Lecture 4-5: Bits, Bytes, and Integers

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com

Slides adapted from:
• Jinyang Li
• Bryant and O’Hallaron
• Clark Barrett
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
Binary Representations

3.3V
2.8V
0.5V
0.0V

0 1 0
Encoding Byte Values

- **Byte = 8 bits**
  - Binary 00000000₂ to 11111111₂
  - Decimal: 0₁₀ to 255₁₀
  - Hexadecimal 00₁₆ to FF₁₆
    - Base 16 number representation
    - Use characters '0' to '9' and 'A' to 'F'
    - Write FA1D37B₁₆ in C language as
      - 0xFA1D37B
      - 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
  – Until recently, most machines used 32-bit (4-byte) words
    • Limits addresses to 4GB
    • Becoming too small for memory-intensive applications
  – These days, most new systems use 64-bit (8-byte) words
    • Potential address space $\approx 1.8 \times 10^{19}$ bytes
    • x86-64 machines support 48-bit addresses: 256 Terabytes
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 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>pointer</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>
Byte Ordering

• How are 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
    • 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></td>
<td>01</td>
<td>23</td>
<td>45</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

• Example Fragment

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Code</th>
<th>Assembly Rendition</th>
</tr>
</thead>
<tbody>
<tr>
<td>8048365:</td>
<td>5b</td>
<td>pop %ebx</td>
</tr>
<tr>
<td>8048366:</td>
<td>81 c3 ab 12 00 00</td>
<td>add $0x12ab,%ebx</td>
</tr>
<tr>
<td>804836c:</td>
<td>83 bb 28 00 00 00</td>
<td>cmp $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
  - 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\t%2x\n", start+i, start[i]);
    printf("\n");
}
```

printf directives:
%p:  Print pointer
%x:  Print Hexadecimal
show_bytes Execution Example

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

Result (Linux):

```plaintext
int a = 15213;
0x11fffffcb8 0x6d
0x11fffffcb9 0x3b
0x11fffffcb9 0x00
0x11fffffcb9 0x00
```

Note: 15213 in decimal is 3B6D in hexadecimal
Representing Integers

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

int A = 15213;

long int C = 15213;

int B = -15213;

Two’s complement representation (Covered later)
Representing Pointers

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

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
- Byte ordering not an issue

```c
char S[6] = "18243";
```
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
Boolean Algebra

• Developed by George Boole in 19th Century
  – Algebraic representation of logic
    • Encode “True” as 1 and “False” as 0

And

■ A&B = 1 when both A=1 and B=1

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A&amp;B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Or

■ A|B = 1 when either A=1 or B=1

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A&amp;B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Not

■ ~A = 1 when A=0

<table>
<thead>
<tr>
<th>A</th>
<th>~ A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Exclusive-Or (Xor)

■ A^B = 1 when either A=1 or B=1, but not both

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A&amp;B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
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
General Boolean Algebras

• Operate on Bit Vectors
  – Operations applied bitwise

\[
\begin{array}{c}
& 01101001 & 01101001 & 01101001 \\
& \& 01010101 & \| 01010101 & ^ 01010101 & \sim 01010101 \\
\hline
01000001 & 01111101 & 00111100 & 10101010
\end{array}
\]

• 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)**
  – \( \sim 0x41 = 0xBE \)
    • \( \sim 01000001_2 = 1011110_2 \)
  – \( \sim 0x00 = 0xFF \)
    • \( \sim 00000000_2 = 11111111_2 \)
  – \( 0x69 \& 0x55 = 0x41 \)
    • \( 01101001_2 \& 01010101_2 = 01000001_2 \)
  – \( 0x69 \mid 0x55 = 0x7D \)
    • \( 01101001_2 \mid 01010101_2 = 0111101_2 \)
Contrast: Logic Operations in C

- Contrast to Logical Operators
  - &&, |\|, !
    - View 0 as “False”
    - Anything nonzero as “True”
    - Always return 0 or 1
    - Early termination

- Examples (char data type)
  - !0x41 = 0x00
  - !0x00 = 0x01
  - !!0x41 = 0x01
  - 0x69 && 0x55 = 0x01
  - 0x69 |\| 0x55 = 0x01
  - p && *p (avoids null pointer access)
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

| Argument \( x \) | 01100010
|----|----|
| \( << 3 \) | 00010000
| Log. \( >> 2 \) | 00011000
| Arith. \( >> 2 \) | 00011000

| Argument \( x \) | 10100010
|----|----|
| \( << 3 \) | 00010000
| Log. \( >> 2 \) | 00101000
| Arith. \( >> 2 \) | 11101000
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 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**
  - short int \( x = 15213; \)
  - short int \( y = -15213; \)

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>15213</td>
<td>3B 6D</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>( y )</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
</tbody>
</table>

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

• Unsigned Values
  – $U_{\text{min}} = 000..0 = 0$
  – $U_{\text{max}} = 111..1 = 2^w - 1$

• Two’s Complement Values
  – $T_{\text{min}} = 100..0 = -2^{w-1}$
  – $T_{\text{max}} = 011..1 = 2^{w-1} - 1$
  – 111...1 =

Values for $W = 16$

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>65535</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>Tmax</td>
<td>32767</td>
<td>7F FF</td>
<td>01111111 11111111</td>
</tr>
<tr>
<td>Tmin</td>
<td>-32768</td>
<td>80 00</td>
<td>10000000 00000000</td>
</tr>
<tr>
<td></td>
<td>-1</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td></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>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td></td>
</tr>
<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>
<td></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>
<td></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>
<td></td>
</tr>
</tbody>
</table>

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

- **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

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

• Representing information as bits
• Bit-level manipulations
• Integers
  – Representation: unsigned and signed
  – Conversion, casting
  – Expanding, truncating
  – Addition, negation, multiplication, shifting
• Summary
## 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>

Keep bit representations and reinterpret.

+=16
Signed vs. Unsigned in C

- **Constants**
  - By default, signed integers
  - Unsigned with "U" as suffix
    - 0U, 4294967259U

- **Casting**
  - Explicit casting between signed & unsigned
    ```
    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 <, >, ==, <=, >=
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
Expanding

• Convert $w$-bit signed integer to $w+k$-bit with same value
• Convert unsigned: pad $k$ 0 bits in front
• Convert signed: make $k$ copies of sign bit
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>3B 6D 00111011 01101101</td>
</tr>
<tr>
<td>ix</td>
<td>15213</td>
<td>00 00 3B 6D 00000000 00000000 00111011 01101101</td>
</tr>
<tr>
<td>y</td>
<td>-15213</td>
<td>C4 93 11000100 10010011</td>
</tr>
<tr>
<td>iy</td>
<td>-15213</td>
<td>FF FF C4 93 11111111 11111111 11000100 10010011</td>
</tr>
</tbody>
</table>

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

• Example: from int to short (i.e. from 32-bit to 16-bit)
• High-order bits are truncated
• Value is altered → must reinterpret
• The non-intuitive behavior can lead to buggy code!
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

• The complement of $x$ satisfies
  $T\text{Comp}(x) + x = 0$
  $T\text{Comp}(x) = \sim x + 1$

• Proof sketch
  – Observation: $\sim x + x = 1111...111 = -1$

\[
\begin{array}{c}
x \quad \begin{array}{c}1\ 0\ 0\ 1\ 1\ 1\ 0\ 1\end{array}\\
+ \quad \sim x \quad \begin{array}{c}0\ 1\ 1\ 0\ 0\ 0\ 1\ 0\end{array}\\
\hline 
-1 \quad \begin{array}{c}1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\end{array}
\end{array}
\]
## Complement Examples

### x = 15213

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>15213</td>
<td>3B 6D</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>(\neg x)</td>
<td>-15214</td>
<td>C4 92</td>
<td>11000100 10010010</td>
</tr>
<tr>
<td>(\neg x + 1)</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>(T_{\text{comp}}(x))</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
</tbody>
</table>

### x = 0

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00 00</td>
<td>00000000 00000000</td>
</tr>
<tr>
<td>(\neg 0)</td>
<td>-1</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>(\neg 0 + 1)</td>
<td>0</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

- **Standard Addition Function**
  - Ignores carry output
  
  $s = UAdd_w(u, v) = u + v \mod 2^w$
Mathematical Properties

• Modular Addition Forms an Abelian Group
  – \textbf{Closed} under addition
    \[ 0 \leq \text{UAdd}_w(u, v) \leq 2^w - 1 \]
  – \textbf{Commutative}
    \[ \text{UAdd}_w(u, v) = \text{UAdd}_w(v, u) \]
  – \textbf{Associative}
    \[ \text{UAdd}_w(t, \text{UAdd}_w(u, v)) = \text{UAdd}_w(\text{UAdd}_w(t, u), v) \]
  – \textbf{0} is additive identity
    \[ \text{UAdd}_w(u, 0) = u \]
  – Every element has additive \textbf{inverse}
    \[ \text{UAdd}_w(u, \text{Ucomp}(u)) = 0 \]
Two’s Complement Addition

Operands: $w$ bits

$u \quad \cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots $ 

$+ \quad v \quad \cdots $ 

$u + v \quad \cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots $ 

True Sum: $w+1$ bits

Discard Carry: $w$ bits

$TAdd_w(u, v) \quad \cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots $

- $TAdd$ and $UAdd$ have Identical Bit-Level Behaviour
- $TAdd$ treat remaining bits as 2’s comp. integer
  - If sum $\geq 2^{w-1}$, becomes negative (positive overflow)
  - If sum $< -2^{w-1}$, becomes positive (negative overflow)
Mathematical Properties of TAdd

- Two’s Complement Under TAdd Forms a Group
  - Closed, Commutative, Associative, 0 is additive identity
  - Every element has additive inverse

\[ T\text{Comp}_w(u) = \begin{cases} 
-u & u \neq T\text{Min}_w \\
T\text{Min}_w & u = T\text{Min}_w 
\end{cases} \]
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 $2^w$ 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 $2^{w-1}$ bits
  - **Two’s complement max:** $x \times y \leq (-2^{w-1})^2 = 2^{2w-2}$
    - Up to $2^w$ bits, but only for $(TMin^w)^2$

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

Operands: $w$ bits

True Product: $2w$ bits

Discard $w$ bits: $w$ bits

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

• 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);
```
XDR Code

```c
void* copy_elements(void *ele_src[], int ele_cnt, size_t ele_size) {
    /*
    * Allocate buffer for ele_cnt objects, each of ele_size bytes
    * and copy from locations designated by ele_src
    */
    void* result = malloc(ele_cnt * ele_size);
    if (result == NULL)
        /* malloc failed */
        return NULL;
    void* next = result;
    int i;
    for (i = 0; i < ele_cnt; i++) {
        /* Copy object i to destination */
        memcpy(next, ele_src[i], ele_size);
        /* Move pointer to next memory region */
        next += ele_size;
    }
    return result;
}
```
XDR Vulnerability

\[ \text{malloc(ele\_cnt} \times \text{ele\_size}) \]

- **What if:**
  - \( \text{ele\_cnt} = 2^{20} + 1 \)
  - \( \text{ele\_size} = 4096 = 2^{12} \)
  - Allocation = ??

- **How can I make this function secure?**
Power-of-2 Multiply with Shift

- **Operation**
  - \( u << k \) gives \( u \times 2^k \)
  - Both signed and unsigned

- **Examples**
  - \( u << 3 \) == \( u \times 8 \)
  - \((u << 5) - (u << 3)\) == \( u \times 24 \)
  - Most machines shift and add faster than multiply
    - Compiler generates this code automatically

Operands: \( w \) bits

True Product: \( w+k \) bits

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

UMult\(_w\)(\( u \), \( 2^k \))

TMult\(_w\)(\( u \), \( 2^k \))
Compiled Multiplication Code

C Function

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

Compiled Arithmetic Operations

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

Explanation

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

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

### Quotient of Unsigned by Power of 2

- \( u \gg k \) gives \( \lfloor u / 2^k \rfloor \)

### Division Computed Hex Binary

<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>15213</td>
<td>3B 6D 00111011 01101101</td>
</tr>
<tr>
<td>x &gt;&gt; 1</td>
<td>7606.5</td>
<td>7606</td>
<td>1D B6 00011101 10110110</td>
</tr>
<tr>
<td>x &gt;&gt; 4</td>
<td>950.8125</td>
<td>950</td>
<td>03 B6 00000011 10110110</td>
</tr>
<tr>
<td>x &gt;&gt; 8</td>
<td>59.4257813</td>
<td>59</td>
<td>00 3B 00000000 00111011</td>
</tr>
</tbody>
</table>
Compiled Unsigned Division Code

C Function

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

Compiled Arithmetic Operations

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

Explanation

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

- Uses logical shift for unsigned
- For Java Users
  - Logical shift written as >>>
Signed Power-of-2 Divide with Shift

- **Quotient of Signed by Power of 2**
  - $x \gg k$ gives $\lfloor x / 2^k \rfloor$
  - Uses arithmetic shift
  - Rounds wrong direction when $u < 0$

- **Operands:**
  - $x$
  - $2^k$

- **Result:**
  - RoundDown($x / 2^k$)

- **Division Computed Hex Binary**
<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y$</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>$y \gg 1$</td>
<td>-7606.5</td>
<td>E2 49</td>
<td>11100010 01001001</td>
</tr>
<tr>
<td>$y \gg 4$</td>
<td>-950.8125</td>
<td>FC 49</td>
<td>11111100 01001001</td>
</tr>
<tr>
<td>$y \gg 8$</td>
<td>-59.4257813</td>
<td>FF C4</td>
<td>11111111 11000100</td>
</tr>
</tbody>
</table>
Correct Power-of-2 Divide

- Quotient of Negative Number by Power of 2
  - Want \( \left\lfloor \frac{x}{2^k} \right\rfloor \) (Round Toward 0)
  - Compute as \( \left\lfloor \frac{x+2^k-1}{2^k} \right\rfloor \)
    - In C: \( (x + (1<<k)-1) >> k \)
    - Biases dividend toward 0

Exploiting the property that:
\[
\left\lfloor \frac{x}{y} \right\rfloor = \left\lfloor \frac{x+y-1}{y} \right\rfloor
\]
Compiled Signed Division Code

C Function

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

Compiled Arithmetic Operations

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

Explanation

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

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

• Everything is stored in memory as 1s and 0s
• The binary presentation by itself does not carry a meaning, it depends on the interpretation.
• When to use signed and when to use unsigned?