Bits, Bytes, and Integers

CSci 2021: Machine Architecture and Organization
Lectures #2-4, January 23rd-28th, 2015

Your Instructor: Stephen McCamant

Based on slides originally by:
Randy Bryant, Dave O’Hallaron, Antonia Zhai

Encoding Byte Values

- **Byte = 8 bits**
  - Binary 00000000 to 11111111
  - Decimal: 0 to 255
  - Hexadecimal 00 to FF

- **Base 16 number representation**
- Use characters '0' to '9' and 'A' to 'F'
- Write 0xA1D37B in C as 0xFA1D37B

Byte-Oriented Memory Organization

- **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 = 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

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)

Binary Representations
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

- How should bytes within a multi-byte word be ordered in memory?
- Conventions
  - Big Endian: Sun, PPC Mac, Internet convention
    - Least significant byte has highest address
  - Little Endian: x86, VAX
    - Least significant byte has lowest address

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 00</td>
<td>cmpl $0x0,0x28(%ebx)</td>
</tr>
</tbody>
</table>

- Deciphering Numbers
  - Value: 0x01234567
  - Pad to 32 bits: 0x000001234567
  - Split into bytes: 01 23 45 67
  - Reverse: 67 45 23 01

Examining Data Representations

- Code to Print Byte Representation of Data
  - Casting pointer to unsigned char * creates byte array

```c
void show_bytes(unsigned char *start, int len){
    int i;
    for (i = 0; i < len; i++)
        printf("%p\t%02x\n", start+i, start[i]);
}
```

- Print directives:
  - %p: Print pointer
  - %x: Print Hexadecimal

show_bytes Execution Example

```c
int a = 15213;
printf("int a = %d;\n", a);
show_bytes((unsigned char *)&a, sizeof(int));
```

Result (Linux):

```c
int a = 15213;
0xbffffcb8 0x6d
0xbffffcb9 0x3b
0xbffffcbb 0x00
```
**Representing Integers**

int A = 15213;

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

int B = -15213;

Two’s complement representation (Covered later)

int C = 15213;

**Representing Pointers**

int B = -15213;

long int C = 15213;

**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

- **Aside: ASCII table**

**Today: Bits, Bytes, and Integers**

- Representing information as bits
- (Logistics interlude)
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
- Summary

**Homework turn-in process**

- For full credit: turn in at the beginning of class on the due date
  - On-time = 3:35pm, or when I start lecturing, whichever is later
  - Yes, this means you have to come to class on time (that day)
- We strongly recommend typing your assignments on a computer, not hand-writing
- Late submissions only will be online using the Moodle
- Do not turn in paper assignments at other times
  - This helps us stay organized
2021-dedicated VMs now available

- SSH into: x-A-B.cselabs.umn.edu
  - Where A is 21, 22, or 23
  - And B is 01, 02, 03, 04, or 05
  - E.g., x22-02.cselabs.umn.edu
- 32-bit version of Ubuntu Linux version 14.04
- Do not run graphical programs (Firefox, etc.) on these machines (it would be slow anyway)
- If you prefer to use other CSE Labs Linux machines, give the \(-m32\) option to GCC to get 32-bit binaries

**Boolean Algebra**

- Developed by George Boole in 19th Century
  - Algebraic representation of logic
  - Encode "True" as 1 and "False" as 0
- And (math: \( \land \))
  \[-A \land -B = 1 \text{ when both } A=1 \text{ and } B=1\]
- Or (math: \( \lor \))
  \[-A \lor -B = 1 \text{ when either } A=1 \text{ or } B=1\]
- Not (math: \( \neg \))
  \[-A = 1 \text{ when } A=0\]
- Exclusive-Or "xor" (math: \( \oplus \))
  \[-A \oplus -B = 1 \text{ when either } A=1 \text{ or } B=1, \text{ but not both}\]

**Application of Boolean Algebra**

- Applied to Digital Systems by Claude Shannon
  - 1937 MIT Master’s Thesis
  - Reason about networks of relay switches
    - Encode closed switch as 1, open switch as 0
  - \(A \land -B \lor -A \land B\)
  - \(A \land B\)

**Representing & Manipulating Sets**

- **Representation**
  - Width \( w \) bit vector represents subsets of \( \{0, \ldots, w-1\} \)
  - \(A_i = 1\) if \(i \in A\)
  - \(\{0, 3, 5, 6\}\)
  - \(\{0, 2, 4, 6\}\)
  - \(\{0, 6\}\)
- **Operations**
  - \& Intersection
  - \| Union
  - ^ Symmetric difference
  - ~ Complement

**General Boolean Algebras**

- Operate on Bit Vectors
  - Operations applied bitwise
  - \(01101001 \land 01010101 \rightarrow 01000001\)
  - \(01101001 \lor 01010101 \rightarrow 01111101\)
  - \(\sim 0x00 \rightarrow 0xFF\)
  - \(\sim 0x69 \land 0x55 \rightarrow 0x41\)

**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
  - \(~0x41 \rightarrow 0xBE\)
  - \(~01000001 \rightarrow 11111110\)
  - \(\sim 0x00 \rightarrow 0xFF\)
  - \(~00000000 \rightarrow 11111111\)
  - \(0x69 \land 0x55 \rightarrow 0x41\)
  - \(0x11010101 \land 0x10101010 \rightarrow 0x1000001\)
  - \(0x69 \land 0x55 \rightarrow 0x1D\)
  - \(0x11010101 \land 0x10101010 \rightarrow 0x1111101\)
Contrast: Logic Operations in C

- **Contrast to Logical Operators**
  - &&, ||, !
  - View 0 as “False”
  - Anything nonzero as “True”
  - Always return 0 or 1
  - Early termination (aka “short-circuit evaluation”)

- **Examples (char data type)**
  - 10x41 → 0x00
  - 10x90 → 0x01
  - 10x41 → 0x01
  - 0x69 & 0x55 → 0x01
  - 0x69 || 0x5d → 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 ≥ word size

---

Exercise break: flip case

```c
/* Convert lowercase to uppercase and vice-versa, return any other characters unchanged */
char flip_case(char c) {
    if (c >= 'A' && c <= 'Z') {
        return c | 0x20;
    } else if (c >= 'a' && c <= 'z') {
        return c & ~0x20;
    } else {
        return c;
    }
}
```

- Fill in the blanks, using bitwise operators

---

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

---

Encoding Example (Cont.)

<table>
<thead>
<tr>
<th>x</th>
<th>00110010</th>
<th>01101101</th>
<th>01101101</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>-15213</td>
<td>110010100</td>
<td>10101011</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Weight</th>
<th>15213</th>
<th>-15213</th>
</tr>
</thead>
<tbody>
<tr>
<td>4096</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2048</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1024</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>512</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>256</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>128</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>64</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Sign Bit

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

---

Encoding Integers

<table>
<thead>
<tr>
<th>Unsigned</th>
<th>Two's Complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>(R_{2U}(x) = \sum_{i=0}^{w} 2^i)</td>
<td>(R_{2T}(x) = -x_{w-1} 2^{w-1} + \sum_{i=0}^{w-2} 2^i)</td>
</tr>
</tbody>
</table>

- 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>00110010</td>
</tr>
<tr>
<td>y</td>
<td>-15213</td>
<td>110010100</td>
</tr>
</tbody>
</table>

- Sign Bit

- C short 2 bytes long
### Numeric Ranges

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

- **Two's Complement Values**
  - Tmin = $-2^{w-1}$
  - 101...1
  - Tmax = $2^{w-1} - 1$
  - 011...1

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

<table>
<thead>
<tr>
<th>Values for W = 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal</td>
</tr>
<tr>
<td>UMax</td>
</tr>
<tr>
<td>UMin</td>
</tr>
<tr>
<td>TMax</td>
</tr>
<tr>
<td>TMin</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

- $|TMin| = TMax + 1$
- Asymmetric range
- $UMax = 2 * TMax + 1$

### C Programming

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

### 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

### Announcement interlude: Lab 1 out

- Lab 0 (Hello, world) is due tonight
- Lab 1 on data representation is out
- Basic idea: puzzles implementing operations with other operations
  - E.g., implement logical right shift using only arithmetic right shift
- Most problems relate to bitwise operations and two's complement rules
  - I.e., you can start working on them now
- Increasing difficulty, try the easier ones first
- Two questions relating to floating point

### Mapping Between Signed & Unsigned

- 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>

Relation between Signed & Unsigned

Two’s Complement

<table>
<thead>
<tr>
<th>$x$</th>
<th>$T_{2U}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$w-1$</td>
<td>$w-1$</td>
</tr>
</tbody>
</table>

$t_{2U} = \begin{cases} x & x \geq 0 \\ x + 2^w & x < 0 \end{cases}$

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;

Conversion Visualized

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

<table>
<thead>
<tr>
<th>$T_{\text{Max}}$</th>
<th>$T_{\text{Min}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$U_{\text{Max}}$</td>
<td>$U_{\text{Min}}$</td>
</tr>
<tr>
<td>$T_{\text{Max}} + 1$</td>
<td>$T_{\text{Min}}$</td>
</tr>
</tbody>
</table>

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 $W = 32$: $T_{\text{Min}} = -2,147,483,648$, $T_{\text{Max}} = 2,147,483,647$

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 $W = 32$: $T_{\text{Min}} = -2,147,483,648$, $T_{\text{Max}} = 2,147,483,647$

<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>2147483647</td>
<td>-2147483647-1</td>
<td>$&gt;$</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647U</td>
<td>-2147483647-1</td>
<td>$&gt;$</td>
<td>signed</td>
</tr>
<tr>
<td>-1</td>
<td>-2</td>
<td>$&gt;$</td>
<td>unsigned</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>
Code Security Example

/* Kernel memory region holding user-accessible data */
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 */
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);
    ...
}

Malicious Usage

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

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

/* Copy at most maxlen bytes from kernel region to user buffer */
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);
}

Summary

Casting Signed ↔ Unsigned: Basic Rules

- Bit pattern is maintained
- But reinterpretated
- Can have unexpected effects: adding or subtracting $2^w$

Expression containing signed and unsigned int

- int is cast to unsigned!!

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:
  - $X' = x_{w-1}x_{w-2}...x_0$,
  - $X'_{w-k}...x_{w-2}x_0$
- $k$ copies of MSB

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 Example

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

<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>11000100 10010011</td>
</tr>
<tr>
<td>y</td>
<td>-15213</td>
<td>11111111 11111111 11000100 10010011</td>
</tr>
</tbody>
</table>

Decimal | Hex | Binary
--- | --- | ---
x | 15213 | 00111011 01101101 |
x | 3B 6D | 00111011 01101101 |
y | -15213 | 11000100 10010011 |
y | C4 93 | 11000100 10010011 |
y | -15213 | 11111111 11111111 11000100 10010011 |

Summary:
- Expanding (e.g., short int to int)
  - Unsigned: zeros added (“zero extension”)
  - 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 behaviour

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
  \[-x + 1 == -x\]
- Complement
  - Observation:
    \[\neg x + x == 1111\ldots111 == -1\]
    \[\neg 0 + 0 == \text{FF FF} == -1\]
  - Where would we fill in gaps for a more complete proof?
  - Note: operation can apply to unsigned as well
  - Two values for which \(x\) and \(-x\) have the same sign

Complement & Increment Examples

\[u = 15213\]
<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>x</td>
<td>3B 6D</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>x</td>
<td>-15213</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>y</td>
<td>-15213</td>
<td>11111111 11111111 11000100 10010011</td>
</tr>
</tbody>
</table>

\[x = 0\]
<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00 00</td>
<td>00000000 00000000</td>
</tr>
<tr>
<td>-01</td>
<td>00 00</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>

Unsigned Addition
- Operands: \(w\) bits
- True Sum: \(w+1\) bits
- Discard Carry: \(w\) bits

\[\text{UAdd}(u, v) = u + v \text{ mod } 2^w\]
- Standard Addition Function
  - Ignores carry output
- Implements Modular Arithmetic
  \[s = \text{UAdd}(u, v) = u + v \text{ mod } 2^w\]
Visualizing (Mathematical) Integer Addition

- Integer Addition
  - 4-bit integers \( u, v \)
  - Compute true sum \( \text{Add}_4(u, v) \)
  - Values increase linearly with \( u \) and \( v \)
  - Forms planar surface

Visualizing Unsigned Addition

- Wraps Around
  - If true sum \( \geq 2^w \)
  - At most once

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
    \[ \text{UAdd}_w(\text{UAdd}_w(u, v), w) = \text{UAdd}_w(u, \text{UAdd}_w(v, w)) \]
  - 0 is additive identity
    \[ \text{UAdd}_w(u, 0) = u \]
  - Every element has additive inverse
    - Let \( \text{UComp}_w(u) = 2^w - u \)
    - \( \text{UAdd}_w(u, \text{UComp}_w(u)) = 0 \)

Two’s Complement Addition

- TAdd and UAdd have Identical Bit-Level Behavior
  - Signed vs. unsigned addition in C:
    \[ \text{int } s, t, u, v; \]
    \[ s = (\text{int}) \{(\text{unsigned}) u + (\text{unsigned}) v\}; \]
    \[ t = u + v \]
  - Will give \( s == t \)

TAdd Overflow

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

Visualizing 2’s Complement Addition

- Values
  - 4-bit two’s comp.
  - Range from -8 to +7
- Wraps Around
  - If sum \( \geq 2^w \)
    - Becomes negative
    - At most once
  - If sum \( < -2^w \)
    - 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

\[
\text{TAdd}(u, v) = u + v + 2^{w-1} \text{ if } u + v < T_{\text{Min}}
\]
\[
\text{TAdd}(u, v) = u + v \text{ if } T_{\text{Min}} \leq u + v \leq T_{\text{Max}}
\]
\[
\text{TAdd}(u, v) = u + v - 2^{w-1} \text{ if } u + v > T_{\text{Max}}
\]

Mathematical Properties of TAdd

- **Isomorphic Group to unsigned with UAdd**
  - \( \text{TAdd}(u, v) = U2T(U\text{Add}(T2U(u), T2U(v))) \)
  - 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

\[
\text{TComp}(u) = \begin{cases} -u & u \neq T_{\text{Min}} \\ T_{\text{Min}} & u = T_{\text{Min}} \end{cases}
\]

Exercise break: ten’s complement

- Before digital computers, there were mechanical computers that used base 10
- There’s an analog of two’s complement called ten’s complement that works in decimal
- Suppose we have an adding machine with 10 decimal digits, 10^4 instead of 2^32.
- What should be the ten’s complement representation of -21?
- i.e., we want a number \( x \) so that adding \( x \) is the same as subtracting 21, when you only have 4 digits

\[
x = 10000 - 21 = 9979
\]

Ten’s complement answer

Signed/Unsigned Overflow Differences

- **Unsigned:**
  - Overflow if carry out of last position
  - Also just called “carry” (C)
- **Signed:**
  - Result wrong if input signs are the same but output sign is different
  - In CPUs, unqualified “overflow” usually means signed (O or V)

Multiplication

- **Computing Exact Product of \( w \)-bit numbers \( x, y \)**
  - Either signed or unsigned
- **Ranges**
  - **Unsigned:** \( 0 \leq x \cdot y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1 \)
    - Up to 2w bits
  - **Two’s complement min:** \( x \cdot y \leq -(2^{w-1})(2^{w-1} - 1) = -2^{2w-2} + 2^{w-1} \)
    - Up to 2w-1 bits
  - **Two’s complement max:** \( x \cdot y \leq (2^{w-1})^2 = 2^{2w-2} \)
    - Up to 2w bits, but only for \( T_{\text{Min}} \) range
- **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

Operands: $w$ bits
$u \cdot v$ bits
True Product: $2^w$ bits
$u \cdot v \bmod 2^w$
Discard $w$ bits: $w$ bits

- Standard Multiplication Function
  - Ignores high order $w$ bits
- Implements Modular Arithmetic
  $\text{UMult}_w(u,v) = u \cdot v \mod 2^w$

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) {
    /* Allocate buffer for ele_cnt objects, each of ele_size bytes */
    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 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 */
    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:
  - $\text{ele_cnt} = 2^{10} + 1$
  - $\text{ele_size} = 4096 = 2^{12}$
  - Allocation = ??

- How can I make this function secure?

Signed Multiplication in C

Operands: $w$ bits
$u \cdot v$ bits
True Product: $2^w$ bits
$u \cdot v \bmod 2^w$
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 $u \cdot 2^k$
  - Both signed and unsigned

```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 */
    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;
}
```

- Examples
  - $u \ll 3 = u \cdot 8$
  - $u \ll 5 - u \ll 3 = u \cdot 24$
  - Most machines shift and add faster than multiply
    - Compiler generates this code automatically
Compiled Multiplication Code

C Function

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

Compiled Arithmetic Operations

<table>
<thead>
<tr>
<th>leal (t, eax, 0x2), t</th>
<th>t &lt;= x * x * 2</th>
<th>Compiled Multiplication Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>sall $2, t</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Background: Rounding in Math

- How to round to the nearest integer?
- Cannot have both:
  - \( \text{round}(x + k) = \text{round}(x) + k \) (\( k \) integer), "translation invariance"
  - \( \text{round}(-x) = \text{round}(x) \) "negation invariance"

- [\( \lfloor x \rfloor \), read "floor": always round down (to \(-\infty\)):]
  - \( \lfloor 2.0 \rfloor = 2, \lfloor 1.7 \rfloor = 1, \lfloor -2.2 \rfloor = -3 \)

- [\( \lceil x \rceil \), read "ceiling": always round up (to \(+\infty\)):]
  - \( \lceil 2.0 \rceil = 2, \lceil 1.7 \rceil = 2, \lceil -2.2 \rceil = -2 \)

- C integer operators mostly use round to zero, which is like floor for positive and ceiling for negative

Division in C

- Integer division \( \div \): rounds towards 0
  - Choice (settled in C99) is historical, via FORTRAN and most CPUs
- Division by zero: undefined, usually fatal
- Unsigned division: no overflow possible
- Signed division: overflow almost impossible
  - Exception: TMin/-1 is un-representable, and so undefined
  - On x86 this too is a default-fatal exception

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

<table>
<thead>
<tr>
<th>Operands: ( u \gg k )</th>
<th>Binary Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>( u \gg k )</td>
<td></td>
</tr>
<tr>
<td>( u / 2^k )</td>
<td></td>
</tr>
<tr>
<td>Result: ( \lfloor u / 2^k \rfloor )</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>x</td>
<td>15213</td>
<td>CB 6D</td>
<td>0011101101100111</td>
</tr>
<tr>
<td>x &gt;&gt; 1</td>
<td>7606.5</td>
<td>E2 49</td>
<td>1100010010010011</td>
</tr>
<tr>
<td>x &gt;&gt; 4</td>
<td>950.8125</td>
<td>FC 49</td>
<td>1111101001001001</td>
</tr>
<tr>
<td>x &gt;&gt; 8</td>
<td>59.4257813</td>
<td>FF C4</td>
<td>111111111000000</td>
</tr>
</tbody>
</table>

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 \gg k )</th>
<th>Binary Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x \gg k )</td>
<td></td>
</tr>
<tr>
<td>( x / 2^k )</td>
<td></td>
</tr>
<tr>
<td>Result: ( \lfloor x / 2^k \rfloor )</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>-CB 6D</td>
<td>101010101101100</td>
</tr>
<tr>
<td>y &gt;&gt; 1</td>
<td>-7606.5</td>
<td>-E2 49</td>
<td>1110001010010011</td>
</tr>
<tr>
<td>y &gt;&gt; 4</td>
<td>-950.8125</td>
<td>-FC 49</td>
<td>1011110101001001</td>
</tr>
<tr>
<td>y &gt;&gt; 8</td>
<td>-59.4257813</td>
<td>-FF C4</td>
<td>1111111111000000</td>
</tr>
</tbody>
</table>

Compiled Unsigned Division Code

C Function

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

Compiled Arithmetic Operations

<table>
<thead>
<tr>
<th>shr1 $3, t</th>
<th>return $x &gt;&gt; 3</th>
<th>Compiled Unsigned Division Code</th>
</tr>
</thead>
</table>

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

Unsigned 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 \gg k )</th>
<th>Binary Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x \gg k )</td>
<td></td>
</tr>
<tr>
<td>( x / 2^k )</td>
<td></td>
</tr>
<tr>
<td>Result: ( \lfloor x / 2^k \rfloor )</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>-CB 6D</td>
<td>101010101101100</td>
</tr>
<tr>
<td>y &gt;&gt; 1</td>
<td>-7606.5</td>
<td>-E2 49</td>
<td>1110001010010011</td>
</tr>
<tr>
<td>y &gt;&gt; 4</td>
<td>-950.8125</td>
<td>-FC 49</td>
<td>1011110101001001</td>
</tr>
<tr>
<td>y &gt;&gt; 8</td>
<td>-59.4257813</td>
<td>-FF C4</td>
<td>1111111111000000</td>
</tr>
</tbody>
</table>
Correct Power-of-2 Divide

**Quotient of Negative Number by Power of 2**
- Want \( \lceil x / 2^k \rceil \) \text{ (Round Toward 0)}
- Compute as \( \lceil (x+2^{k-1}) / 2^k \rceil \)
  - In C: \( x + (1<<k) - 1 \gg k \)
  - Biases dividend toward 0

**Case 1: No rounding**

<table>
<thead>
<tr>
<th>Dividend: ( x )</th>
<th>( x + (1&lt;&lt;k) - 1 \gg k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 0 )</td>
<td>( 0 )</td>
</tr>
<tr>
<td>( 1 )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>( 10 )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>( 11 )</td>
<td>( 1 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Divisor: 1 ( \gg k )</th>
<th>( x / 2^k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 0 )</td>
<td>( 0 )</td>
</tr>
<tr>
<td>( 1 )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>( 10 )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>( 11 )</td>
<td>( 1 )</td>
</tr>
</tbody>
</table>

**Binary Point**
- Incremented by 1

*Biasing has no effect*

Correct Power-of-2 Divide (Cont.)

**Case 2: Rounding**

<table>
<thead>
<tr>
<th>Dividend: ( x )</th>
<th>( x + (1&lt;&lt;k) - 1 \gg k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 0 )</td>
<td>( 0 )</td>
</tr>
<tr>
<td>( 1 )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>( 10 )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>( 11 )</td>
<td>( 1 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Divisor: 1 ( \gg k )</th>
<th>( x / 2^k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 0 )</td>
<td>( 0 )</td>
</tr>
<tr>
<td>( 1 )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>( 10 )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>( 11 )</td>
<td>( 1 )</td>
</tr>
</tbody>
</table>

*Incremented by 1*

*Biases adds 1 to final result*

Compiled Signed Division Code

**C Function**

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

**Explanation**
- Uses arithmetic shift for int
- For Java Users
  - Arith. shift written as >>

Remainder operator

- Written as \( \% \) in C
- \( x \ % \ y \) is the remainder after division \( x / y \)
- E.g., \( x \ % \ 10 \) is the lowest digit of non-negative \( x \)
- Behavior for negative values matches \( / \)'s rounding toward zero
  - \( b \times (a / b) + (a \ % \ b) = a \)
  - I.e. sign of remainder matches sign of dividend
- (Some other languages have other conventions: sign of result equals sign of divisor, sometimes distinguished as “modulo”, or always positive)

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)

- **Left shift**
  - Unsigned/signed: multiplication by 2^w
  - Always logical shift

- **Right shift**
  - Unsigned: logical shift, div (division + round to zero) by 2^w
  - Signed: arithmetic shift
    - Positive numbers: div (division + round to zero) by 2^w
    - Negative numbers: div (division + round away from zero) by 2^w
    - 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 mod \( 2^w \)
- Comparison to (Mathematical) Integer Arithmetic
  - Both are rings
  - Integers obey ordering properties, e.g.,
    \[ u > 0 \Rightarrow u + v > v \]
    \[ u > 0, v > 0 \Rightarrow u \cdot v > 0 \]
  - These properties are not obeyed by two’s comp. arithmetic
    \( \text{TMax} + 1 = \text{TMin} \)
    \[ 15213 \cdot 30426 = -10030 \] (16-bit words)

Why Should I Use Unsigned?

- Don’t Use Just Because Number Nonnegative
  - Easy to make mistakes
    \[ \text{unsigned } i; \]
    \[ \text{for } (i = \text{cnt} - 2; i >= 0; i--) \]
    \[ a[i] += a[i+1]; \]
  - Can be very subtle
    \#define DELTA sizeof(int)
    \[ \text{int } i; \]
    \[ \text{for } (i = \text{CNT}; i - \text{DELTA} >= 0; i += \text{DELTA}) \]
- Do Use When Performing Modular Arithmetic
  - E.g., used in multiprecision arithmetic
- Do Use When Using Bits to Represent Sets
  - Logical right shift, no sign extension

Integer C Puzzles

- \( x < 0 \Rightarrow (x \cdot 2) < 0 \)
- \( uz >> 0 \Rightarrow (x<30) < 0 \)
- \( uz >> -1 \Rightarrow x < y \)
- \( x \cdot x >> 0 \Rightarrow x < y \)
- \( x > 0 \& \& y > 0 \Rightarrow x + y > 0 \)
- \( x >> 0 \Rightarrow x < 0 \)
- \( x <= 0 \Rightarrow x > 0 \)
- \( x >> 3 == x/8 \)
- \( x \& (x-1) \neq 0 \)

Initialization

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