
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
Logical Systems in Computers

- **Binary (0 and 1)**
  - Example: computers we are using today

- **Ternary (-1, 0, +1)**
  - Example: Setun ternary computer designed by Nikolay Brusentsov in the Soviet Union (1958 Moscow State University)

- **Decimal arithmetic**

Sources:
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
For example, can count in binary

**Base 2 Number Representation**

- Represent $15213_{10}$ as $11101101101101_2$
- Represent $1.20_{10}$ as $1.0011001100110011[0011]..._2$
- Represent $1.5213 \times 10^4$ as $1.1101101101101101_2 \times 2^{13}$
Encoding Byte Values

- **Byte = 8 bits**
  - Binary: 00000000₂ to 11111111₂
  - Decimal: 0₁₀ to 255₁₀
  - Hexadecimal: 00₁₆ to FF₁₆
    - Base 16 number representation
    - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
    - Write FA1D37B₁₆ in C as
      - 0xFA1D37B
      - 0xFA1D37B

Q: Why a byte is 8-bits?

A: Due to IBM 360 (~1964)

John von Neumann: “Young man, in mathematics you don't understand things. You just get used to them.”

Reply, according to Dr. Felix T. Smith of Stanford Research Institute, to a physicist friend who had said "I'm afraid I don't understand the method of characteristics," as quoted in The Dancing Wu Li Masters: An Overview of the New Physics (1979) by Gary Zukav, Bantam Books, p. 208, footnote.
## 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, shifting
  - 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 \( B=1 \)

<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>

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

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

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

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

Exclusive-Or (Xor)
- \( A \wedge B = 1 \) when either \( A=1 \) or \( B=1 \), but not both

<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>
General Boolean Algebras

- **Operate on Bit Vectors**
  - Operations applied bitwise

  \[
  \begin{array}{c}
  01101001 \\
  \& 01010101 \\
  \hline
  01000001
  \end{array} \\
  \begin{array}{c}
  01101001 \\
  | 01010101 \\
  \hline
  01111101
  \end{array} \\
  \begin{array}{c}
  01101001 \\
  ^ 01010101 \\
  \hline
  00111100
  \end{array} \\
  \begin{array}{c}
  01010101 \\
  ~ 01010101 \\
  \hline
  10101010
  \end{array}
  

- **All of the Properties of Boolean Algebra Apply**
Example: Representing & Manipulating Sets

■ Representation
  ▪ Width w bit vector represents subsets of \{0, \ldots, w-1\}
  ▪ \(a_j = 1 \text{ if } j \in A\)
  ▪ 01101001 \(\{0, 3, 5, 6\}\)
  ▪ 76543210
  ▪ 01010101 \(\{0, 2, 4, 6\}\)
  ▪ 76543210

■ Operations
  ▪ \& Intersect 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)

- ~0x41 → 0xBE
  - ~01000001₂ → 10111110₂
- ~0x00 → 0xFF
  - ~00000000₂ → 11111111₂
- 0x69 & 0x55 → 0x41
  - 01101001₂ & 01010101₂ → 01000001₂
- 0x69 | 0x55 → 0x7D
  - 01101001₂ | 01010101₂ → 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)**
  - !\text{0x41} \Rightarrow \text{0x00}
  - !\text{0x00} \Rightarrow \text{0x01}
  - !!\text{0x41} \Rightarrow \text{0x01}
  - 0x69 && 0x55 \Rightarrow 0x01
  - 0x69 || 0x55 \Rightarrow 0x01
  - p && *p (avoids null pointer access)
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)

Watch out for && vs. & (and || vs. |)... one of the more common oopsies in C programming
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 \( \geq \) word size

<table>
<thead>
<tr>
<th>Argument ( x )</th>
<th>( 01100010 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( &lt;&lt; 3 )</td>
<td>( 00010000 )</td>
</tr>
<tr>
<td>Log. &gt;&gt; 2</td>
<td>( 00011000 )</td>
</tr>
<tr>
<td>Arith. &gt;&gt; 2</td>
<td>( 00011000 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Argument ( x )</th>
<th>( 10100010 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( &lt;&lt; 3 )</td>
<td>( 00010000 )</td>
</tr>
<tr>
<td>Log. &gt;&gt; 2</td>
<td>( 00101000 )</td>
</tr>
<tr>
<td>Arith. &gt;&gt; 2</td>
<td>( 11101000 )</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

Summary
Encoding Integers

Unsigned

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

Two’s Complement

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

C short 2 bytes long

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

Sign Bit

- For 2’s complement, most significant bit indicates sign
  - 0 for nonnegative
  - 1 for negative
Two-complement Encoding Example (Cont.)

\[ x = 15213: \quad 00111011 \quad 01101101 \]
\[ y = -15213: \quad 11000100 \quad 10010011 \]

<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>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>32</td>
</tr>
<tr>
<td>64</td>
<td>1</td>
<td>64</td>
</tr>
<tr>
<td>128</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>256</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>512</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1024</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2048</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4096</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8192</td>
<td>1</td>
<td>0</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 and -15213
Numeric Ranges

- **Unsigned Values**
  - $U_{Min} = 0$
  - $U_{Max} = 2^w - 1$

- **Two’s Complement Values**
  - $T_{Min} = -2^{w-1}$
  - $T_{Max} = 2^{w-1} - 1$

- **Other Values**
  - Minus 1
    - 111...1

### 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>18,446,744,073,709,551,615</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
- $|T_{Min}| = T_{Max} + 1$
  - Asymmetric range
- $U_{Max} = 2 \times T_{Max} + 1$

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

#### Can Invert Mappings
- $U2B(x) = B2U^{-1}(x)$
  - Bit pattern for unsigned integer
- $T2B(x) = B2T^{-1}(x)$
  - Bit pattern for two’s comp integer
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
Mapping Between Signed & Unsigned

Two’s Complement

Unsigned

Maintain Same Bit Pattern

Unsigned

Two’s Complement

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

**T2U** (Signed to Unsigned) and **U2T** (Unsigned to Signed) mappings are shown.
# 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>
Relation between Signed & Unsigned

Two’s Complement → \( T2B \) → \( X \) → \( B2U \) → Unsigned

Maintain Same Bit Pattern

\( w-1 \) \( 0 \)

\( ux \)  

\( x \)

Large negative weight  \( becomes \)  Large positive weight
Conversion Visualized

- **2’s Comp. → Unsigned**
  - Ordering Inversion
  - Negative → Big Positive
Signed vs. Unsigned in C

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

- **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 there is a mix of unsigned and signed in single expression, 
    _signed values implicitly cast to unsigned_
  ▪ Including comparison operations <, >, ==, <=, >=
  ▪ Examples for W = 32: \text{TMIN} = -2,147,483,648, \text{TMAX} = 2,147,483,647

■ Constant

<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>==</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>&gt;</td>
<td>signed</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>2147483648U</td>
<td>&lt;</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
- But reinterpreted
- Can have unexpected effects: adding or subtracting $2^w$

- Expression containing signed and unsigned 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, shifting
- Summary

Representations in memory, pointers, strings
**Sign Extension**

- **Task:**
  - Given a \( w \)-bit signed integer \( x \)
  - Convert it to a \( w+k \)-bit integer with the same value

- **Rule:**
  - Make \( k \) copies of the sign bit:
  
  \[
  X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0
  \]

---

**Diagram**

![Diagram of sign extension](image)
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 int to int)**
  - Unsigned: zeros added
  - Signed: sign extension
  - Both yield expected result

- **Truncating (e.g., unsigned to unsigned short)**
  - Unsigned/signed: bits are truncated
  - Result reinterpreted
  - Unsigned: mod operation
  - Signed: similar to mod
    - For small numbers yields expected behavior
    - For others?
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**
  \[
  s = \text{UAdd}_w(u, v) = u + v \mod 2^w
  \]
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

$\text{UAdd}_4(u, v)$
Two’s Complement Addition

Operands: \( w \) bits

\[
\begin{array}{c}
\text{True Sum: } w+1 \text{ bits} \\
\hline
u \quad \cdots \quad \cdots \\
+ \quad v \quad \cdots \quad \cdots \\
\hline
u + v \quad \cdots \quad \cdots
\end{array}
\]

Discard Carry: \( w \) bits

\[
\text{TAdd}_w(u, v) \quad \cdots \quad \cdots
\]

- **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 MSB
- Treat remaining bits as 2’s comp. integer

**True Sum**
- $0\,111\ldots1$
- $0\,100\ldots0$
- $0\,000\ldots0$
- $1\,011\ldots1$
- $1\,000\ldots0$

**TAdd Result**
- $2^w - 1$
- $0$
- $-2^w$
- $2^w - 1$
- $011\ldots1$
- $000\ldots0$
- $100\ldots0$
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
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: $2 \times w$ bits

Discard $w$ bits: $w$ bits

- **Standard Multiplication Function**
  - Ignores high order $w$ bits

- **Implements Modular Arithmetic**

  $$\text{UMult}_w(u, v) = u \cdot v \mod 2^w$$
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
Power-of-2 Multiply with Shift

**Operation**
- $u << k$ gives $u \times 2^k$
- Both signed and unsigned

**Examples**
- $u << 3 == u \times 8$
- $(u << 5) - (u << 3) == u \times 24$
- 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**
  - $u >> k$ gives $\lfloor u / 2^k \rfloor$
  - Uses logical shift

![Diagram showing division process and operands with binary and hex representations for various division results.]

**Table:**

<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 &gt;&gt; 1</td>
<td>7606.5</td>
<td>1D B6</td>
<td>00011101 10110110</td>
</tr>
<tr>
<td>x &gt;&gt; 4</td>
<td>950.8125</td>
<td>03 B6</td>
<td>00000011 10110110</td>
</tr>
<tr>
<td>x &gt;&gt; 8</td>
<td>59.4257813</td>
<td>00 3B</td>
<td>00000000 00111011</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
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
    
    ```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)
    
    . . .
    ```

    ( unsigned ⇒ always >= 0 )
    
    ( i could be something we did not intent, for example cnt=1 ⇒ i=1-2 = (unsigned)(-1))
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
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
Machine Words

- Any given computer has a “Word Size”
  - Nominal size of integer-valued data
    - and of addresses (i.e. pointers)
  - 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)

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

Little/Big Endian: Jonathan Swift’s book “Gulliver’s Travels” two kinds of religious groups (one prefer to eat their egg starting from the little end, others from the big end)
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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>23</td>
<td>45</td>
<td>67</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>67</td>
<td>45</td>
<td>23</td>
<td>01</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Representing Integers

Decimal: 15213
Binary: 0011 1011 0110 1101
Hex: 3B6D

int A = 15213;

long int C = 15213;

int B = -15213;

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\t0x%.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;
0x7fffb7f71dbc  6d
0x7fffb7f71dbd  3b
0x7fffb7f71dbe  00
0x7fffb7f71dbf  00
```
Representing Pointers

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

Different compilers & machines assign different locations to objects

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";
```
Integer C Puzzles

- \(x < 0\)
- \(u_x \geq 0\)
- \(x \& 7 == 7\)
- \(u_x > -1\)
- \(x > y\)
- \(x \times x \geq 0\)
- \(x > 0 \&\& y > 0\)
- \(x \geq 0\)
- \(x \leq 0\)
- \((x|\neg x)\gg 31 == -1\)
- \(u_x \gg 3 == u_x/8\)
- \(x \gg 3 == x/8\)
- \(x \& (x-1) != 0\)

**Initialization**

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

Connection when

\[ A \& \sim B \mid \sim A \& B \]

\[ = A^B \]
Binary Number Property

Claim

\[ 1 + 1 + 2 + 4 + 8 + \ldots + 2^{w-1} = 2^w \]

\[ \sum_{i=0}^{w-1} 2^i = 2^w \]

- **w = 0:**
  - \( 1 = 2^0 \)

- **Assume true for w-1:**
  - \[ 1 + 1 + 2 + 4 + 8 + \ldots + 2^{w-1} + 2^w = 2^w + 2^w = 2^{w+1} \]
  - \[ = 2^w \]
Code Security Example

```c
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

- Similar to code found in FreeBSD’s implementation of `getpeername`
- There are legions of smart people trying to find vulnerabilities in programs
Typical Usage

/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}

#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, MSIZE);
    printf(“%s\n”, mybuf);
}
Malicious Usage

/* Declaration of library function memcpy */
void *memcpy(void *dest, void *src, size_t n);

/* Kernel memory region holding user-accessible data */
define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
define MSIZE 528
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}

#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, -MSIZE);
    . . .
}

*/
Mathematical Properties

- **Modular Addition Forms an Abelian Group**
  - **Closed** under addition
    \[ 0 \leq \text{UAdd}_w(u, v) \leq 2^w - 1 \]
  - **Commutative**
    \[ \text{UAdd}_w(u, v) = \text{UAdd}_w(v, u) \]
  - **Associative**
    \[ \text{UAdd}_w(t, \text{UAdd}_w(u, v)) = \text{UAdd}_w(\text{UAdd}_w(t, u), v) \]
  - **0 is additive identity**
    \[ \text{UAdd}_w(u, 0) = u \]
  - **Every element has additive inverse**
    - Let \( \text{UComp}_w(u) = 2^w - u \)
    \[ \text{UAdd}_w(u, \text{UComp}_w(u)) = 0 \]
Mathematical Properties of TAdd

- Isomorphic Group to unsigneds with UAdd
  - \( T\text{Add}_w(u, v) = \text{U2T}(U\text{Add}_w(T2U(u), T2U(v))) \)
    - Since both have identical bit patterns

- Two’s Complement Under TAdd Forms a Group
  - Closed, Commutative, Associative, 0 is additive identity
  - Every element has additive inverse

\[
T\text{Comp}_w(u) = \begin{cases} 
-u & u \neq T\text{Min}_w \\
T\text{Min}_w & u = T\text{Min}_w 
\end{cases}
\]
Characterizing TAdd

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

\[
TAdd_w(u, v) = \begin{cases} 
  u + v + 2^w & u + v < TMin_w \\
  u + v & TMin_w \leq u + v \leq TMax_w \\
  u + v - 2^w & TMax_w < u + v
\end{cases}
\]

- **NegOver**: \( u + v < TMin_w \)
- **PosOver**: \( TMax_w < u + v \)
Negation: Complement & Increment

- Claim: Following Holds for 2’s Complement
  \[ \sim x + 1 == -x \]

- Complement
  - Observation: \[ \sim x + x == 1111...111 == -1 \]

  \[
  \begin{array}{c}
  x \ 10011101 \\
  + \sim x \ 01100010 \\
  \hline
  -1 \ 1111111111
  \end{array}
  \]

- Complete Proof?
# Complement & Increment Examples

### $x = 15213$

<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>

### $x = 0$

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00 00</td>
<td>00000000 00000000</td>
</tr>
<tr>
<td>$\sim 0$</td>
<td>-1</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>$\sim 0 + 1$</td>
<td>0</td>
<td>00 00</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>
Code Security Example #2

- SUN XDR library
  - Widely used library for transferring data between machines

\[
\text{void* copy_elements(void *ele_src[], int ele_cnt, size_t ele_size);}\
\]

\[
\text{malloc(ele_cnt * ele_size)}\
\]
void* copy_elements(void *ele_src[], int ele_cnt, size_t ele_size) {
    /*
     * Allocate buffer for ele_cnt objects, each of ele_size bytes
     * and copy from locations designated by ele_src
     */
    void *result = malloc(ele_cnt * ele_size);
    if (result == NULL) {
        /* malloc failed */
        return NULL;
    }
    void *next = result;
    int i;
    for (i = 0; i < ele_cnt; i++) {
        /* Copy object i to destination */
        memcpy(next, ele_src[i], ele_size);
        /* Move pointer to next memory region */
        next += ele_size;
    }
    return result;
}
XDR Vulnerability

malloc(ele_cnt * ele_size)

- **What if:**
  - ele_cnt = $2^{20} + 1$
  - ele_size = 4096 = $2^{12}$
  - Allocation = ??

- **How can I make this function secure?**
C compiler automatically generates shift/add code when multiplying by constant
Compiled Unsigned Division Code

C Function

```c
unsigned long udiv8 (unsigned long x)
{
    return x/8;
}
```

Compiled Arithmetic Operations

- `shrq $3, %rax`
  
  # Logical shift
  
  return x >> 3;

- Uses logical shift for unsigned
- For Java Users
  - Logical shift written as >>>

Signed Power-of-2 Divide with Shift

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

\[
\begin{array}{c|c|c|c}
\text{Division} & \text{Computed} & \text{Hex} & \text{Binary} \\
\hline
y & -15213 & -15213 & \text{C4 93} & 11000100 10010011 \\
y >> 1 & -7606.5 & -7607 & \text{E2 49} & 11100010 01001001 \\
y >> 4 & -950.8125 & -951 & \text{FC 49} & 11111100 01001001 \\
y >> 8 & -59.4257813 & -60 & \text{FF C4} & 11111111 11000100 \\
\end{array}
\]
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: \((x + (1<<k) - 1) \gg k\)
  - Biases dividend toward 0

Case 1: No rounding

Dividend:

\[
\begin{array}{c}
\text{u} \\
+2^k - 1
\end{array}
\]

Divisor:

\[
\begin{array}{c}
\text{u} \\
2^k
\end{array}
\]

\[
\left\lfloor \frac{u}{2^k} \right\rfloor
\]

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

Case 2: Rounding

Dividend:

\[ x + 2^k - 1 \]

Divisor:

\[ x \div 2^k \]

Biasing adds 1 to final result
Compiled Signed Division Code

C Function

```c
long idiv8(long x)
{
    return x/8;
}
```

Compiled Arithmetic Operations

```
testq %rax, %rax
js    L4
L3:
sarq $3, %rax
ret
L4:
addq $7, %rax
jmp   L3
```

Explanation

```
if x < 0
    x += 7;
# Arithmetic shift
return x >> 3;
```

- Uses arithmetic shift for int
- For Java Users
  - Arith. shift written as `>>`
Arithmetic: Basic Rules

- Unsigned ints, 2’s complement ints are isomorphic rings: isomorphism = casting

- Left shift
  - Unsigned/signed: multiplication by $2^k$
  - Always logical shift

- Right shift
  - Unsigned: logical shift, div (division + round to zero) by $2^k$
  - Signed: arithmetic shift
    - Positive numbers: div (division + round to zero) by $2^k$
    - Negative numbers: div (division + round away from zero) by $2^k$

Use biasing to fix
Properties of Unsigned Arithmetic

Unsigned Multiplication with Addition Forms

Commutative Ring

- Addition is commutative group
- Closed under multiplication
  \[ 0 \leq \text{UMult}_w(u, v) \leq 2^w - 1 \]
- Multiplication Commutative
  \[ \text{UMult}_w(u, v) = \text{UMult}_w(v, u) \]
- Multiplication is Associative
  \[ \text{UMult}_w(t, \text{UMult}_w(u, v)) = \text{UMult}_w(\text{UMult}_w(t, u), v) \]
- 1 is multiplicative identity
  \[ \text{UMult}_w(u, 1) = u \]
- Multiplication distributes over addition
  \[ \text{UMult}_w(t, \text{UAdd}_w(u, v)) = \text{UAdd}_w(\text{UMult}_w(t, u), \text{UMult}_w(t, v)) \]
Properties of Two’s Comp. Arithmetic

Isomorphic Algebras
- Unsigned multiplication and addition
  - Truncating to $w$ bits
- Two’s complement multiplication and addition
  - Truncating to $w$ bits

Both Form Rings
- Isomorphic to ring of integers mod $2^w$

Comparison to (Mathematical) Integer Arithmetic
- Both are rings
- Integers obey ordering properties, e.g.,
  \[ u > 0 \quad \Rightarrow \quad u + v > v \]
  \[ u > 0, \ v > 0 \quad \Rightarrow \quad u \cdot v > 0 \]
- These properties are not obeyed by two’s comp. arithmetic
  \[ TMax + 1 =\ TMin \]
  \[ 15213 \times 30426 = -10030 \] (16-bit words)
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</td>
<td>cmpl $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