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

![Diagram showing binary representations with voltage levels at 0.0V, 0.5V, 2.8V, 3.3V, and 0.0V, and a binary sequence 0 1 0.](image-url)
Encoding Byte Values

- **Byte = 8 bits**
  - Binary \(00000000_2\) to \(11111111_2\)
  - Decimal: \(0_{10}\) to \(255_{10}\)
  - Hexadecimal \(00_{16}\) to \(FF_{16}\)
    - Base 16 number representation
    - Use characters '0' to '9' and 'A' to 'F'
    - Write \(FA1D37B_{16}\) in C language as
      - 0xFA1D37B
      - 0xfa1d37b

<table>
<thead>
<tr>
<th>Hex</th>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>0111</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>A</td>
<td>10</td>
<td>1010</td>
</tr>
<tr>
<td>B</td>
<td>11</td>
<td>1011</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td>1100</td>
</tr>
<tr>
<td>D</td>
<td>13</td>
<td>1101</td>
</tr>
<tr>
<td>E</td>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>F</td>
<td>15</td>
<td>1111</td>
</tr>
</tbody>
</table>
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 manipulate 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>01</td>
<td>23</td>
<td>45</td>
<td>67</td>
</tr>
</tbody>
</table>

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

• Disassembly
  – Text representation of binary machine code

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

• Deciphering Numbers
  – Value: 0x12ab
  – Pad to 32 bits: 0x000012ab
  – Split into bytes: 00 00 12 ab
  – Reverse: ab 12 00 00
Examining Data Representations

- Code to print Byte Representation of data

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):

```c
int a = 15213;
0x11ffffffcb8 0x6d
0x11ffffffcb9 0x3b
0x11ffffffcba 0x00
0x11ffffffcbb 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;

IA32, x86-64       Sun
| 6D  | 00  |
| 3B  | 00  |
| 00  | 00  |
| 00  | 00  |

long int C = 15213;

IA32
| 6D  |
| 3B  |
| 00  |
| 00  |

x86-64
| 6D  |
| 3B  |
| 00  |
| 00  |

Sun
| 00  |
| 00  |
| 3B  |
| 6D  |

IA32, x86-64       Sun
| 93  | FF  |
| C4  | FF  |
| FF  | C4  |
| FF  | 93  |

Two’s complement representation (Covered later)

int B = -15213;

IA32, x86-64       Sun
| 93  | FF  |
| C4  | FF  |
| FF  | 93  |
| FF  | C4  |

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

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

- $A \mid B = 1$ 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

- $\sim A = 1$ when $A=0$

<table>
<thead>
<tr>
<th>A</th>
<th>$\sim A$</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)

- $A \oplus B = 1$ 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

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

• 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)**
  – ~0x41 = 0xBE
    • ~01000001₂ = 1011110₂
  – ~0x00 = 0xFF
    • ~00000000₂ = 1111111₂
  – 0x69 & 0x55 = 0x41
    • 01101001₂ & 01010101₂ = 01000001₂
  – 0x69 | 0x55 = 0x7D
    • 01101001₂ | 01010101₂ = 0111101₂
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 ≥ word size

<table>
<thead>
<tr>
<th>Argument ( x )</th>
<th>01100010</th>
</tr>
</thead>
<tbody>
<tr>
<td>( &lt;&lt; 3 )</td>
<td>00010000</td>
</tr>
<tr>
<td>Log. ( &gt;&gt; 2 )</td>
<td>00011000</td>
</tr>
<tr>
<td>Arith. ( &gt;&gt; 2 )</td>
<td>00011000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Argument ( x )</th>
<th>10100010</th>
</tr>
</thead>
<tbody>
<tr>
<td>( &lt;&lt; 3 )</td>
<td>00010000</td>
</tr>
<tr>
<td>Log. ( &gt;&gt; 2 )</td>
<td>00101000</td>
</tr>
<tr>
<td>Arith. ( &gt;&gt; 2 )</td>
<td>11101000</td>
</tr>
</tbody>
</table>
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

<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**
  - $U_{min} = 000..0 = 0$
  - $U_{max} = 111..1 = 2^w - 1$

- **Two's Complement Values**
  - $T_{min} = 100..0 = -2^{w-1}$
  - $T_{max} = 011..1 = 2^{w-1} - 1$
  - $111...1 = -1$

Values for $W = 16$

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>65535</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>Tmax</td>
<td>32767</td>
<td>7F FF</td>
<td>01111111 11111111</td>
</tr>
<tr>
<td>Tmin</td>
<td>-32768</td>
<td>80 00</td>
<td>10000000 00000000</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>
Values for Different Word Sizes

<table>
<thead>
<tr>
<th>W</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>255</td>
<td>65,535</td>
<td>4,294,967,295</td>
<td>18,446,744,073,709,551,615</td>
</tr>
<tr>
<td>Tmax</td>
<td>127</td>
<td>32,767</td>
<td>2,147,483,647</td>
<td>9,223,372,036,854,775,807</td>
</tr>
<tr>
<td>Tmin</td>
<td>-128</td>
<td>-32,768</td>
<td>-2,147,483,648</td>
<td>-9,223,372,036,854,775,808</td>
</tr>
</tbody>
</table>

- **Observations**
  - \(|T_{\text{Min}}| = T_{\text{Max}} + 1\)
    - Asymmetric range
  - \(U_{\text{Max}} = 2 \times T_{\text{Max}} + 1\)

- **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 a unique integer value
  - Each representable integer has a 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 for mapping.

\[ +/\ 16 \]

\[ = \]
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></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>ix</td>
<td>15213</td>
<td>00 00 3B 6D</td>
<td>00000000 00000000 00111011 01101101</td>
</tr>
<tr>
<td>y</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>iy</td>
<td>-15213</td>
<td>FF FF C4 93</td>
<td>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 $\rightarrow$ 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
  $$\text{TComp}(x) + x = 0$$
  $$\text{TComp}(x) = \sim x + 1$$

• Proof sketch
  – Observation: $\sim x + x = 1111\ldots11 = -1$

\[
\begin{array}{c}
x & 1 0 0 1 1 1 0 1 \\
+ & \sim x & 0 1 1 0 0 0 1 0 \\
\hline
-1 & 1 1 1 1 1 1 1 1
\end{array}
\]
**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>(~x)</td>
<td>-15214</td>
<td>C4 92</td>
<td>11000100 10010010</td>
</tr>
<tr>
<td>(~x+1)</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>( T_{complement}(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>(~0)</td>
<td>-1</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>(~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
  \[ s = \text{UAdd}_w(u, v) = u + v \mod 2^w \]
### Two’s Complement Addition

- **Operands**: $w$ bits
  - $u$
  - $v$

- **True Sum**: $w+1$ bits
  - $u + v$

- **Discard Carry**: $w$ bits
  - $\text{TAdd}_w(u, v)$

#### Rules:
- **TAdd and UAdd have Identical Bit-Level Behavior**
- $-2^{w-1} \leq u, v \leq 2^{w-1} - 1$
- **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)
Multiplication

• Computing Exact Product of \( w \)-bit numbers \( x, y \)
  - Either signed or unsigned

• Ranges
  - Unsigned: \( 0 \leq x \times y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1 \)
    • Up to \( 2w \) bits
  - Two’s complement min: \( x \times y \geq (-2^{w-1}) \times (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1} \)
    • Up to \( 2w-1 \) bits
  - Two’s complement max: \( x \times y \leq (-2^{w-1})^2 = 2^{2w-2} \)
    • Up to \( 2w \) bits, but only for \( (TMin_w)^2 \)

• Maintaining Exact Results
  - Would need to keep expanding word size with each product computed
Unsigned/Signed Multiplication in C

- **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);
```
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

malloc(ele_cnt * ele_size)

• What if:
  - ele_cnt = \(2^{20} + 1\)
  - 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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>( u &lt;&lt; k )</td>
<td>( u \times 2^k )</td>
</tr>
<tr>
<td>((u &lt;&lt; 5) - (u &lt;&lt; 3))</td>
<td>( u \times 24 )</td>
</tr>
</tbody>
</table>

• **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
**Compiled Multiplication Code**

C Function

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

Compiled Arithmetic Operations

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

Explanation

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

- *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
  \[ u \gg k \text{ gives } \lfloor u / 2^k \rfloor \]

Operands:

\[
\begin{array}{c}
\begin{array}{c}
\text{Operand} \\
\text{u}
\end{array}
\begin{array}{c}
\text{Shift} \\
\text{k}
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\text{u} \\
\text{0 \cdots 0}
\end{array}
\begin{array}{c}
\text{k}
\end{array}
\end{array}
\]

Result:

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

<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>3B 6D</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>( x \gg 1)</td>
<td>7606.5</td>
<td>1D B6</td>
<td>00011101 10110110</td>
</tr>
<tr>
<td>( x \gg 4)</td>
<td>950.8125</td>
<td>03 B6</td>
<td>00000011 10110110</td>
</tr>
<tr>
<td>( x \gg 8)</td>
<td>59.4257813</td>
<td>00 3B</td>
<td>00000000 00111011</td>
</tr>
</tbody>
</table>
Compiled Unsigned Division Code

C Function

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

Compiled Arithmetic Operations

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

Explanation

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

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

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

**Operands:**

\[
x / 2^k \begin{array}{c}
0 \cdots 0 \underbrace{1} \cdots \underbrace{0} \cdots \underbrace{0}
\end{array}
\]

**Result:**

\[
\text{RoundDown}(x / 2^k) \begin{array}{c}
\cdots
\end{array}
\]

<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>11100001 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>
Trick! Correct Power-of-2 Divide

- Quotient of Negative Number by Power of 2
  - Want \( \lceil x / 2^k \rceil \) (Round Toward 0)
  - Compute as \( \lfloor (x + 2^k - 1) / 2^k \rfloor \)

- In C: \( (x + (1 << k) - 1) \gg k \)

Exploiting the property that:
\[
\lceil x / y \rceil = \lfloor (x + y - 1) / y \rfloor
\]
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

```c
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?