Bits, Bytes, and Integers

CENG331 - Computer Organization

Adapted from slides of the textbook: http://csapp.cs.cmu.edu/
Hello World!

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

int main() {
    printf("Hello World");
}
```
Compilation of hello.c
# Preprocessing

Preprocessor (cpp)

Source program (text)

Modified source program (text)

```c
#include <stdio.h>

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

```bash
cpp hello.c > hello.i
```

```
#include <stdio.h>

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

```
# 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");
}
```
Compiler

```
# 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");
}
```

```
.gloabl main
@function 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

.ident "GCC: (GNU) 3.4.1"
```

```
.gloabl main
@function 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
```

```
.gloabl main
@function 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
```

```
gcc -Wall -S hello.i > hello.s
```
```
.file "hello.c"
.section
.rodata
.LC0:
.string "Hello World"
.text
.globl main
.type main, @function
main:
  pushl %ebp
  movl %esp, %ebp
  subl $8, %esp
  addl $-16, %esp
  movl $0, %eax
  addl $15, %eax
  shrl $4, %eax
  sal $4, %eax
  subl %eax, %esp
  subl $12, %esp
  pushl $.LC0
  call printf
  addl $16, %esp
  leave
  ret
.main
.size main, 

.note.GNU-stack,"",@progbits
.ident "GCC: (GNU) 3.4.1"
```
Linker

```c
printf.o
```

```c
hello.o
```

```c
linker (ld)
```

`gcc hello.o -o hello`

```c
# hello
```

```c
od -a hello
```

---

Finally...

```sh
$ 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

I/O bus

Expansion slots for other devices such as network adapters

Disk controller

Disk

hello executable stored on disk

USB controller

Graphics adapter

Mouse

Keyboard

Display

PC

Bus interface

System bus

Memory bus
"hello"

User types "hello"

Reading hello command from keyboard

Main memory

System bus

Memory bus

I/O bridge

Bus interface

CPU

Register file

ALU

PC

Bus interface

I/O bridge

I/O bus

Expansion slots for other devices such as network adapters

Disk controller

Disk

Graphics adapter

Display

Keyboard

Mouse

USB controller

User types "hello"
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
For example, can count in binary

- **Base 2 Number Representation**
  - Represent $15_{10}$ as $11101101101101_2$
  - Represent $1.20_{10}$ as $1.0011001100110011[0011]..._2$
  - Represent $1.5213 \times 10^4$ as $1.1101101101101_2 \times 2^{13}$
Encoding Byte Values

- **Byte = 8 bits**
  - Binary 00000000$_2$ to 11111111$_2$
  - Decimal: 0$_{10}$ to 255$_{10}$
  - Hexadecimal 00$_{16}$ to FF$_{16}$
    - Base 16 number representation
    - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
    - Write FA1D37B$_{16}$ in C as
      - 0xFA1D37B
      - 0xfa1d37b

  » Convert in 4-bit groups

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

\[
\begin{array}{c|cc}
\& & 0 & 1 \\
0 & 0 & 0 \\
1 & 0 & 1 \\
\end{array}
\]

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

\[
\begin{array}{c|cc}
\sim & 0 & 1 \\
0 & 1 \\
1 & 0 \\
\end{array}
\]

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

\[
\begin{array}{c|cc}
\mid & 0 & 1 \\
0 & 0 & 1 \\
1 & 1 & 1 \\
\end{array}
\]

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

\[
\begin{array}{c|cc}
\wedge & 0 & 1 \\
0 & 0 & 1 \\
1 & 1 & 0 \\
\end{array}
\]
General Boolean Algebras

- Operate on Bit Vectors
  - Operations applied bitwise

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

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

**Representation**

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

  - \( 01101001 \) \( \{ 0, 3, 5, 6 \} \)
  - \( 76543210 \)

  - \( 01010101 \) \( \{ 0, 2, 4, 6 \} \)
  - \( 76543210 \)

**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)**
  - `~0x41 = 0xBE`
    - `~01000001_2 = 10111110_2`
  - `~0x00 = 0xFF`
    - `~00000000_2 = 11111111_2`
  - `0x69 & 0x55 = 0x41`
    - `01101001_2 & 01010101_2 = 01000001_2`
  - `0x69 | 0x55 = 0x7D`
    - `01101001_2 | 01010101_2 = 01111101_2`
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 due to lazy evaluation)
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:** \( x \ll y \)
  - Shift bit-vector \( x \) left \( y \) positions
    - Throw away extra bits on left
      - Fill with 0’s on right

- **Right Shift:** \( x \gg y \)
  - 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 right

- **Undefined Behavior**
  - Shift amount \(< 0\) or \(\geq\) word size

C, however, has only one right shift operator, \( \gg \). Many C compilers choose which right shift to perform depending on what type of integer is being shifted; often signed integers are shifted using the arithmetic shift, and unsigned integers are shifted using the logical shift.
Swap

```c
void swap(int *x, int *y)
{
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
```

Challenge: Can you code swap function without using a temporary variable?
Hint: Use bitwise XOR (^)
Swapping with Xor

- Bitwise Xor is a form of addition
- With extra property that every value is its own additive inverse
  \[ A \oplus A = 0 \]

```c
void funny(int *x, int *y)
{
    *x = *x ^ *y;    /* #1 */
    *y = *x ^ *y;    /* #2 */
    *x = *x ^ *y;    /* #3 */
}
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>x</strong></td>
<td><strong>y</strong></td>
</tr>
<tr>
<td>Begin</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>A(^\oplus)B</td>
</tr>
<tr>
<td>2</td>
<td>A(^\oplus)B</td>
</tr>
<tr>
<td>3</td>
<td>(A(^\oplus)B)(^\oplus)A = B</td>
</tr>
<tr>
<td>End</td>
<td>B</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**

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

short int x = 15213;
short int y = -15213;
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>3B 6D 00111011 01101101</td>
</tr>
<tr>
<td>y</td>
<td>-15213</td>
<td>C4 93 11000100 10010011</td>
</tr>
</tbody>
</table>

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

short int x = 15213;
short int y = -15213;
Two-complement Encoding Example (Cont.)

\[
\begin{align*}
x &= 15213: & 00111011 & 01101101 \\
y &= -15213: & 11000100 & 10010011
\end{align*}
\]

<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>0</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>0</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>0</td>
</tr>
<tr>
<td>256</td>
<td>1</td>
<td>256</td>
</tr>
<tr>
<td>512</td>
<td>1</td>
<td>512</td>
</tr>
<tr>
<td>1024</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2048</td>
<td>1</td>
<td>2048</td>
</tr>
<tr>
<td>4096</td>
<td>1</td>
<td>4096</td>
</tr>
<tr>
<td>8192</td>
<td>1</td>
<td>8192</td>
</tr>
<tr>
<td>16384</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-32768</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Sum**  \(15213\)  \(-15213\)
Numeric Ranges

- **Unsigned Values**
  - $UMin = 0$
    - 000...0
  - $UMax = 2^w - 1$
    - 111...1
Numeric Ranges

- **Two’s Complement Values**
  - $TMin = -2^{w-1}$
    - 100...0
  - $TMax = 2^{w-1} - 1$
    - 011...1

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

■ Unsigned Values
  - $U_{Min} = 0$
    - 000...0
  - $U_{Max} = 2^w - 1$
    - 111...1

■ Two's Complement Values
  - $T_{Min} = -2^{w-1}$
    - 100...0
  - $T_{Max} = 2^{w-1} - 1$
    - 011...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>$U_{Max}$</td>
<td>65535</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>$T_{Max}$</td>
<td>32767</td>
<td>7F FF</td>
<td>01111111 11111111</td>
</tr>
<tr>
<td>$T_{Min}$</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

- **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

<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>
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

\[ x \xrightarrow{T2B} T2U \xrightarrow{B2U} \]

Maintain Same Bit Pattern

Unsigned

\[ \text{ux} \xrightarrow{T2U} \text{ux} \]

Two’s Complement

\[ \text{ux} \xrightarrow{U2B} U2T \xrightarrow{B2T} \]

Maintain Same Bit Pattern

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>
## 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

Maintain Same Bit Pattern

Unsigned

\[ u_x \]

\[ x \] → \[ T2B \] → \[ T2U \] → \[ B2U \] → \[ u_x \]

Large negative weight becomes Large positive weight

\[ w-1 \] \hspace{1cm} 0

\[ u_x \] \[ + + + \] \[ \cdots \] \[ + + + \]

\[ x \] \[ - + + \] \[ \cdots \] \[ + + + \]
Conversion Visualized

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

2’s Complement Range

0

-1

-2

TMin

TMax

UMax

UMax – 1

TMax + 1

TMax

Unsigned Range

Conversion Visualized

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

2’s Complement Range

0

-1

-2

TMin

TMax

UMax

UMax – 1

TMax + 1

TMax

Unsigned Range
Signed vs. Unsigned in C

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

- **Casting**
  - 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 $W = 32$: $\text{TMIN} = -2,147,483,648$, $\text{TMAX} = 2,147,483,647$

<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 \( w \)-bit signed integer \( x \)
- Convert it to \( w+k \)-bit integer with same value

Rule:
- Make \( k \) copies of sign bit:
- \( X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0 \)
Sign Extension Example

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

```
short int x = 15213;
int ix = (int) x;
short int y = -15213;
int iy = (int) y;
```

<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>ix</td>
<td>15213</td>
<td>00000000 00000000 00111011 01101101</td>
</tr>
<tr>
<td>y</td>
<td>-15213</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>iy</td>
<td>-15213</td>
<td>11111111 11111111 11000100 10010011</td>
</tr>
</tbody>
</table>
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
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

\[
\begin{array}{c}
\text{\( u \)} \\
\text{\( + \)} \\
\text{\( v \)} \\
\hline
\text{UAdd}_w(u, v)
\end{array}
\]

True Sum: \( w+1 \) bits

\[
\text{\( u + v \)}
\]

Discard Carry: \( w \) bits

\[
\text{\( \quad \)}
\]

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

- **Implements Modular Arithmetic**

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

\[
\text{UAdd}_w(u, v) = \begin{cases} 
  u + v & u + v < 2^w \\
  u + v - 2^w & u + v \geq 2^w 
\end{cases}
\]
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

![Diagram showing UAdd function](image)
Two’s Complement Addition

Operands: \( w \) bits

\[
\begin{array}{c}
  u \\
  + v \\
  u + v \\
  \text{TAdd}_w(u, v)
\end{array}
\]

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

**True Sum**

- \( 0\ \text{111…1} \)
- \( 0\ \text{100…0} \)
- \( 0\ \text{000…0} \)
- \( 1\ \text{011…1} \)
- \( 1\ \text{000…0} \)

**TAdd Result**

- \( 2^w - 1 \)
- \( 2^w - 1 - 1 \)
- \( 0 \)
- \( -2^w \)
- \( -2^w - 1 \)

- PosOver
- NegOver
Visualizing 2’s Complement Addition

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

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

- **Goal: Computing Product of** $w$-**bit numbers** $x, y$
  - Either signed or unsigned
- **But, exact results can be bigger than** $w$ **bits**
  - Unsigned: up to $2w$ bits
    - Result range: $0 \leq x \times y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
  - Two’s complement min (negative): Up to $2w-1$ bits
    - Result range: $x \times y \geq (-2^{w-1}) \times (2^{w-1}-1) = -2^{2w-2} + 2^{w-1}$
  - 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: \( 2^w \) 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**
  - \( u \ll k \) gives \( u \times 2^k \)
  - Both signed and unsigned

  Operands: \( w \) bits

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

  True Product: \( w+k \) bits

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

  \[
  \begin{array}{c}
  UMult_w(u, 2^k) \\
  \end{array}
  \]

  \[
  \begin{array}{c}
  TMult_w(u, 2^k) \\
  \end{array}
  \]

- **Examples**
  - \( u \ll 3 \equiv u \times 8 \)
  - \( (u \ll 5) - (u \ll 3) \equiv u \times 24 \)
  - Most machines shift and add faster than multiply
    - Compiler generates this code automatically
Compiled Multiplication Code

C Function

```c
long mul12(long x)
{
    return x*12;
}
```

Compiled Arithmetic Operations

```
leaq (%rax,%rax,2), %rax
salq $2, %rax
```

Explanation

```
t <- x+x*2
return t << 2;
```

- C compiler automatically generates shift/add code when multiplying by constant
Unsigned Power-of-2 Divide with Shift

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

### Operands:

\[
\begin{array}{c|c|c}
\times & 15213 & 15213 \\
\times \gg 1 & 7606.5 & 7606 \\
\times \gg 4 & 950.8125 & 950 \\
\times \gg 8 & 59.4257813 & 59 \\
\end{array}
\]

### Computed

\[
\begin{array}{c|c|c|c}
\times & 15213 & 15213 & 3B 6D \\
\times \gg 1 & 7606.5 & 7606 & 1D B6 \\
\times \gg 4 & 950.8125 & 950 & 03 B6 \\
\times \gg 8 & 59.4257813 & 59 & 00 3B \\
\end{array}
\]

### Hex

\[
\begin{array}{c|c|c|c}
\times & 00111011 & 01101101 \\
\times \gg 1 & 00011101 & 10110110 \\
\times \gg 4 & 00000011 & 10110110 \\
\times \gg 8 & 00000000 & 00111011 \\
\end{array}
\]
Compiled Unsigned Division Code

C Function

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

Compiled Arithmetic Operations

```
shrq $3, %rax
```

Explanation

```
# 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**
  - $x \gg k$ gives $\lfloor x / 2^k \rfloor$
  - Uses arithmetic shift
  - Rounds wrong direction when $u < 0$

Operands:

$\begin{array}{c}
\frac{\mathbf{x}}{2^k}
\end{array}$

Division:

$x / 2^k$

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

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 01010011</td>
</tr>
<tr>
<td>y &gt;&gt; 1</td>
<td>-7606.5</td>
<td>E2 49</td>
<td>11100010 01001001</td>
</tr>
<tr>
<td>y &gt;&gt; 4</td>
<td>-950.8125</td>
<td>FC 49</td>
<td>11111100 01001001</td>
</tr>
<tr>
<td>y &gt;&gt; 8</td>
<td>-59.4257813</td>
<td>FF C4</td>
<td>11111111 11000100</td>
</tr>
</tbody>
</table>
Correct Power-of-2 Divide

Quotient of Negative Number by Power of 2

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

Case 1: No rounding

Dividend:

\[
\begin{array}{c}
\text{Dividend:} \\
1 \ldots 0 \ldots 0 \\
+2^k-1 \\
\hline
1 \ldots 1 \ldots 1 \\
\end{array}
\]

Divisor:

\[
\begin{array}{c}
\text{Divisor:} \\
\hline
u / 2^k \\
1 \ldots 1 \ldots 1
\end{array}
\]

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

Case 2: Rounding

Dividend: \( x \)

\[
\begin{array}{c}
\hline
x & \ldots & \ldots & \ldots & \ldots \\
\hline
+2^k - 1 & 0 & \ldots & 0 & 0 & 1 & \ldots & 1 & 1 \\
\hline
\end{array}
\]

Divisor: \( l \)

\[
\begin{array}{c}
\hline
l & 2^k & 0 & \ldots & 0 & 1 & 0 & \ldots & 0 & 0 \\
\hline
\lfloor x / 2^k \rfloor & 1 & \ldots & 1 & 1 & 1 & \ldots \\
\hline
\end{array}
\]

Biasing adds 1 to final result

Incremented by 1
Compiled Signed Division Code

C Function

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

Compiled Arithmetic Operations

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

Explanation

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

- Uses arithmetic shift for int
- For Java Users
  - Arith. shift written as `>>`
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)
Multiplication

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

- **Ranges**
  - Unsigned: $0 \leq x \times y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
    - Up to $2w$ bits
  - Two’s complement min: $x \times y \geq (-2^{w-1}) \times (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$
    - Up to $2w-1$ bits
  - Two’s complement max: $x \times y \leq (-2^{w-1})^2 = 2^{2w-2}$
    - Up to $2w$ bits, but only for $(TMin_w)^2$

- **Maintaining Exact Results**
  - Would need to keep expanding word size with each product computed
  - Done in software by “arbitrary precision” arithmetic packages
float sum_elements(float a[], unsigned length) {
    int i;
    float result = 0;

    for (i = 0; i <= length - 1; i++)
        result += a[i];
    return result;
}

- What’s the bug?
- When will it return an error or produce a wrong result?
- How can you fix it?
Buggy code - 2

/* Prototype for library function strlen */
size_t strlen(const char *s)

/* whether string s is longer than string t */
int strlonger(char *s, char *t){
  return strlen(s)-strlen(t) >0;
}

■ What’s the bug?
■ When will it return an error or produce a wrong result?
■ How can you fix it?
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**
  
  ```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
    - $0 - 1 \rightarrow UMax$
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
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
  - \( T\text{Add}_w(u, v) = U2T(U\text{Add}_w(T2U(u), T2U(v))) \)
    - 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

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

Positive Overflow

Negative Overflow

\[
\text{NegOver} \quad u + v < 0
\]

\[
\text{PosOver} \quad u + v > 0
\]
Negation: Complement & Increment

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

- Complement
  - Observation: \[ \sim x + x = 1111\ldots111 = -1 \]

\[
\begin{array}{c}
x \quad 10011101 \\
+ \quad \sim x \quad 01100010 \\
\hline
-1 \quad 1111111111
\end{array}
\]
## 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>0</td>
<td>00 00 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>0</td>
<td>00 00 00000000 00000000</td>
</tr>
</tbody>
</table>
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
    \[
    T_{Max} + 1 = T_{Min}
    \]
    \[
    15213 \times 30426 = -10030 \quad (16\text{-bit words})
    \]
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
  - 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 $18.4 \times 10^{15}$
  - 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>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>0004</td>
<td></td>
<td>0001</td>
<td>0001</td>
</tr>
<tr>
<td>0008</td>
<td></td>
<td>0002</td>
<td>0002</td>
</tr>
<tr>
<td>0012</td>
<td></td>
<td>0003</td>
<td>0003</td>
</tr>
<tr>
<td></td>
<td>0004</td>
<td>0004</td>
<td>0004</td>
</tr>
<tr>
<td></td>
<td>0008</td>
<td>0005</td>
<td>0005</td>
</tr>
<tr>
<td></td>
<td>0012</td>
<td>0006</td>
<td>0006</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0007</td>
<td>0007</td>
</tr>
<tr>
<td></td>
<td>0000</td>
<td>0008</td>
<td>0008</td>
</tr>
<tr>
<td></td>
<td>0004</td>
<td>0009</td>
<td>0009</td>
</tr>
<tr>
<td></td>
<td>0008</td>
<td>0010</td>
<td>0010</td>
</tr>
<tr>
<td></td>
<td>0012</td>
<td>0011</td>
<td>0011</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0013</td>
<td>0013</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0014</td>
<td>0014</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0015</td>
<td>0015</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

<table>
<thead>
<tr>
<th>Bits</th>
<th>32-bit Words</th>
<th>64-bit Words</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr</td>
<td>= 0000</td>
<td>= 0000</td>
</tr>
<tr>
<td>Addr</td>
<td>= 0004</td>
<td></td>
</tr>
<tr>
<td>Addr</td>
<td>= 0008</td>
<td>= 0008</td>
</tr>
<tr>
<td>Addr</td>
<td>= 0012</td>
<td></td>
</tr>
<tr>
<td>Addr</td>
<td>= 0000</td>
<td></td>
</tr>
<tr>
<td>Addr</td>
<td>= 0001</td>
<td></td>
</tr>
<tr>
<td>Addr</td>
<td>= 0002</td>
<td></td>
</tr>
<tr>
<td>Addr</td>
<td>= 0003</td>
<td></td>
</tr>
<tr>
<td>Addr</td>
<td>= 0004</td>
<td></td>
</tr>
<tr>
<td>Addr</td>
<td>= 0005</td>
<td></td>
</tr>
<tr>
<td>Addr</td>
<td>= 0006</td>
<td></td>
</tr>
<tr>
<td>Addr</td>
<td>= 0007</td>
<td></td>
</tr>
<tr>
<td>Addr</td>
<td>= 0008</td>
<td></td>
</tr>
<tr>
<td>Addr</td>
<td>= 0009</td>
<td></td>
</tr>
<tr>
<td>Addr</td>
<td>= 0010</td>
<td></td>
</tr>
<tr>
<td>Addr</td>
<td>= 0011</td>
<td></td>
</tr>
<tr>
<td>Addr</td>
<td>= 0012</td>
<td></td>
</tr>
<tr>
<td>Addr</td>
<td>= 0013</td>
<td></td>
</tr>
<tr>
<td>Addr</td>
<td>= 0014</td>
<td></td>
</tr>
<tr>
<td>Addr</td>
<td>= 0015</td>
<td></td>
</tr>
</tbody>
</table>
The origin of the word Endian from Wikipedia

- *Big-endian* and *little-endian* were taken from Jonathan Swift's satiric novel *Gulliver's Travels*.

- Gulliver finds two groups of people in Lilliput and Blefuscu conflict over which end of an egg to crack.
Byte Ordering Example

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

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

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

int A = 15213;

long int C = 15213;

int B = -15213;

Two’s complement representation

Decimal: 15213
Binary: 0011 1011 0110 1101
Hex: 3 B 6 D
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

```
char S[6] = "18213";
```
Integer C Puzzles

True or False?

• \( x < 0 \) \( \Rightarrow \ (x \times 2) < 0 \)
• \( ux \geq 0 \)
• \( x \& 7 == 7 \) \( \Rightarrow (x \ll 30) < 0 \)
• \( ux > -1 \)
• \( x > y \) \( \Rightarrow -x < -y \)
• \( x \times x \geq 0 \)
• \( x > 0 \&\& y > 0 \) \( \Rightarrow x + y > 0 \)
• \( x \geq 0 \) \( \Rightarrow -x \leq 0 \)
• \( x \leq 0 \) \( \Rightarrow -x \geq 0 \)
• \( (x|\neg x)\gg 31 == -1 \)
• \( ux \gg 3 == ux/8 \)
• \( x \gg 3 == x/8 \)
• \( x \& (x-1) != 0 \)

Initialization

```c
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```
Integer C Puzzles

• $x < 0 \implies ((x*2) < 0)$

True or False?

Initialization

```c
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```
Integer C Puzzles

• ux >= 0

True or False?

Initialization

```c
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```
Integer C Puzzles

• \( x \& 7 == 7 \quad \Rightarrow \quad (x\ll30) < 0 \)

True or False?

Initialization

```c
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```
Integer C Puzzles

- $ux > -1$

True or False?

Initialization

```c
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```
Integer C Puzzles

- $x > y \implies -x < -y$

True or False?

Initialization

```c
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```
Integer C Puzzles

• \( x \times x \geq 0 \)

True or False?

Initialization

```
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```
Integer C Puzzles

\[ x > 0 \land y > 0 \implies x + y > 0 \]

True or False?

Initialization

```c
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```
Integer C Puzzles

- \( x \geq 0 \) \quad \implies \quad -x \leq 0

True or False?

Initialization

```c
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```
Integer C Puzzles

• \( x \leq 0 \)  \( \implies \) \( -x \geq 0 \)

True or False?

Initialization

```
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```
Integer C Puzzles

• $(x \mid -x) \gg 31 == -1$

True or False?

Initialization

```c
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```
Integer C Puzzles

• \(ux >> 3 == ux / 8\)

True or False?

Initialization

```c
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```
Integer C Puzzles

• $x \gg 3 = x / 8$

True or False?

Initialization

```c
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```
Integer C Puzzles

• \( x \& (x-1) \neq 0 \)

True or False?

Initialization

```c
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```
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);
}
/* 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);
    . . .
}
Code Security Example #2

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

```c
void* copy_elements(void *ele_src[], int ele_cnt, size_t ele_size);
```

Diagram:
```
void* copy_elements(void *ele_src[], int ele_cnt, size_t ele_size);
```
XDR Code

```c
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;
}
```
Left-over slides
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

\[ A \land \neg B \lor \neg A \land B = A^\lor B \]
Binary Number Property

Claim

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

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

- **w = 0:**
  - 1 = 2^0

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