Bits, Bytes, and Integers

Spring, 2013
Euiseong Seo
(euiseong@skku.edu)

This Powerpoint slides are modified from its original version available at http://www.cs.cmu.edu/afs/cs/academic/class/15213-s09/www/lectures/ppt-sources/
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
Binary Representations
**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 $\text{FA1D37B}_{16}$ in C as
      - 0xFA1D37B
      - 0xfa1d37b

### Hex / Decimal / Binary Table

<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>
Programs refer to virtual addresses

- Conceptually very large array of bytes
- Actually implemented with hierarchy of different memory types
- System provides address space private to particular “process”
  - Program being executed
  - Program can clobber its own data, but not that of others

Compiler + run-time system control allocation

- Where different program objects should be stored
- All allocation within single virtual address space
Machine has “Word Size”

- Nominal size of integer-valued data
  - Including addresses
- Most current machines use 32 bits (4 bytes) words
  - Limits addresses to 4GB
  - Becoming too small for memory-intensive applications
- High-end systems use 64 bits (8 bytes) words
  - Potential address space ≈ 1.8 X 10^{19} bytes
  - x86-64 machines support 48-bit addresses: 256 Terabytes
- Machines support multiple data formats
  - Fractions or multiples of word size
  - Always integral number of bytes
Addresses specify byte locations

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

- Computer and compiler support multiple data formats
  - Using different ways to encode data
    - Integers and floating point
  - Using different lengths
## Data Representations

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Typical 32-bit</th>
<th>Intel IA32</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>4</td>
<td>8</td>
</tr>
<tr>
<td>long long</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>8</td>
<td>10/12</td>
<td>10/16</td>
</tr>
<tr>
<td>pointer</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>
Byte Ordering

A multi-byte object is stored as a contiguous sequence of bytes
  - With a address of the object given by the smallest address of the bytes

How should bytes within a multi-byte word be ordered in memory?

Conventions
  - Big Endian: Sun, PPC Mac, Internet
    - Least significant byte has highest address
  - Little Endian: x86
    - Least significant byte has lowest address
**Byte Ordering Example**

- **Big Endian**
  - Least significant byte has highest address
- **Little Endian**
  - Least significant byte has lowest address
- **Example**
  - Variable x has 4-byte representation `0x01234567`
  - Address given by `&x` is `0x100`

Big Endian

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

Little Endian

<table>
<thead>
<tr>
<th>Address</th>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bytes</td>
<td>67</td>
<td>45</td>
<td>23</td>
<td>01</td>
</tr>
</tbody>
</table>
Reading Byte-Reversed Listings

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

- Example Fragment

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Code</th>
<th>Assembly Rendition</th>
</tr>
</thead>
<tbody>
<tr>
<td>8048365:</td>
<td>5b</td>
<td>pop %ebx</td>
</tr>
<tr>
<td>8048366:</td>
<td>81 c3 ab 12 00 00</td>
<td>add $0x12ab,%ebx</td>
</tr>
<tr>
<td>804836c:</td>
<td>83 bb 28 00 00 00 00</td>
<td>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
Code to print byte representation of data

- Textbook Figure 2.4 at page 42
- Casting pointer to **unsigned char** * creates byte array

```c
typedef unsigned char *pointer;

void show_bytes(pointer start, int len){
    int i;
    for (i = 0; i < len; i++)
        printf("%p\t0x%.2x\n", start+i, start[i]);
    printf("\n");
}
```
ShowBytes Execution Example

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

Result (Linux):

```plaintext
int a = 15213;
0x11ffffffcb8 0x6d
0x11ffffffcb9 0x3b
0x11ffffffcba 0x00
0x11ffffffcbb 0x00
```
Representing Integers

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

int A = 15213;

long int C = 15213;

int B = -15213;

Two’s complement representation
Different compilers & machines assign different locations to objects
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] = "18243";

<table>
<thead>
<tr>
<th>Linux/Alpha</th>
<th>Sun</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td>38</td>
<td>38</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>34</td>
<td>34</td>
</tr>
<tr>
<td>33</td>
<td>33</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>
Encode Program as Sequence of Instructions

- Each simple operation
  - Arithmetic operation
  - Read or write memory
  - Conditional branch

- Instructions encoded as bytes
  - Alpha’s, Sun’s, Mac’s use 4 byte instructions
    - Reduced Instruction Set Computer (RISC)
  - PC’s use variable length instructions
    - Complex Instruction Set Computer (CISC)

- Different instruction types and encodings for different machines
  - Most code not binary compatible

Programs are Byte Sequences Too!
For this example, Alpha & Sun use two 4-byte instructions
- Use differing numbers of instructions in other cases

PC uses 7 instructions with lengths 1, 2, and 3 bytes
- Same for NT and for Linux
- NT / Linux not fully binary compatible

```
int sum(int x, int y)
{
    return x+y;
}
```

Different machines use totally different instructions and encodings
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
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|B = 1 when either A=1 or B=1

<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>1</td>
</tr>
</tbody>
</table>

**not**

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

**exclusive-or (xor)**

A^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

\[
\begin{align*}
01101001 & \quad 01101001 & \quad 01101001 & \quad 01101001 \\
& \quad 01010101 & \quad \mid 01010101 & \quad ^\ 01010101 & \quad \sim 01010101 \\
\hline
01000001 & \quad 01111101 & \quad 00111100 & \quad 10101010
\end{align*}
\]

- All of the Properties of Boolean Algebra Apply
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
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)
## 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 right

### Example

<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>

- **Undefined Behavior**
  - Shift amount < 0 or \( \geq \) word size
Cool Stuff with XOR

- Bitwise xor is a form of addition.
- With an extra property that every value is its own additive inverse.
  - $A ^ A = 0$

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

<table>
<thead>
<tr>
<th></th>
<th>*x</th>
<th>*y</th>
</tr>
</thead>
<tbody>
<tr>
<td>Begin</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>A^B</td>
<td>B</td>
</tr>
<tr>
<td>2</td>
<td>A^B</td>
<td>(A^B)^B = A</td>
</tr>
<tr>
<td>3</td>
<td>(A^B)^A = B</td>
<td>A</td>
</tr>
<tr>
<td>End</td>
<td>B</td>
<td>A</td>
</tr>
</tbody>
</table>
Summary

- It’s all about bits & bytes
  - Numbers
  - Programs
  - Text
- Different machines follow different conventions
  - Word size
  - Byte ordering
  - Representations and encoding
- Boolean algebra is mathematical basis
  - Basic form encodes “false” as 0, “true” as 1
  - General form like bit-level operations in C
    • Good for representing & manipulating sets
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
**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 \( y = -15213; \)

**C short 2 bytes long**

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

<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>
### Encoding Example

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

<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>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>64</td>
<td>1</td>
<td>0</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**

<table>
<thead>
<tr>
<th></th>
<th>15213</th>
<th>-15213</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sum</strong></td>
<td>15213</td>
<td>-15213</td>
</tr>
</tbody>
</table>
### Numeric Ranges

- **Unsigned Values**
  - **UMin** = 0
    - 000...0
  - **UMax** = \(2^w - 1\)
    - 111...1

- **Two’s Complement Values**
  - **TMin** = \(-2^{w-1}\)
    - 100...0
  - **TMax** = \(2^{w-1} - 1\)
    - 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>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
- \(|\text{TMin} | = \text{TMax} + 1
  - Asymmetric range
- \(\text{UMax} = 2 \times \text{TMax} + 1

### C Programming
- `#include <limits.h>`
- Declares constants, e.g.,
  - `ULONG_MAX`
  - `LONG_MAX`
  - `LONG_MIN`
- Values platform specific
- **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>
**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**
**Mapping Between Signed & Unsigned**

- **Two's Complement**
  - Maintain Same Bit Pattern
  - Mappings between unsigned and two’s complement numbers
    - keep bit representations and **reinterpret**

- **Unsigned**
  - Maintain Same Bit Pattern
# 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** and **U2T** indicate the conversion between signed and unsigned numbers.
# 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>

= +/- 16
**Relation between Signed & Unsigned**

Two’s Complement

\[ ux = \begin{cases} x & x \geq 0 \\ x + 2^w & x < 0 \end{cases} \]

Large negative weight becomes Large positive weight

Maintain Same Bit Pattern
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
  - int tx, ty;
  - unsigned ux, uy;
  - tx = (int) ux;
  - uy = (unsigned) ty;

- Implicit casting also occurs via assignments and procedure calls
  - tx = ux;
  - uy = ty;
# 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 `<`, `>`, `==`, `<=`, `>=`
- Example: \( w = 32; \text{TMIN} = -2,147,483,648; \text{TMAX} = 2,147,483,647 \)

<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>-2147483648</td>
<td>&gt;</td>
<td>signed</td>
</tr>
<tr>
<td>2147483647U</td>
<td>2147483648</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>
Casting 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
**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

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

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>15123</td>
<td>3B 6D</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>ix</td>
<td>15123</td>
<td>00 00 3B 6D</td>
<td>00000000 00000000 00111011 01101101</td>
</tr>
<tr>
<td>y</td>
<td>-15123</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>iy</td>
<td>-15123</td>
<td>FF FF C4 93</td>
<td>11111111 11111111 11000100 10010011</td>
</tr>
</tbody>
</table>
Prove correctness by induction on $k$

- Induction step
  - Extending by single bit maintains value

Key observation: $-2^w = -2^{w+1} + 2^w$
Truncating Numbers

- Truncating a number can alter its value
  - A form of overflow
- For an unsigned number of $x$
  - Result of truncating it to $k$ bits is equivalent to computing $x \mod 2^k$

```c
int x = 50323;
short int ux = (short) x; // -15213
int y = sx; // -15213
```

\[ B2U_k ([x_k, x_{k-1}, \ldots, x_0]) = B2U_w ([x_w, x_{w-1}, \ldots, x_0]) \mod 2^k \]
\[ B2T_k ([x_k, x_{k-1}, \ldots, x_0]) = U2T_k (B2U_w ([x_w, x_{w-1}, \ldots, x_0]) \mod 2^k) \]
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
**Advice on Singed and Unsigned**

- Implicit conversion of singed to unsigned
  - Can lead to error or vulnerabilities
- Be careful when using unsigned numbers
  - Java supports only signed integers
  - `>>` : arithmetic shift
  - `>>>` : logical shift
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
Claim: following holds for 2’s complement

\[ \sim x + 1 = -x \]

Complement

Observation: \[ \sim x + x = 1111...111_2 = -1 \]

<table>
<thead>
<tr>
<th>x</th>
<th>10011101</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ \sim x</td>
<td>01100010</td>
</tr>
<tr>
<td>-1</td>
<td>11111111</td>
</tr>
</tbody>
</table>

Increment

\[ \sim x + x + (-x + 1) = -1 + (-x + 1) \]

\[ \sim x + 1 = -x \]
## 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>$-x$</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>
**UNSIGNED ADDITION**

- Standard addition function
  - Ignores CARRY output
- Implements modular arithmetic
  \[ s = UAdd_w(u,v) = u + v \mod 2^w \]

\[
UAdd_w(u,v) = \begin{cases} 
  u + v & u + v < 2^w \\
  u + v - 2^w & u + v \geq 2^w 
\end{cases}
\]
**Visualizing 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
Wraps Around

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

True Sum

$2^{w+1}$

$2^w$

0

Modular Sum

Overflow

$UAdd_4(u, v)$
**Mathematical Properties of UAdd**

- Modular Addition Forms an *Abelian Group*
  - **Closed** under addition
    
    \[ 0 \leq UAdd_w(u,v) \leq 2^w - 1 \]
  - **Commutative**
    
    \[ UAdd_w(u,v) = UAdd_w(v,u) \]
  - **Associative**
    
    \[ UAdd_w(t, UAdd_w(u,v)) = UAdd_w(UAdd_w(t,u), v) \]
  - **0 is additive identity**
    
    \[ UAdd_w(u, 0) = u \]
  - **Every element has additive inverse**
    
    - Let
      
      \[ UComp_w(u) = 2^w - u \]
      
      \[ UAdd_w(u, UComp_w(u)) = 0 \]
### Two’s Complement Addition

<table>
<thead>
<tr>
<th>Operands: $w$ bits</th>
<th>$u$ + $v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>True Sum: $w+1$ bits</td>
<td>$u + v$</td>
</tr>
<tr>
<td>Discard Carry: $w$ bits</td>
<td>TAdd$_w(u, v)$</td>
</tr>
</tbody>
</table>

- **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 $s == t$
True sum requires \( w+1 \) bits
Drop off MSB
Treat remaining bits as 2’s complement integer
Visualizing 2’s Complement Addition

- Values
  - 4-bit two’s comp.
  - Range from -8 to +7
- Wraps around
  - If sum $\geq 2^{w-1}$
    - Becomes negative
    - At most once
  - If sum $< -2^{w-1}$
    - Becomes positive
    - At most once
**Characterizing TAdd**

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

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

- **Positive Overflow**
  - \( u > 0 \) and \( v < 0 \)

- **Negative Overflow**
  - \( u < 0 \) and \( v > 0 \)
Mathematical Properties of TAdd

- Isomorphisic group to unsigned with UAdd
  \[ TAdd(u, v) = U2T(UAdd(T2U(u), T2U(v))) \]
  - 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
  - Let
    \[ TComp(u) = U2T(UComp(T2U(u))) \]
    \[ TAdd(u, TComp(u)) = 0 \]

\[
TComp(u) = \begin{cases} 
-u & u \neq TMin_w \\
TMin_w & u = TMin_w 
\end{cases}
\]
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 $(\text{TMin}_w)^2$

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

<table>
<thead>
<tr>
<th>Operands: $w$ bits</th>
<th>$u$</th>
<th>$v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u \cdot v$</td>
<td>$\star v$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>True Product: $2^w$ bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\cdots$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Discard: $w$ bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\cdots$</td>
</tr>
</tbody>
</table>

**UMult$_w(u, v)$**

- **Standard multiplication function**
  - Ignores high order $w$ bits
- **Implements modular arithmetic**
  - $\text{UMult}_w(u, v) = u \cdot v \mod 2^w$
**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 of data transfer and allocation]

```c
malloc(ele_cnt*ele_size)
```
```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;
}
```
What if:

- \( \text{ele}_\text{cnt} = 2^{20} + 1 \)
- \( \text{ele}_\text{size} = 4096 = 2^{12} \)
- Allocation = ??

How can I make this function secure?

\[
\text{malloc}(\text{ele}_\text{cnt} \times \text{ele}_\text{size})
\]
**Signed Multiplication in C**

<table>
<thead>
<tr>
<th>Operands: ( w ) bits</th>
<th>( u \times v \times v )</th>
<th>Discard: ( w ) bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>True Product: ( 2w ) bits</td>
<td>( u \times v \times v )</td>
<td>( \text{TMult}_{w}(u, v) )</td>
</tr>
</tbody>
</table>

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

- **Unsigned multiplication**
  
  ```
  unsigned ux = (unsigned) x;
  unsigned uy = (unsigned) y;
  unsigned up = ux * uy
  ```
  - Truncates product to w-bit number
    ```
    up = UMultw(ux, uy)
    ```
  - Modular arithmetic
    ```
    up = ux * uy mod 2^w
    ```

- **Two’s Complement Multiplication**
  
  ```
  int x, y;
  int p = x * y;
  ```
  - Compute exact product of two w-bit numbers x, y
  - Truncate result to w-bit number
    ```
    p = TMultw(x, y)
    ```
**Power-of-2 Multiply with Shift**

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

<table>
<thead>
<tr>
<th>Operands: $w$ bits</th>
<th>True Product: $w+k$ bits</th>
<th>Discard $k$ bits: $w$ bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u$</td>
<td>$u \times 2^k$</td>
<td>UMult$_w(u, 2^k)$</td>
</tr>
</tbody>
</table>

- **Examples**
  - $u \ll 3 = u \times 8$
  - $u \ll 5 - u \ll 3 = u \times 24$
  - Most machines shift and add faster than multiply

  - Compiler generates this code automatically
C compiler automatically generates shift/add code when multiplying by constant.

C Function

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

Compiled Arithmetic Operations

- `leal (%eax,%eax,2), %eax`
- `sall $2, %eax`

Explanation

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

\[
\begin{array}{cccccc}
\text{u} & \cdots & \cdots & \cdots & \cdots \\
\end{array}
\]

\[
\begin{array}{cccccc}
\text{u} / 2^k & 0 & \cdots & 0 & 1 & 0 & \cdots & 0 & 0 \\
\end{array}
\]

**Division:**

\[
\begin{array}{cccccc}
\text{u} / 2^k & 0 & \cdots & 0 & 0 & \cdots \cdots \\
\end{array}
\]

**Result:**

\[
\begin{array}{cccccc}
\lfloor u / 2^k \rfloor & 0 & \cdots & 0 & 0 & \cdots \cdots \\
\end{array}
\]

**Table:**

<table>
<thead>
<tr>
<th>x</th>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>15213</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>7606</td>
<td>1D B6</td>
<td>00011101 10110110</td>
</tr>
<tr>
<td>x &gt;&gt; 4</td>
<td>950.8125</td>
<td>950</td>
<td>03 B6</td>
<td>00000001 10110110</td>
</tr>
<tr>
<td>x &gt;&gt; 8</td>
<td>59.4257813</td>
<td>59</td>
<td>00 3B</td>
<td>00000000 00111011</td>
</tr>
</tbody>
</table>
C Function

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

Compiled Arithmetic Operations

```
shrl $3, %eax
```

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 $u < 0$

![Division Diagram]

<table>
<thead>
<tr>
<th>Operands:</th>
<th>$x / 2^k$</th>
<th>Division:</th>
<th>$x / 2^k$</th>
<th>Result:</th>
<th>RoundDown($x / 2^k$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y$</td>
<td>$-15213$</td>
<td>$-15213$</td>
<td>C4 93</td>
<td>11000100 10010011</td>
<td></td>
</tr>
<tr>
<td>$y \gg 1$</td>
<td>$-7606.5$</td>
<td>$-7607$</td>
<td>E2 49</td>
<td>11100010 01001001</td>
<td></td>
</tr>
<tr>
<td>$y \gg 4$</td>
<td>$-950.8125$</td>
<td>$-951$</td>
<td>FC 49</td>
<td>11111100 01001001</td>
<td></td>
</tr>
<tr>
<td>$y \gg 8$</td>
<td>$-59.4257813$</td>
<td>$-60$</td>
<td>FF C4</td>
<td>11111111 11000100</td>
<td></td>
</tr>
</tbody>
</table>
**Correct Power-of-2 Divide**

- **Quotient of Negative Number by Power of 2**
  - Want \([x \div 2^k]\) (Round Toward 0)
  - \([x \div y] = \lceil (x + y - 1)/y \rceil\)
  - \([x \div 2^k] = \lceil (x + 2^k - 1)/2^k \rceil\)
  - Compute as \([ (x + 2^k - 1)/2^k \rceil\)
    - In C: \((x + (1<<k)-1) >> k\)
    - Biases dividend toward 0

- **Case 1: No rounding**

<table>
<thead>
<tr>
<th>Bias: (+2^k-1)</th>
<th>Dividend: (1\ldots;1\ldots;0)</th>
<th>Divisor: (2^k)</th>
<th>([u/2^k])</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0\ldots;0;1;1)</td>
<td>(1\ldots;1;1)</td>
<td>(0\ldots;0;1;0)</td>
<td>(1\ldots;1)</td>
</tr>
</tbody>
</table>

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

Case 2: Rounding

Biasing adds 1 to final result
C Function

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

Compiled Arithmetic Operations

```
testl %eax, %eax  
js   L4
L3:  
sarl $3, %eax
ret
L4:   
addl $7, %eax
jmp  L3
```

Explanation

- Uses arithmetic shift for `int`
- For Java Users
  - Arithmetic shift written as `>>`

```java
if x < 0
    x += 7;
# Arithmetic shift
return x >> 3;
```
**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)
**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
Today: Integers

- Representation: unsigned and signed
- Conversion, casting
- Expanding, truncating
- Addition, negation, multiplication, shifting
- Summary
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 \( \text{mod } 2^w \)

- Comparison to (mathematical) integer arithmetic
  - Both are rings
  - Integers obey ordering properties, e.g.,
    \[
    u > 0 \implies u + v > v \\
    u > 0, \ v > 0 \implies u \cdot v > 0
    \]
  - These properties are not obeyed by two’s comp. arithmetic
    \[
    T_{\text{Max}} + 1 = T_{\text{Min}} \\
    15213 \times 30426 = -10030 \quad (16\text{-bit words})
    \]
**Why Should I Use Unsigned?**

- **Practice Problem 2.23**
- **Don’t** use just because number nonnegative
  - 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)
    ```
- **Do** use when performing modular arithmetic
  - Multiprecision arithmetic
- **Do** use when using bits to represent sets
  - Logical right shift, no sign extension
### Integer C Puzzles

- \( x < 0 \implies ((x*2) < 0) \)
- \( ux >= 0 \)
- \( x & 7 == 7 \implies (x<<30) < 0 \)
- \( ux > -1 \)
- \( x > y \implies -x < -y \)
- \( x * x >= 0 \)
- \( x > 0 && y > 0 \implies x + y > 0 \)
- \( x >= 0 \implies -x <= 0 \)
- \( x <= 0 \implies -x >= 0 \)
- \( (x|\neg x)\gg31 == -1 \)
- \( ux \gg 3 == ux/8 \)
- \( x \gg 3 == x/8 \)
- \( x & (x-1) != 0 \)

---

**Initialization**

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