Some slides adapted and modified from:
• Clark Barrett
• Jinyang Li
• Bryant and O’Hallaron
Bits and Bytes

- Representing information as bits
- How bits are manipulated?
- Integers
- Floating points
High-level language program (in C)

```c
swap(int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

Assembly language program

```assembly
swap:
    muli $2, $5, 4
    add $2, $4, $2
    lw $15, 0($2)
    lw $16, 4($2)
    sw $16, 0($2)
    sw $15, 4($2)
    jr $31
```

Binary machine language program

```
0000000001010000100000000011000
000000000001100001100000110001
10001100110010010000100000000000
10001101111010000000000010
100110111110100000000000000000
100011001110100000000000000000
100110111110100000000000000000
10001100110010010000100000000000
000000111100000000000000001000
```
Our First Steps...
How do we represent data in a computer?

• How do we represent information using electrical signals?
• At the lowest level, a computer is an electronic machine.
• Easy to recognize two conditions:
  – presence of a voltage - we call this state “1”
  – absence of a voltage - we call this state “0”
Binary Representations

0.0V
0.5V
2.8V
3.3V

0 1 0
A Computer is a Binary Digital Machine

- Basic unit of information is the binary digit, or bit.
- Values with more than two states require multiple bits.
  - A collection of two bits has four possible states: 00, 01, 10, 11
  - A collection of three bits has eight possible states: 000, 001, 010, 011, 100, 101, 110, 111
  - A collection of \( n \) bits has \( 2^n \) possible states.
George Boole

- (1815-1864)
- English mathematician and philosopher
- Inventor of Boolean Algebra
- Now we can use things like: AND, OR, NOT, XOR, XNOR, NAND, NOR, ....

Source: http://history-computer.com/ModernComputer/thinkers/Boole.html
Claude Shannon

- (1916–2001)
- American mathematician and electronic engineer
- His work is the foundation for using switches (i.e. transistors), and hence binary numbers, to implement Boolean function.

Source: http://history-computer.com/ModernComputer/thinkers/Shannon.html
So, we use **transistors to implement logic gates.**

Logic gates manipulate binary numbers to implement **Boolean functions.**

Boolean functions solve problems.

.... Simply Speaking ... 😊
Encoding Byte Values

- **Byte = 8 bits**
  - Binary: 00000000₂ to 11111111₂
  - Decimal: 0₁₀ to 255₁₀
  - Hexadecimal: 00₁₆ to FF₁₆
    - Base 16 number representation
    - Every 4 bits → 1 hexadecimal digit
    - Use characters '0' to '9' and 'A' to 'F'
    - Write FA1D37B₁₆ in C language as
      - 0xFA1D37B
      - 0xfa1d37b
## 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>

The most widely used in servers, desktops, laptops, ... .
Byte Ordering

- How are bytes within a multi-byte word ordered in memory?

- Conventions
  - **Big Endian**: Sun, PPC, Internet
    - Most significant byte has lowest address
  - **Little Endian**: x86
    - Most significant byte has highest address
Byte Ordering Example

- **Big Endian**
  - Most significant byte has lowest address
- **Little Endian**
  - Most significant byte has highest 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**
  - given the binary file, get the assembly

- **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 (int is 4 bytes): 0x000012ab
  - Split into bytes: 00 00 12 ab
  - Reverse (little endian): ab 12 00 00
Examining Data Representations

• Code to print Byte Representation of data

```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):

```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;
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

• **Byte ordering not an issue**

```
char S[6] = "18243";
```
How to Manipulate Bits?
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

| A | B | A|B |
|---|---|----|
| 0 | 0 | 0  |
| 0 | 1 | 1  |
| 1 | 0 | 1  |
| 1 | 1 | 1  |

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^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

Transistor
General Boolean Algebras

- Operate on Bit Vectors (e.g. an integer is a bit vector of 4 bytes = 32 bits)
  - Operations applied bitwise
    
    | 01101001 | 01101001 | 01101001 |
    |----------|----------|----------|
    | 01010101 | 01010101 | 01010101 |
    |----------|----------|----------|
    | & 01000001 | 01111101 | 00111100 | ~ 10101010 |
Bit-Level Operations in C

- **Operations &**, **|**, **~**, **^** Available in C
  - Apply to any “integral” data type
    - long, int, short, char, unsigned

- **Examples (Char data type)**
  - ~0x41 = 0xBE
    - ~01000001\_2 = 10111110\_2
  - ~0x00 = 0xFF
    - ~00000000\_2 = 11111111\_2
  - 0x69 & 0x55 = 0x41
    - 01101001\_2 & 01010101\_2 = 01000001\_2
  - 0x69 | 0x55 = 0x7D
    - 01101001\_2 | 01010101\_2 = 01111101\_2
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:** \( x \ll y \)
  - Shift \( x \) left by \( y \) positions
    - Throw away extra bits on left
    - Fill with 0’s on right
• **Right Shift:** \( x \gg y \)
  - Shift \( x \) right \( y \) positions
    - Throw away extra bits on right
    - type 1: Logical shift
      - Fill with 0’s on left
    - type 2: Arithmetic shift (covered later)
      - Replicate most significant bit on right
• **Undefined Behavior**
  - Shift amount < 0 or ≥ size of \( x \)
How to present Integers? (unsigned and signed)
Two Type of Integers

• Unsigned
  – positive numbers and 0
• Signed numbers
  – negative numbers as well as positive numbers and 0
Unsigned Integers

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

1 0 1 1 1 1 0 1 1

128 64 32 16 8 4 2 1

187
Unsigned Integers

- An $n$-bit unsigned integer represents $2^n$ values: from 0 to $2^n-1$. 

<table>
<thead>
<tr>
<th></th>
<th>$2^2$</th>
<th>$2^1$</th>
<th>$2^0$</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>
Unsigned Binary Arithmetic

- Base-2 addition - just like base-10
  - add from right to left, propagating carry

\[
\begin{align*}
10010 + 1001 &= 11011 \\
10010 + 1011 &= 11101 \\
10010 + 1 &= 10000
\end{align*}
\]
## What About Negative Numbers?

People have tried several options:

<table>
<thead>
<tr>
<th>Sign Magnitude:</th>
<th>One's Complement</th>
<th>Two's Complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 = +0</td>
<td>000 = +0</td>
<td>000 = +0</td>
</tr>
<tr>
<td>001 = +1</td>
<td>001 = +1</td>
<td>001 = +1</td>
</tr>
<tr>
<td>010 = +2</td>
<td>010 = +2</td>
<td>010 = +2</td>
</tr>
<tr>
<td>011 = +3</td>
<td>011 = +3</td>
<td>011 = +3</td>
</tr>
<tr>
<td>100 = -0</td>
<td>100 = -3</td>
<td>100 = -4</td>
</tr>
<tr>
<td>101 = -1</td>
<td>101 = -2</td>
<td>101 = -3</td>
</tr>
<tr>
<td>110 = -2</td>
<td>110 = -1</td>
<td>110 = -2</td>
</tr>
<tr>
<td>111 = -3</td>
<td>111 = -0</td>
<td>111 = -1</td>
</tr>
</tbody>
</table>

- **Issues**: balance, number of zeros, ease of operations
- **Which one is best?** Why?
Signed Integers

• With \( n \) bits, we have \( 2^n \) distinct values.
  – assign about half to positive integers and about half to negative

• Positive integers
  – just like unsigned: zero in most significant (MS) bit
    \( 00101 = 5 \)

• Negative integers
  – In two’s complement form

In general: a 0 at the MS bit indicates positive and a 1 indicates negative.
Two’s Complement

- Two’s complement representation developed to make circuits easy for arithmetic.
  - for each positive number (X), assign value to its negative (-X), such that \( X + (-X) = 0 \) with “normal” addition, ignoring carry out.

\[
\begin{align*}
00101 & \quad (5) \\
+ 11011 & \quad (-5) \\
\hline
00000 & \quad (0)
\end{align*}
\]

\[
\begin{align*}
01001 & \quad (9) \\
+ 10111 & \quad (-9) \\
\hline
00000 & \quad (0)
\end{align*}
\]
Two’s Complement Signed Integers

- **MS bit is sign bit.**
- **Range of an n-bit number:** \(-2^{n-1}\) through \(2^{n-1} - 1\).
  - The most negative number \((-2^{n-1})\) has no positive counterpart.

<table>
<thead>
<tr>
<th>(-2^3)</th>
<th>(2^2)</th>
<th>(2^1)</th>
<th>(2^0)</th>
<th>(-2^3)</th>
<th>(2^2)</th>
<th>(2^1)</th>
<th>(2^0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Converting Binary (2’s C) to Decimal

1. If MS bit is one, take two’s complement to get a positive number.

2. Get the decimal as if the number is unsigned (using power of 2s).

3. If original number was negative, add a minus sign.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$2^n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
</tr>
<tr>
<td>6</td>
<td>64</td>
</tr>
<tr>
<td>7</td>
<td>128</td>
</tr>
<tr>
<td>8</td>
<td>256</td>
</tr>
<tr>
<td>9</td>
<td>512</td>
</tr>
<tr>
<td>10</td>
<td>1024</td>
</tr>
</tbody>
</table>
Examples

\[ X = 00100111_{\text{two}} \]
\[ = 2^5 + 2^2 + 2^1 + 2^0 = 32 + 4 + 2 + 1 \]
\[ = 39_{\text{ten}} \]

\[ X = 11100110_{\text{two}} \]
\[ -X = 00011010 \]
\[ = 2^4 + 2^3 + 2^1 = 16 + 8 + 2 \]
\[ = 26_{\text{ten}} \]
\[ X = -26_{\text{ten}} \]

\[ \begin{array}{|c|c|}
\hline
n & 2^n \\
\hline
0 & 1 \\
1 & 2 \\
2 & 4 \\
3 & 8 \\
4 & 16 \\
5 & 32 \\
6 & 64 \\
7 & 128 \\
8 & 256 \\
9 & 512 \\
10 & 1024 \\
\hline
\end{array} \]
## Numeric Ranges

### Example: Assume 16-bit numbers

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unsigned</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max</td>
<td>65535</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td><strong>Signed</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max</td>
<td>32767</td>
<td>7F FF</td>
<td>01111111 11111111</td>
</tr>
<tr>
<td>Min</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 Sizes

<table>
<thead>
<tr>
<th></th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unsig. Max</strong></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><strong>Signed Max</strong></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><strong>Signed Min</strong></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>

## C Programming

- `#include <limits.h>`
- Declares constants, e.g.,
  - `ULONG_MAX`
  - `LONG_MAX`
  - `LONG_MIN`
- Values platform specific
What happens if you change the type of a variable (aka type casting)?
# 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>
Signed vs. Unsigned in C

• Constants
  – By default, signed integers
  – Unsigned with “U” as suffix
    0U, 4294967259U

• Casting
  – Explicit casting between signed & unsigned
    ```c
    int tx, ty;
    unsigned ux, uy;
    tx = (int) ux;
    uy = (unsigned) ty;
    ```
  – Implicit casting also occurs via assignments and procedure calls
    ```c
    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 `<`, `>`, `==`, `<=`, `>=`

If there is an expression that has many types, the compiler follows these rules.
Example

#include <stdio.h>

main() {

    int i = 17;
    char c = 'c'; /* ascii value is 99 */
    float sum;

    sum = i + c;
    printf("Value of sum : %f\n", sum);
}

Source: https://www.tutorialspoint.com/cprogramming/c_type_casting.htm
Expanding and Truncating a variable
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

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</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</td>
<td>C4 93</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 $\implies$ must reinterpret
- This non-intuitive behavior can lead to buggy code! $\implies$ So don’t do it!
Addition, negation, multiplication, and shifting
Negation: Complement & Increment

• The complement of $x$ satisfies
  \[ \text{Two'sComp}(x) + x = 0 \]
  \[ \text{Two'sComp}(x) = \sim x + 1 \]

• Proof sketch
  – Observation: $\sim x + x = 1111...111 = -1$
    $\Rightarrow \sim x + x + 1 = 0$
    $\Rightarrow (\sim x + 1) + x = 0$
    $\Rightarrow \text{Two'sComp}(x) + x = 0$

\[
\begin{array}{c}
\text{x} & 100111101 \\
+ \sim \text{x} & 01100010 \\
\hline
-1 & 111111111 \\
\end{array}
\]
Unsigned Addition

Operands: \( w \) bits

True Sum: \( w+1 \) bits

Discard Carry: \( w \) bits
Two’s Complement Addition

Operands: \( w \) bits

\[
\begin{array}{c}
\quad u \\
+ \quad v \\
\hline
u + v
\end{array}
\]

True Sum: \( w+1 \) bits

Discard Carry: \( w \) bits

\[
\text{TAdd}_w(u, v)
\]

- If sum \( \geq 2^{w-1} \), becomes negative (positive overflow)
- If sum \( < -2^{w-1} \), becomes positive (negative overflow)
Multiplication

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

• Ranges
  – Unsigned: \( 0 \leq x \cdot y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1 \)
  – Two’s complement min: \( x \cdot y \geq (-2^{w-1}) \cdot (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1} \)
  – Two’s complement max: \( x \cdot y \leq (-2^{w-1})^2 = 2^{2w-2} \)
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
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
  \[ u \gg k \text{ gives } \lfloor u / 2^k \rfloor \]

Examples:

<table>
<thead>
<tr>
<th></th>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>15213</td>
<td>15213</td>
<td>3B 6D</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>x &gt;&gt; 1</td>
<td>7606.5</td>
<td>7606</td>
<td>1D B6</td>
<td>00011101 10110110</td>
</tr>
<tr>
<td>x &gt;&gt; 4</td>
<td>950.8125</td>
<td>950</td>
<td>03 B6</td>
<td>00000011 10110110</td>
</tr>
<tr>
<td>x &gt;&gt; 8</td>
<td>59.4257813</td>
<td>59</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 \( \lfloor \frac{x}{2^k} \rfloor \)
  - Uses arithmetic shift

Examples

<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>
Floating Points
Background: Fractional binary numbers

• What is $1011.101_2$?
Background: Fractional Binary Numbers

- Value:
  \[ \sum_{k=-j}^{i} b_k \times 2^k \]
### Fractional Binary Numbers: Examples

<table>
<thead>
<tr>
<th>Value</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 3/4</td>
<td>101.11&lt;sub&gt;2&lt;/sub&gt;</td>
</tr>
<tr>
<td>2 7/8</td>
<td>10.111&lt;sub&gt;2&lt;/sub&gt;</td>
</tr>
</tbody>
</table>
Why not fractional binary numbers?

• Not efficient

  $3 \times 2^{100} \rightarrow 1010000000 \ldots 0$

  100 zeros

  Given a finite length (e.g. 32-bits), cannot represent very large nor very small numbers ($\varepsilon \rightarrow 0$)
IEEE Floating Point

• IEEE Standard 754
  – Supported by all major CPUs

• Driven by numerical concerns
  – Standards for rounding, overflow, underflow
  – Hard to make fast in hardware
    • Numerical analysts predominated over hardware designers in defining standard
Floating Point Representation

• Numerical Form: $(-1)^s \ M \ 2^E$
  
  – Sign bit $s$ determines whether number is negative or positive
  
  – Significand $M$ a fractional value
  
  – Exponent $E$ weights value by power of two

• Encoding
  
  – MSB $s$ is sign bit $s$
  
  – $exp$ field encodes $E$
  
  – $frac$ field encodes $M$
Precisions

• **Single precision: 32 bits**

  - $s$ (1-bit)
  - `exp` (8-bits)
  - `frac` (23-bits)

• **Double precision: 64 bits**

  - $s$ (1-bit)
  - `exp` (11-bits)
  - `frac` (52-bits)

• **Extended precision: 80 bits (Intel only)**

  - $s$ (1-bit)
  - `exp` (15-bits)
  - `frac` (63 or 64-bits)
Based on $\exp$ we have 3 encoding schemes

- $\exp \neq 0..0$ or $11...1 \rightarrow$ normalized encoding
- $\exp = 0...000 \rightarrow$ denormalized encoding
- $\exp = 1111...1 \rightarrow$ special value encoding
  - $\text{frac} = 000...0$
  - $\text{frac} = \text{something else}$
1. Normalized Encoding

- **Condition:** \( \text{exp} \neq 000...0 \) and \( \text{exp} \neq 111...1 \)
  
  \( \text{Exponent is: } E = \text{Exp} - (2^{k-1} - 1) \), \( k \) is the # of exponent bits
  
  - Single precision: \( E = \text{exp} - 127 \)
  - Double precision: \( E = \text{exp} - 1023 \)

- **Significand is:** \( M = 1.xxx...x_2 \)
  
  - Range(\( M \)) = \([1.0, 2.0-\epsilon]\)
  - Get extra leading bit for free
Normalized Encoding Example

• Value: Float F = 15213.0;
  \(15213_{10} = 11101101101101_2\)
  \(= 1.1101101101101_2 \times 2^{13}\)

• Significand
  \(M = 1.\overline{1101101101101}_2\)
  \(\frac{\text{frac}}{1101101101101000000000000}_2\)

• Exponent
  \(E = \exp - \text{Bias} = \exp - 127 = 13\)
  \(\Rightarrow \exp = 140 = 10001100_2\)

• Result:
  \[
  \begin{array}{c|c|c}
    s & \exp & \frac{\text{frac}}{1101101101101000000000000}_2
  \end{array}
  \]
2. Denormalized Encoding

- **Condition:** \( \text{exp} = 000\ldots0 \)

- **Exponent value:** \( E = 1 - \text{Bias} \) (instead of \( E = 0 - \text{Bias} \))

- **Significand is:** \( M = 0.xxx\ldots x_2 \) (instead of \( M=1.xxxx_2 \))

- **Cases**
  - \( \text{exp} = 000\ldots0, \, \text{frac} = 000\ldots0 \)
    - Represents zero
    - Note distinct values: +0 and -0
  - \( \text{exp} = 000\ldots0, \, \text{frac} \neq 000\ldots0 \)
    - Numbers very close to 0.0
3. Special Values Encoding

- **Condition**: \( \exp = 111...1 \)

- **Case**: \( \exp = 111...1, \frac{}{} = 000...0 \)
  - Represents value \( \infty \) (infinity)
  - Operation that overflows
  - E.g., \( 1.0/0.0 = -1.0/-0.0 = +\infty, \ 1.0/-0.0 = -\infty \)

- **Case**: \( \exp = 111...1, \frac{}{} \neq 000...0 \)
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., \( \sqrt{-1}, \infty - \infty, \infty \times 0 \)
Visualization: Floating Point Encodings
Floating Point in C

- **C:**
  - float  single precision
  - double  double precision

- **Conversions/Casting**
  - Casting between int, float, and double changes bit representation, examples:
    - double/float → int
      - Truncates fractional part
      - Not defined when out of range or NaN
    - int → double
      - Exact conversion
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.
• IEEE Floating Point has clear mathematical properties
  – Represents numbers as: \((-1)^S \times M \times 2^E\)