BITS, BYTES, AND INTEGERS

Spring, 2015
Euiseong Seo
(euiseong@skku.edu)
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 $FA1D37B_{16}$ in C as
      - 0xFA1D37B
      - 0xfa1d37b

<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 Words

- 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 \( \approx 1.8 \times 10^{19} \) 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
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)
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`

<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>
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>c3 ab 12 00 00</td>
<td>add $0x12ab,%ebx</td>
</tr>
<tr>
<td>804836c:</td>
<td>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
EXAMINING DATA REPRESENTATIONS

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

Result (Linux):

<table>
<thead>
<tr>
<th>int a = 15213;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x11fffffb8   0x6d</td>
</tr>
<tr>
<td>0x11fffffc9   0x3b</td>
</tr>
<tr>
<td>0x11fffffc0a  0x00</td>
</tr>
<tr>
<td>0x11fffffcbb  0x00</td>
</tr>
</tbody>
</table>
### Representing Integers

**int A = 15213;**

**Dec: 15213**

**Bin: 0011 1011 0110 1101**

**Hex: 3 B 6 D**

**int B = -15213;**

**Dec: 15213**

**Bin: 0011 1011 0110 1101**

**Hex: 3 B 6 D**

**long int C = 15213;**

**Dec: 15213**

**Bin: 0011 1011 0110 1101**

**Hex: 3 B 6 D**

**Two’s complement representation**
### Representing Pointers

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

<table>
<thead>
<tr>
<th>Sun</th>
<th>IA32</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>EF</td>
<td>D4</td>
<td>0C</td>
</tr>
<tr>
<td>FF</td>
<td>F8</td>
<td>89</td>
</tr>
<tr>
<td>FB</td>
<td>FF</td>
<td>EC</td>
</tr>
<tr>
<td>2C</td>
<td>BF</td>
<td>FF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7F</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00</td>
</tr>
</tbody>
</table>

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";
MACHINE-LEVEL CODE REPRESENTATION

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

```c
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

<table>
<thead>
<tr>
<th>and</th>
<th>or</th>
</tr>
</thead>
<tbody>
<tr>
<td>A&amp;B = 1 when both A=1 and B=1</td>
<td>A</td>
</tr>
<tr>
<td>&amp;</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</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></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></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

\[
\begin{array}{c}
01101001 \\
\& 01010101 \\
\hline
01000001
\end{array}
\quad
\begin{array}{c}
01101001 \\
| 01010101 \\
\hline
01111101
\end{array}
\quad
\begin{array}{c}
01101001 \\
^ 01010101 \\
\hline
00111100
\end{array}
\quad
\begin{array}{c}
\sim 01010101
\end{array}
\]

- 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`
    - `~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`
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

- 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>
Cool Stuff 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 */
}
```

<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
  o Representation: unsigned and signed
  o Conversion, casting
  o Expanding, truncating
  o Addition, negation, multiplication, shifting

► Summary
**Encoding Integers**

Unsigned

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

Two’s Complement

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

- **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></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>(y)</td>
<td>-15213</td>
<td>C4 93</td>
<td>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>1</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 \[ 15213 \] \[ -15213 \]
**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><strong>UMax</strong></td>
<td>65535</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td><strong>TMax</strong></td>
<td>32767</td>
<td>7F FF</td>
<td>01111111 11111111</td>
</tr>
<tr>
<td><strong>TMin</strong></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></th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
</tr>
<tr>
<td>UMax</td>
<td>255</td>
</tr>
<tr>
<td>Tmax</td>
<td>127</td>
</tr>
<tr>
<td>Tmin</td>
<td>-128</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
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>
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
Mappings between Signed & Unsigned

- Two’s Complement
  - x → T2B → B2U → ux
  - Maintain Same Bit Pattern

- Unsigned
  - ux → U2B → B2T → x
  - 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>

=  

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

Two’s Complement  \( \xrightarrow{T2U} \)  Unsigned

- **T2U**: Convert to unsigned
- **T2B**: Convert to two’s complement
- **B2U**: Convert back to unsigned

**Function**: \( ux = \begin{cases} 
  x & x \geq 0 \\
  x + 2^w & x < 0 
\end{cases} \)

- Large negative weight becomes large positive weight.

---

- **T2U**: Convert to unsigned
- **T2B**: Convert to two’s complement
- **B2U**: Convert back to unsigned

**Function**: \( ux = \begin{cases} 
  x & x \geq 0 \\
  x + 2^w & x < 0 
\end{cases} \)

- Large negative weight becomes large positive weight.
**CONVERSION VISUALIZED**

> 2’s Comp. → Unsigned

- Ordering Inversion
- Negative → Big Positive

---

2’s Complement Range

0

TMax

TMin

-2

-1

0

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
    • 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:
    - $X = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0$

![Diagram showing sign extension process](image)
**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>
**Justification for Sign Extension**

- 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
```

$$B_{2U_k}([x_k, x_{k-1}, \ldots, x_0]) = B_{2U_w}([x_w, x_{w-1}, \ldots, x_0]) \mod 2^k$$

$$B_{2T_k}([x_k, x_{k-1}, \ldots, x_0]) = U_{2T_k}(B_{2U_w}([x_w, x_{w-1}, \ldots, x_0]) \mod 2^k)$$
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 SIGNED AND UNSIGNED

- Implicit conversion of signed 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
Negation: Complement & Increment

- Claim: following holds for 2’s complement
  \[ \sim x + 1 = -x \]

- Complement
  - Observation: \[ \sim x + x = 1111\ldots111_2 = -1 \]

- 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</td>
<td>6D 00111011 01101101</td>
</tr>
<tr>
<td>$\neg x$</td>
<td>-15214</td>
<td>C4</td>
<td>92 11000100 10010010</td>
</tr>
<tr>
<td>$\neg x + 1$</td>
<td>-15213</td>
<td>C4</td>
<td>93 11000100 10010011</td>
</tr>
<tr>
<td>$-x$</td>
<td>-15213</td>
<td>C4</td>
<td>93 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</td>
<td>00 00000000 00000000</td>
</tr>
<tr>
<td>$\neg 0$</td>
<td>-1</td>
<td>FF</td>
<td>FF 11111111 11111111</td>
</tr>
<tr>
<td>$\neg 0 + 1$</td>
<td>0</td>
<td>00</td>
<td>00 00000000 00000000</td>
</tr>
</tbody>
</table>
## Unsigned Addition

<table>
<thead>
<tr>
<th><strong>Operands:</strong> w bits</th>
<th><strong>True Sum:</strong> w+1 bits</th>
<th><strong>Discard Carry:</strong> w bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>u</td>
<td>+ v</td>
<td>UAdd&lt;sub&gt;w&lt;/sub&gt;(u, v)</td>
</tr>
<tr>
<td>[ u + v ]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **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
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

$UAdd_4(u, v)$

Overflow
Mathematical Properties of $U\text{Add}$

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

<table>
<thead>
<tr>
<th>Operands: ( w ) bits</th>
<th>( u ) [\begin{array}{c} \ldots \end{array}] [\begin{array}{c} \ldots \end{array}] [\begin{array}{c} \ldots \end{array}] [\begin{array}{c} \ldots \end{array}]</th>
<th>+</th>
<th>( v ) [\begin{array}{c} \ldots \end{array}] [\begin{array}{c} \ldots \end{array}] [\begin{array}{c} \ldots \end{array}] [\begin{array}{c} \ldots \end{array}]</th>
<th>True Sum: ( w+1 ) bits</th>
<th>( u + v ) [\begin{array}{c} \ldots \end{array}] [\begin{array}{c} \ldots \end{array}] [\begin{array}{c} \ldots \end{array}] [\begin{array}{c} \ldots \end{array}]</th>
<th>Discard Carry: ( w ) bits</th>
<th>( \text{TAdd}_w(u, v) ) [\begin{array}{c} \ldots \end{array}] [\begin{array}{c} \ldots \end{array}] [\begin{array}{c} \ldots \end{array}] [\begin{array}{c} \ldots \end{array}]</th>
</tr>
</thead>
</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 \( s == t \)
**TAdd Overflow**

- True sum requires $w+1$ bits
- Drop off MSB
- Treat remaining bits as 2’s complement integer

![Diagram showing True Sum and TAdd Result](image)
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 \text{TAdd}

\textbf{Functionality}

- True sum requires $w+1$ bits
- Drop off MSB
- Treat remaining bits as 2’s complement integer

\begin{align*}
\text{TAdd}(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} \\
\text{NegOver} &\quad \text{PosOver}
\end{align*}
Mathematical Properties of TAdd

- Isomorphic group to unsigned with UAdd
  - $TAdd_w(u,v) = U2T(UAdd_w(T2U(u),T2U(v)))$
    - 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
  - Let
    - $TComp_w(u) = U2T(UComp_w(T2U(u)))$
    - $TAdd_w(u,TComp_w(u)) = 0$

$$TComp_w(u) = \begin{cases} -u & u \neq TMin_w \\ TMin_w & u = TMin_w \end{cases}$$
**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>( u \cdot v )</th>
<th>( \ast )</th>
<th>( v )</th>
</tr>
</thead>
<tbody>
<tr>
<td>True Product: ( 2\ast w ) bits</td>
<td>( \cdots )</td>
<td>( \cdots )</td>
<td>( \cdots )</td>
<td>( \cdots )</td>
</tr>
<tr>
<td>Discard: ( w ) bits</td>
<td>( \cdots )</td>
<td>( \cdots )</td>
<td>( \cdots )</td>
<td>( \cdots )</td>
</tr>
</tbody>
</table>

- **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:
- `ele_src`
- `malloc(ele_cnt*ele_size)`
- Allocation and copying of elements
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

What if:

- ele_cnt = $2^{20} + 1$
- ele_size = 4096
- Allocation = ??

How can I make this function secure?
### Signed Multiplication in C

<table>
<thead>
<tr>
<th>Operands: $w$ bits</th>
<th>$u \cdot v$</th>
<th>$u$</th>
</tr>
</thead>
<tbody>
<tr>
<td>True Product: $2w$ bits</td>
<td>$\cdots$</td>
<td>$\cdots$</td>
</tr>
<tr>
<td>Discard: $w$ bits</td>
<td>$\cdots$</td>
<td>$\cdots$</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>
</tr>
</thead>
<tbody>
<tr>
<td>( u \times 2^k )</td>
<td>Discard ( k ) bits: ( w ) bits</td>
</tr>
<tr>
<td>( u \ll k )</td>
<td>( \text{UMult}_w(u, 2^k) )</td>
</tr>
<tr>
<td>( u \ll 3 )</td>
<td>( \text{TMult}_w(u, 2^k) )</td>
</tr>
</tbody>
</table>

- **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
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:**

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

**Division:**

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

**Result:**

\[
\begin{array}{c}
\lfloor u / 2^k \rfloor \\
\hline
\text{Binary Point} \\
\end{array}
\]

<table>
<thead>
<tr>
<th></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 \gg 1 )</td>
<td>7606.5</td>
<td>7606</td>
<td>1D B6</td>
<td>00011101 10110110</td>
</tr>
<tr>
<td>( x \gg 4 )</td>
<td>950.8125</td>
<td>950</td>
<td>03 B6</td>
<td>00000011 10110110</td>
</tr>
<tr>
<td>( x \gg 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 $u < 0$

<table>
<thead>
<tr>
<th>Operands: $x / 2^k$</th>
<th>Result: $\text{RoundDown}(x / 2^k)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>$\text{RoundDown}(x / 2^k)$</td>
</tr>
<tr>
<td>Binary Point</td>
<td></td>
</tr>
</tbody>
</table>

<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>-15213 C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>$y \gg 1$</td>
<td>-7606.5</td>
<td>-7607 E2 49</td>
<td>11100010 01001001</td>
</tr>
<tr>
<td>$y \gg 4$</td>
<td>-950.8125</td>
<td>-951 FC 49</td>
<td>11111100 01001001</td>
</tr>
<tr>
<td>$y \gg 8$</td>
<td>-59.4257813</td>
<td>-60 FF C4</td>
<td>11111111 11000100</td>
</tr>
</tbody>
</table>
Correct Power-of-2 Divide

Quotient of Negative Number by Power of 2

- Want \( \left\lceil \frac{x}{2^k} \right\rceil \) (Round Toward 0)
- \( \left\lceil \frac{x}{y} \right\rceil = \left\lfloor \frac{x + y - 1}{y} \right\rfloor \)
- \( \left\lceil \frac{x}{2^k} \right\rceil = \left\lfloor \frac{x + 2^k - 1}{2^k} \right\rfloor \)
- Compute as \( \left\lfloor \frac{x + 2^k - 1}{2^k} \right\rfloor \)
  - 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: ( \frac{u}{2^k} )</th>
<th>Divisor: ( 2^k )</th>
<th>( \left\lceil \frac{u}{2^k} \right\rceil )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 [ \cdots ] 0</td>
<td>1 [ \cdots ] 1</td>
<td>0 [ \cdots ] 0</td>
<td>1 [ \cdots ] 1</td>
</tr>
<tr>
<td>0 [ \cdots ] 1</td>
<td>0 [ \cdots ] 1</td>
<td>0 [ \cdots ] 0</td>
<td>1 [ \cdots ] 1</td>
</tr>
<tr>
<td>0 [ \cdots ] 0</td>
<td>1 [ \cdots ] 0</td>
<td>0 [ \cdots ] 0</td>
<td>1 [ \cdots ] 1</td>
</tr>
</tbody>
</table>

Binary Point

Biasing has no effect
CORRECT POWER-OF-2 DIVIDE (CONT.)

Case 2: Rounding

Biasing adds 1 to final result
**Compiled Signed Division Code**

**C Function**

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

**Compiled Arithmetic Operations**

```assembly
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 $\text{mod } 2^w$
    - Mathematical addition + possible subtraction of $2^w$
  - Signed: modified addition $\text{mod } 2^w$ (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 $\text{mod } 2^w$
  - Signed: modified multiplication $\text{mod } 2^w$ (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
    \[ 0 \leq U\text{Mult}_w(u,v) \leq 2^w - 1 \]
  - Multiplication Commutative
    \[ U\text{Mult}_w(u,v) = U\text{Mult}_w(v,u) \]
  - Multiplication is Associative
    \[ U\text{Mult}_w(t,U\text{Mult}_w(u,v)) = U\text{Mult}_w(U\text{Mult}_w(t,u),v) \]
  - 1 is multiplicative identity
    \[ U\text{Mult}_w(u,1) = u \]
  - Multiplication distributes over addition
    \[ U\text{Mult}_w(t,U\text{Add}_w(u,v)) = U\text{Add}_w(U\text{Mult}_w(t,u),U\text{Mult}_w(t,v)) \]
PROPERTIES OF TWO’S COMP. ARITHMETIC

► Isomorphic algebras
  o Unsigned multiplication and addition
    • Truncating to \( w \) bits
  o Two’s complement multiplication and addition
    • Truncating to \( w \) bits

► Both form rings
  o Isomorphic to ring of integers \( \text{mod } 2^w \)

► Comparison to (mathematical) integer arithmetic
  o Both are rings
  o Integers obey ordering properties, e.g.,
    \[
    \begin{align*}
    u > 0 & \implies u + v > v \\
    u > 0, \ v > 0 & \implies u \cdot v > 0
    \end{align*}
    \]
  o These properties are not obeyed by two’s comp. arithmetic
    \[
    \begin{align*}
    \text{TMax} + 1 & = \text{TMin} \\
    15213 \times 30426 & = -10030 \quad (16\text{-bit words})
    \end{align*}
    \]
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**

- \( x < 0 \) \( \Rightarrow \) \( ((x*2) < 0) \)
- \( ux >= 0 \)
- \( x & 7 == 7 \) \( \Rightarrow \) \( (x<<30) < 0 \)
- \( ux > -1 \)
- \( x > y \) \( \Rightarrow \) \( -x < -y \)
- \( x * x >= 0 \)
- \( x > 0 && y > 0 \) \( \Rightarrow \) \( x + y > 0 \)
- \( x >= 0 \) \( \Rightarrow \) \( -x <= 0 \)
- \( x <= 0 \) \( \Rightarrow \) \( -x >= 0 \)
- \( (x|-x)>>31 == -1 \)
- \( ux >> 3 == ux/8 \)
- \( x >> 3 == x/8 \)
- \( x & (x-1) != 0 \)

Initialization

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