Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary
- Representations in memory, pointers, strings

Everything is bits

- Each bit is 0 or 1
- By encoding/interpreting sets of bits in various ways
  - Computers determine what to do (instructions)
  - ... and represent and manipulate numbers, sets, strings, etc...
- Why bits? Electronic Implementation
  - Easy to store with bistable elements
  - Reliably transmitted on noisy and inaccurate wires

For example, can count in binary

- Base 2 Number Representation
  - Represent 15213₁₀ as 1110110110110₁₂
  - Represent 1.2₀₁₀ as 1.0011001100110011[0011]…₂
  - Represent 1.5₂₁₀ × 10⁻₄ as 1.11011011011₀₂ × 2⁻¹³

Encoding Byte Values

- Byte = 8 bits
  - Binary 00000000 to 11111111
  - Decimal: 0 to 255
  - Hexadecimal: 0x0 to FF
    - Base 16 number representation
    - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
    - Write FA1D37Bu in C as
      - 0xFA1D37B
      - 0xfa1d37b

Aside: ASCII table

<table>
<thead>
<tr>
<th>DEC</th>
<th>OCT</th>
<th>HEX</th>
<th>BINARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0</td>
<td>0x0</td>
<td>00000000</td>
</tr>
<tr>
<td>1</td>
<td>0 1</td>
<td>0x1</td>
<td>00000001</td>
</tr>
<tr>
<td>2</td>
<td>0 2</td>
<td>0x2</td>
<td>00000010</td>
</tr>
<tr>
<td>11</td>
<td>0 13</td>
<td>0x13</td>
<td>00001101</td>
</tr>
<tr>
<td>33</td>
<td>0 41</td>
<td>0x2b</td>
<td>00100011</td>
</tr>
<tr>
<td>63</td>
<td>0 7 f</td>
<td>0x3f</td>
<td>00111111</td>
</tr>
<tr>
<td>10</td>
<td>0 e</td>
<td>0x1e</td>
<td>0011110</td>
</tr>
</tbody>
</table>
Example Data Representations

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Typical 32-bit</th>
<th>Typical 64-bit</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>4</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>–</td>
<td>–</td>
<td>10/16</td>
</tr>
<tr>
<td>pointer</td>
<td>4</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary
- Representations in memory, pointers, strings

Boolean Algebra

- Developed by George Boole in 19th Century
  - Algebraic representation of logic
    - Encode "True" as 1 and "False" as 0

  And (math: $\land$)
  
  Or (math: $\lor$)

  Not (math: $\neg$)


General Boolean Algebras

- Operate on bit vectors
  - Operations applied bitwise


Example: Representing & Manipulating Sets

- Representation
  - Width $w$ bit vector represents subsets of $\{0, \ldots, w-1\}$
  - $a_j=1 \iff A \cap B$


Bit-Level Operations in C

- Operations $\&$, $\mid$, $\neg$, $\lor$ 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)
  


Example: Bit-Level Operations in C

- $\&$ & $\mid$, $\neg$, $\lor$ 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)
Contrast: Logic Operations in C

- **Contrast to Logical Operators**
  - `&&`, `||`, `!`
  - View 0 as “False”, 1 as “True”
  - Arith. & Logic
  - Early termination (AKA “short circuit evaluation”)

- **Examples**
  - `0x0970bd90`\( \land 0x2248730c \) → \( 0x01 \)
  - `0x69 || 0x55` → undefined behavior
  - `0xe \& \& 0x55` → 0x01

- **Shift Operations**
  - **Left Shift:** \( x \ll y \)
    - Shift bit-vector \( x \) left \( y \) positions
      - Throw away extra bits on left
      - Fill with 0’s on right
  - **Right Shift:** \( x \gg y \)
    - Shift bit-vector \( x \) right \( y \) positions
      - Throw away extra bits on right
      - Logical shift: fill with 0’s on left
      - Arithmetic shift: replicate most significant bit on left
  - **Undefined Behavior**
    - Shift amount < 0 or ≥ word size
    - Signed shift into or out of sign bit (i.e., arith. behavior not assured)

- **Representations in memory, pointers, strings**
  - Bit: fill with 0’s on right
  - Byte: replicate most significant bit on left

- **Unsigned**
  - `short int x = 15213;` \( \rightarrow 00111011 \) \( \rightarrow 01100010 \)

- **Two’s Complement**
  - `short int y = -15213;` \( \rightarrow 01100010 \) \( \rightarrow 11010000 \)

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

Interlude: ChimeIn

https://chimein.cla.umn.edu/course/view/2021

- Which of the following numbers represented in hex is **not** a multiple of 4?
  - `0x2248730c`
  - `0x4d56f9e`
  - `0x2248730c`
  - `0x4d56f9e`
  - `0x69`
  - `0x55`
  - `0xe`
  - `0x69`

  - **Idea:** it is enough to look at the last digit
    - Like divisibility by 10, 2 and 5 for decimal
    - `0x0`, `0x4`, `0x8`, and `0xe` = decimal 12 are multiples of 4
    - `0xe` = 14 is even but not a multiple of 4

Binary Number Property

- **Claim**
  - \[ 1 + 1 + 2 + 4 + 8 + \ldots + 2^{w-1} = 2^w \]
  - \[ \sum_{i=0}^{w-1} 2^i = 2^w \]

- **Proof**
  - \( w = 0 \)
    - 1 = \( 2^0 \)
  - Assume true for \( w-1 \):
    - \( 1 + 1 + 2 + 4 + \ldots + 2^{w-1} = 2^w \)
    - \[ \sum_{i=0}^{w-2} 2^i = 2^w \]
  - \[ \sum_{i=0}^{w} 2^i = 2^{w+1} \]

Today: Bits, Bytes, and Integers

- **Representing information as bits**
- **Bit-level manipulations**
- **Integers**
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
- **Summary**
  - Representations in memory, pointers, strings
  - **Summary**
  - **Encoding Integers**

<table>
<thead>
<tr>
<th>Unsigned</th>
<th>Two’s Complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>( B_{2U}(X) = \sum_{i=0}^{w-1} x_i 2^i )</td>
<td>( B_{2T}(X) = -x_{w-1} 2^{w-1} + \sum_{i=0}^{w-2} x_i 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>0011101111010001</td>
</tr>
<tr>
<td>Y</td>
<td>-15213</td>
<td>C4 93</td>
</tr>
</tbody>
</table>

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

\[ \begin{array}{cccc} 8 & 4 & 2 & 1 \\ 1 & 1 & 1 & 1 \end{array} \]

This approach can represent 0 through 15

\[ 1111 = 15_{10} \]

Four-bit Example, sign + magnitude

\[ \begin{array}{cccc} 8 & 4 & 2 & 1 \\ 1 & 1 & 1 & 1 \end{array} \]

This approach can represent -7 through 7

\[ 1111 = -7_{10} \]

Four-bit Example, two's complement

\[ \begin{array}{cccc} 8 & 4 & 2 & 1 \\ 1 & 1 & 1 & 1 \end{array} \]

This approach can represent -8 through 7

\[ 1111 = -1_{10} \]

Encoding Integers

<table>
<thead>
<tr>
<th>Short</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dec 1</td>
<td>15213</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>Dec 2</td>
<td>00111011 01101101</td>
<td></td>
</tr>
</tbody>
</table>

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

Two-complement Encoding Example (Cont.)

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

\[ \text{Sum} = 15213 \times 1024 + 0 + 0 + 0 = 16384 \]

\[ \text{Sum} = -15213 \times 1024 + 0 + 0 + 0 = -16384 \]
### Numeric Ranges

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

**Two's Complement Values**
- TMin = -2\(^{n-1}\)
- TMax = \(2^{n-1} - 1\) 011...1

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

Values for W = 16

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>65,535</td>
<td>FF FF</td>
</tr>
<tr>
<td>Tmax</td>
<td>127</td>
<td>7F FF</td>
</tr>
<tr>
<td>Tmin</td>
<td>-32,768</td>
<td>80 00</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>4,294,967,295</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
- \(\lfloor T_{\text{Min}} \rfloor = -T_{\text{Max}} \) + 1
- Asymmetric range
- \(U_{\text{Max}} = 2^n - T_{\text{Max}}\)

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

### Unsigned & Signed Numeric Values

<table>
<thead>
<tr>
<th>x</th>
<th>B2U(x)</th>
<th>B2T(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>1010</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>1011</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>1100</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>1101</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>1110</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>1111</td>
<td>15</td>
<td>15</td>
</tr>
</tbody>
</table>

- **Equivalence**
  - Same encodings for nonnegative values

- **Uniqueness**
  - Every bit pattern represents unique integer value
  - Each representable integer has unique bit encoding

- \(\Rightarrow \text{Can Invert Mappings}\)
  - \(U2B(x) = B2U^{-1}(x)\)
  - Bit pattern for unsigned integer
  - \(T2B(x) = B2T^{-1}(x)\)
  - Bit pattern for two's comp integer

### Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary
  - Representations in memory, pointers, strings

### Mapping Between Signed & Unsigned

**Two's Complement**

\[ x \rightarrow \text{\(T2U\)} \rightarrow \text{\(U2T\)} \rightarrow x \]

**Unsigned**

\[ \text{\(U2U\)} \rightarrow x \]

Maintain Same Bit Pattern

**Two's Complement**

\[ \text{\(U2T\)} \rightarrow 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>

+/- 16

Relation between Signed & Unsigned

Two's Complement

<table>
<thead>
<tr>
<th>Signed</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>-8</td>
<td>8</td>
</tr>
<tr>
<td>-7</td>
<td>9</td>
</tr>
<tr>
<td>-6</td>
<td>10</td>
</tr>
<tr>
<td>-5</td>
<td>11</td>
</tr>
<tr>
<td>-4</td>
<td>12</td>
</tr>
<tr>
<td>-3</td>
<td>13</td>
</tr>
<tr>
<td>-2</td>
<td>14</td>
</tr>
<tr>
<td>-1</td>
<td>15</td>
</tr>
</tbody>
</table>

Conversion Visualized

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

- Signed vs. Unsigned in C
  - Constants
    - By default are considered to be signed integers
    - Signed 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 and Comparison 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: TMIN = -2,147,483,648, TMAX = 2,147,483,647
    - Constant1 Constant2 Relation Evaluation
      0 0U == unsigned
      -1 0 < signed
      -1 0U > unsigned
  - Casting and Comparison 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: TMIN = -2,147,483,648, TMAX = 2,147,483,647
    - Constant1 Constant2 Relation Evaluation
      0 0U == unsigned
      -1 0 < signed
      -1 0U > signed
      2147483647 2147483647U < unsigned
      2147483647 -2147483647 > signed
      (unsigned) -1 2 unsigned
      2147483647 2147483648U < unsigned
      2147483647 (int) 2147483648U > signed

https://chimein.cla.umn.edu/course/view/2021
Brief announcements

- About 1 week left to go on HA1
  - Keep your questions coming, don't put it off
- My office hours tomorrow will move to 3-4pm
  - Still in 4-215E Keller

Summary
Casting Signed ↔ Unsigned: Basic Rules

- Bit pattern is maintained
- But reinterpreted
- Can have unexpected effects: adding or subtracting $2^w$
- Expression containing signed and unsigned int
  - int is cast to unsigned!!

Typical Usage

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

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

```c
#define MSIZE 528
void getstuff() {
  char mybuf[MSIZE];
  copy_from_kernel(mybuf, MSIZE);
  printf("%s", mybuf);
}
```

Malicious Usage

```c
/* Declaration of library function memcpy */
void *memcpy(void *dest, void *src, size_t n);
typedef unsigned long size_t;

/* 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;
}
```

```c
#define MSIZE 528
void getstuff() {
  char mybuf[MSIZE];
  copy_from_kernel(mybuf, -MSIZE);
  ...
}
```

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary
- Representations in memory, pointers, strings

Sign Extension

- Task:
  - Given $w$-bit signed integer $x$
  - Convert it to $w+k$-bit integer with same value
- Rule:
  - Make $k$ copies of sign bit:
    - $X' = X_{w-1} \cdots X_{w-k} \cdots X_0$
    - $k$ copies of MSB
    - $X'$
Sign Extension Example

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

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>15213</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>y</td>
<td>-15213</td>
<td>11001100 10010011</td>
</tr>
</tbody>
</table>

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

Summary: Expanding, Truncating: Basic Rules

- Expanding (e.g., short int to int)
  - Unsigned: zeros added
  - Signed: sign extension
  - Both yield expected result
- Truncating (e.g., unsigned to unsigned short)
  - Unsigned/signed: bits are truncated
  - Result reinterpreted
  - Unsigned: mod operation
  - Signed: similar to mod
  - For small numbers yields expected behavior

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
- Representations in memory, pointers, strings
- Summary

Unsigned Addition

Operands: $w$ bits

True Sum: $w+1$ bits

Discard Carry: $w$ bits

UAdd$(u, v) = u + v \mod 2^w$

- Standard Addition Function
  - Ignores carry output
- Implements Modular Arithmetic

Visualizing (Mathematical) Integer Addition

- Integer Addition
  - 4-bit integers $u, v$
  - Compute true sum $Add(u, v)$
  - 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} \quad \text{Overflow}
```

Modular Sum
Mathematical Properties

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

Two's Complement Addition

- Operands: \( w \) bits
  - \( u \)
  - \( v \)
  - True Sum: \( w+1 \) bits
  - \( u \oplus v \)
  - Discard Carry: \( w \) bits
  - \( \text{TAdd}(u, v) \)

- \( \text{TAdd} \) and \( \text{UAdd} \) have Identical Bit-Level Behavior
  - Signed vs. unsigned addition in C:
    - int \( s, t, u, v; \)
    - \( u = \{ (\text{int}) u + (\text{unsigned}) v \}; \)
    - \( t = u + v \)
    - 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 \(< -2^w-1\):
    - Becomes positive
    - At most once

Mathematical Properties of TAdd

- Isomorphic Group to unsigneds with \( \text{UAdd} \)
  - \( \text{TAdd}(u, v) = \text{U2T}(\text{UAdd}(\text{T2U}(u), \text{T2U}(v))) \)
  - 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
    - \( \text{TComp}(u) = \begin{cases} u \text{ if } u \geq 0 \\ 2^w - u \text{ if } u < 0 \end{cases} \)
  - \( \text{TAdd}(u, v) = \begin{cases} u + v \text{ if } u + v \leq \text{TMax}_w \\ \text{TMin}_w \text{ otherwise} \end{cases} \)

Characterizing TAdd

- Functionality
  - True sum requires \( w+1 \) bits
  - Drop off MSB
  - Treat remaining bits as 2’s comp. integer
Unsigned:
- Result range: 0 ≤ x ≤ 2^n - 1
- Also just called "carry" (C)
Signed:
- Result wrong if input signs are different
- In CPUs, unqualified "overflow" usually means signed (O or V)

Goal: Computing Product of Two's Complement min (negative): Up to 2^n - 2^n-1

Claim: Following Holds for 2's Complement
- C = 0 if x > y
- C = x (high) if x < y

Negation: Complement & Increment
- Claim: Following Holds for 2's Complement
  - \( \neg x = 1 - x \)
- Complement
  - Observation: \( \neg x + x = \overline{1111...111} = -1 \)

Multiplication
- Goal: Computing Product of w-bit numbers x, y
  - Either signed or unsigned
  - But, exact results can be bigger than w bits
    - Unsigned: up to 2w bits
      - Result range: 0 ≤ x * y ≤ (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1
    - Two's complement min (negative): Up to 2w-1 bits
      - Result range: \( x \cdot y \geq -2^{w-1}(2^{w-1} - 1) \)
      - Two's complement max (positive): Up to 2w bits, but only for \( T_{\max} \)
        - Result range: \( x \cdot y \leq 2^{w-2} \)
- So, maintaining exact results...
  - Would need to keep expanding word size with each product computed
  - Is done in software, if needed
    - E.g., by "arbitrary precision" arithmetic packages
### Unsigned Multiplication in C

**Operands:** w bits  
**True Product:** 2^w bits  
**Discard w bits:** w bits

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

### Signed Multiplication in C

**Operands:** w bits  
**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

### 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);
```

```c
void *result = malloc(ele_cnt * ele_size);
if (result == NULL) 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) return NULL;
    void *next = result;
    int i;
    for (i = 0; i < ele_cnt; i++) {
        /* Copy object i to destination */
        memcpy(next, ele_src[i], ele_size);
        /* Move pointer to next memory region */
        next += ele_size;
    }
    return result;
}
```

### XDR Vulnerability

```
malloc(ele_cnt * ele_size)
```

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

- **Chime in:** https://chimein.cla.umn.edu/course/view/2021 (Question 16257)

\[ (2^{20} + 1) \cdot (2^{12} + 2^{12} + 2^{12} + 2^{12}) \]

- **How can I make this function secure?**

### Power-of-2 Multiply with Shift

- **Operation**
  - u \ll k gives \( u \cdot 2^k \)
  - Both signed and unsigned

  **Operands:** w bits  
  **True Product:** w+k bits

  ```plaintext
  u \ll k
  \[
  \begin{array}{c}
  k \\
  u \\
  \end{array}
  \]
  \( \left[ \begin{array}{c}
  * & * & * & * & * & * & * & * & * & 0 & 0 & 0 & 0 & 0 & 0 \\
  \end{array} \right] \)
  \]

- **Discard k bits:** w bits

  ```plaintext
  UMult(u, \ll k)
  TMult(u, \ll k)
  \[
  \left[ \begin{array}{c}
  * & * & * & * & * & * & * & * & * & 0 & 0 & 0 & 0 & 0 & 0 \\
  \end{array} \right] \)
  \]

- **Examples**
  - u \ll 3 = u \cdot 8
  - (u \ll 5) - (u \ll 3) = u \cdot 24
  - Most machines shift and add faster than multiply
Compiled Multiplication Code

C Function

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

Compiled Arithmetic Operations

```
leaq (trax.trax,2), trax
salq $2, trax
```

Explanation

- 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 \(-\infty\))
- \( \lceil x \rceil \), read "ceiling": always round up (to \(+\infty\))
- C integer operators mostly use round to zero, which is like floor for positive and ceiling for negative

Division in C

- Integer division \(/\): 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

Undefined behavior

- Many things you should not do are officially called "undefined" by the C language standard
  - Meaning: compiler can do anything it wants
- Examples:
  - Accessing beyond the ends of an array
  - Dividing by zero
    - Overflow in signed operations
    - Shifts of negative values
  - Gap between standard and lenient practical compilers not yet resolved

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:</th>
<th>Binary Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>( u \gg k )</td>
<td>( \lfloor u / 2^k \rfloor )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>( u \gg 0 )</td>
<td>7606</td>
<td>7606</td>
<td>10 86 00011101 01010101</td>
</tr>
<tr>
<td>( u \gg 1 )</td>
<td>950,8125</td>
<td>950</td>
<td>03 86 00000011 10101101</td>
</tr>
<tr>
<td>( u \gg 2 )</td>
<td>59,425,671</td>
<td>59</td>
<td>00 3B 00000000 00111011</td>
</tr>
</tbody>
</table>

Compiled Unsigned Division Code

C Function

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

```
# Logical shift
return x >> 3;
```

Explanation

- 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 $[x \div 2^k]$  
  - Uses arithmetic shift
  - Rounds wrong direction when $u < 0$

<table>
<thead>
<tr>
<th>Operands: $x \div 2^k$</th>
<th>Quotient of Signed by Power of 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>$[x \div 2^k]$</td>
</tr>
<tr>
<td>$2^k$</td>
<td>$[x \div 2^k]$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Division: $x \div 2^k$</th>
<th>Quotient of Signed by Power of 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>$[x \div 2^k]$</td>
</tr>
<tr>
<td>$2^k$</td>
<td>$[x \div 2^k]$</td>
</tr>
</tbody>
</table>

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

Correct Power-of-2 Divide

- Quotient of Negative Number by Power of 2
  - $\lfloor x / 2^k \rfloor$ (Round Toward 0)
  - Compute as $\lfloor (x+2^k-1) / 2^k \rfloor$
  - In C: $(x + (1<<k) - 1) >> k$
  - Biases dividend toward 0

Case 1: No rounding

<table>
<thead>
<tr>
<th>Dividend: $\lfloor x / 2^k \rfloor$</th>
<th>Quotient of Negative Number by Power of 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>$\lfloor x / 2^k \rfloor$</td>
</tr>
<tr>
<td>$2^k$</td>
<td>$\lfloor x / 2^k \rfloor$</td>
</tr>
</tbody>
</table>

Biasing has no effect

Correct Power-of-2 Divide (Cont.)

Case 2: Rounding

<table>
<thead>
<tr>
<th>Dividend: $\lfloor x / 2^k \rfloor$</th>
<th>RoundTowardZero</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>$\lfloor x / 2^k \rfloor$</td>
</tr>
<tr>
<td>$2^k$</td>
<td>$\lfloor x / 2^k \rfloor$</td>
</tr>
</tbody>
</table>

Biasing adds 1 to final result

Compiled Signed Division Code

C Function

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

Compiled Arithmetic Operations

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

- 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 $x \div y$
- E.g., $x \% 10$ is the lowest digit of non-negative $x$
- Behavior for negative values matches $\div$’s rounding toward zero
  - $b \times \lfloor a / b \rfloor + \lfloor a \mod b \rfloor = a$
- 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)

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
- Summary
- Representations in memory, pointers, strings
**Arithmetic: Basic Rules**

- **Addition:**
  - Unsigned/signed: Normal addition followed by truncate, same operation on bit level
  - Unsigned: addition mod 2<sup>n</sup>
    - Mathematical addition + possible subtraction of 2<sup>n</sup>
  - Signed: modified addition mod 2<sup>n</sup> (result in proper range)
  - Mathematical addition + possible addition or subtraction of 2<sup>n</sup>

- **Multiplication:**
  - Unsigned/signed: Normal multiplication followed by truncate, same operation on bit level
  - Unsigned: multiplication mod 2<sup>n</sup>
  - Signed: modified multiplication mod 2<sup>n</sup> (result in proper range)

**Properties of Unsigned Arithmetic**

- **Unsigned Multiplication with Addition Forms**
  - Commutative Ring
  - Addition is commutative group
  - Closed under multiplication
    - 0 ≤ UMult(u, v) ≤ 2<sup>n</sup> – 1
  - Multiplication Commutative
    - UMult(u, v) = UMult(v, u)
  - Multiplication is Associative
    - UMult(t, UMult(u, v)) = UMult(UMult(t, u), v)
  - 1 is multiplicative identity
    - UMult(u, 1) = u
  - Multiplication distributes over addition
    - UMult(u, UAdd(u, v)) = UAdd(UMult(u, t), UMult(u, v))

**Why Should I Use Unsigned?**

- **Don’t use without understanding implications**
- Easy to make mistakes
  ```c
  unsigned i;
  for (i = cnt-2; i >= 0; i--)
      a[i] += a[i+1];
  ```
- Can be very subtle
  ```c
  #define DELTA sizeof(int)
  int i;
  for (i = 0; i-Delta >= 0; i++)
      .
  ```

**Arithmetic: Basic Rules**

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

- **Left shift**
  - Unsigned/signed: multiplication by 2<sup>k</sup>
  - Always logical shift

- **Right shift**
  - Unsigned: logical shift, div (division + round to zero) by 2<sup>k</sup>
  - Signed: arithmetic shift
    - Positive numbers: div (division + round to zero) by 2<sup>k</sup>
    - Negative numbers: div (division + round away from zero) by 2<sup>k</sup>
  - Use biasing to fix

**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<sup>n</sup>

- **Comparison to (Mathematical) Integer Arithmetic**
  - Both are rings
  - Integers obey ordering properties, e.g.,
    - u > 0 ⇒ u + v > 0
    - u > 0, v > 0 ⇒ u + v > 0
  - These properties are not obeyed by two’s comp. arithmetic
  ```c
  TMax + 1 == TMin
  15213 * 30426 == -10030
  ```

**Counting Down with Unsigned**

- **Proper way to use unsigned as loop index**
  ```c
  unsigned i;
  for (i = cnt-2; i < cnt; i--)
      a[i] += a[i+1];
  ```

- **See Robert Seacord, Secure Coding in C and C++**
  - C Standard guarantees that unsigned addition will behave like modular arithmetic
    - 0 – 1 ⇒ UMax

- **Even better**
  ```c
  size_t i;
  for (i = cnt-2; i < cnt; i--)
      a[i] += a[i+1];
  ```
  - Data type size_t defined as unsigned value with length = word size
  - Code will work even if cnt = UMax
  - What if cnt is signed and < 0?
Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
- Summary
- Representations in memory, pointers, strings

Why Should I Use Unsigned? (cont.)

- Do Use When Performing Modular Arithmetic
  - Multiprecision arithmetic
- Do Use When Using Bits to Represent Sets
  - Logical right shift, no sign extension

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
- Summary
- Representations in memory, pointers, strings

Byte-Oriented Memory Organization

- Programs refer to data by address
  - Conceptually, envision it as a very large array of bytes
  - In reality, it’s not, but can think of it that way
  - An address is like an index into that array
  - and, a pointer variable stores an address
- Note: system provides private address spaces to each “process”
  - Think of a process as a program being executed
  - So, a program can clobber its own data, but not that of others

Machine Words

- Any given computer has a “Word Size”
  - Nominal size of integer-valued data
  - and of addresses
  - Until recently, most machines used 32 bits (4 bytes) as word size
    - Limits addresses to 4GB ($2^{32}$ bytes)
  - Increasingly, machines have 64-bit word size
    - Potentially, could have 18 EB (exabytes) of addressable memory
      - That’s $18.4 \times 10^{18}$
  - Machines still support multiple data formats
    - Fractions or multiples of word size
    - Always integral number of bytes

Example Data Representations

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Typical 32-bit</th>
<th>Typical 64-bit</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>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>-</td>
<td>-</td>
<td>10/16</td>
</tr>
<tr>
<td>pointer</td>
<td>4</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>
Byte Ordering

- So, how are the bytes within a multi-byte word ordered in memory?

  - Conventions
    - Big Endian: Sun, PPC Mac, Internet
      - Least significant byte has highest address
    - Little Endian: x86, ARM processors running Android, iOS, and Windows
      - Least significant byte has lowest address

Byte Ordering Example

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

  Big Endian
  - $0x100$: 0123 45 67

  Little Endian
  - $0x100$: 67 45 23 01

Representing Integers

- $\text{int A} = 15213$
  - Decimal: 15213
  - Binary: 0011 1011 0110 1101
  - Hex: 3B 6D

- $\text{int B} = -15213$
  - Two's complement representation

Examining Data Representations

- Code to Print Byte Representation of Data
  - Casting pointer to unsigned char * allows treatment as a byte array

  ```c
  #define show_bytes(start, len)
  {
      size_t i;
      for (i = 0; i < len; i++)
          printf("%p \t0x%.2x\n", start+i, start[i]);
      printf("\n");
  }
  ``

  ```c
  int *P = &B;
  show_bytes((pointer) &P, sizeof(int));
  ``

  Result (Linux x86-64):
  ```
  int a = 15213;
  printf("int a = 15213;\n");
  show_bytes((pointer) &a, sizeof(int));
  ```

  ```
  int a = 15213;
  0x7fffb7f71dbc 6d
  0x7fffb7f71dbd 3b
  0x7fffb7f71dbe 00
  0x7fffb7f71dbf 00
  ```

Representing Pointers

- Different compilers & machines assign different locations to objects
- Even get different results each time run program
**Representing Strings**

- **Strings in C**
  - Represented by array of characters
  - Each character encoded in ASCII format
    - Standard 7-bit encoding of character set
      - Character "0" has code 0x30
        - Digit i has code 0x30+i
  - String should be null-terminated
    - Final character = 0

- **Compatibility**
  - Byte ordering not an issue

---

**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></td>
<td>8048365:</td>
<td>pop ebx</td>
</tr>
<tr>
<td></td>
<td>8048366:</td>
<td>add $0x12ab, ebx</td>
</tr>
<tr>
<td></td>
<td>804836c:</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 00 00

---

**Integer C Puzzles**

1. \( x < 0 \Rightarrow ((x*2) < 0) \)

2. \( ux > -1 \)

3. \( x > 0 \&\& y > 0 \Rightarrow x + y > 0 \)

4. \( (x|-x)>>31 == -1 \)

**Initialization**

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

https://chimein.cla.umn.edu/course/view/2021

---

**Bonus: More Integer C Puzzles**

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

**Initialization**

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