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

0.0V  0.5V  2.8V  3.3V
0   1   0
Encoding Byte Values

- **Byte = 8 bits**
  - Binary 00000000₂ to 1111111₁₀
  - 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 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
## 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, 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 0x101 0x102 0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>01  23  45  67</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Little Endian</th>
<th>0x100 0x101 0x102 0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>67  45  23  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

```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;
0x11fffffcb8 0x6d
0x11fffffcb9 0x3b
0x11fffffcba 0x00
0x11ffffffcbb 0x00
```

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

Decimal: 15213
Binary: 0011 1011 0110 1101
Hex: 3B6D

int A = 15213;

IA32, x86-64

6D 3B 00 00

Sun

00 00 3B 6D

int B = -15213;

IA32, x86-64

93 C4 FF FF

Sun

FF FF C4 93

IA32

6D 3B 00 00

x86-64

6D 3B 00 00

Sun

00 00 3B 6D

long int C = 15213;

Two’s complement representation (Covered later)
Representing Pointers

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 $$B=1$$

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

Not
- $$\sim A = 1$$ 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
General Boolean Algebras

- Operate on Bit Vectors
  - Operations applied bitwise

\[
\begin{align*}
01101001 & \quad 01101001 & \quad 01101001 \\
\& 01010101 & | 01010101 & ^ 01010101 & ~ 01010101 \\
01000001 & 01111101 & 00111100 & 10101010
\end{align*}
\]
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 = 10111110₂
  - ~0x00 = 0xFF
    - ~00000000₂ = 11111111₂
  - 0x69 & 0x55 = 0x41
    - 01101001₂ & 01010101₂ = 01000001₂
  - 0x69 | 0x55 = 0x7D
    - 01101001₂ | 01010101₂ = 01111101₂
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 by \( y \) positions
  - Throw away extra bits on left
  - Fill with 0’s on right

• **Right Shift:** \( x >> y \)
  - Shift bit-vector \( x \) right by \( y \) 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
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
  – $U_{\text{min}} = 000..0 = 0$
  – $U_{\text{max}} = 111..1 = 2^w - 1$

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

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></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>
</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>
</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**
  - $|T_{Min}| = T_{Max} + 1$
  - Asymmetric range
  - $U_{Max} = 2 * T_{Max} + 1$

- **C Programming**
  - `#include <limits.h>`
  - Declares constants, e.g.,
    - `ULONG_MAX`
    - `LONG_MAX`
    - `LONG_MIN`
  - Values platform specific
Today: Bits, Bytes, and Integers

• Representing information as bits
• Bit-level manipulations
• Integers
  – Representation: unsigned and signed
  – Conversion, casting
  – Expanding, truncating
  – Addition, negation, multiplication, shifting
• Summary
## Mapping Signed $\leftrightarrow$ 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*
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 → 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: $\sim x + x = 1111...111 = -1$

\[
\begin{array}{c}
x \\
+ \sim x \\
\hline
-1
\end{array}
\]

\[
\begin{array}{c}
10011101 \\
01100010 \\
\hline
11111111
\end{array}
\]
## 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>Tcomp(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

\[ u \quad \cdots \quad \cdots \quad \cdots \quad \cdots \quad u + v \quad \cdots \quad \cdots \quad \cdots \quad \cdots \quad u + v \mod 2^w \]

- **Standard Addition Function**
  - Ignores carry output
    \[ s = UAdd_w(u, v) = u + v \mod 2^w \]
Two's Complement Addition

Operands: $w$ bits

True Sum: $w+1$ bits

Discard Carry: $w$ bits

- 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 $2^w$ 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
Power-of-2 Multiply with Shift

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

  Operands: \( w \) bits

  \[
  \begin{array}{c}
  \text{True Product: } w+k \text{ bits} \\
  \hline \\
  u \\
  \end{array}
  \]

  \[
  \begin{array}{c}
  \text{Discard } k \text{ bits: } w \text{ bits} \\
  \hline \\
  u \cdot 2^k \\
  \hline \\
  k
  \end{array}
  \]

  \[
  \begin{array}{c}
  \text{u} \\
  \hline \\
  0 \cdots 0 1 0 \cdots 0 0
  \end{array}
  \]

  \[
  \begin{array}{c}
  u \cdot 2^k \\
  \hline \\
  0 \cdots 0 1 0 \cdots 0 0
  \end{array}
  \]

  \[
  \begin{array}{c}
  \cdots \cdots 0 \cdots 0 0
  \end{array}
  \]

• **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

```c
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
  \(-u >> k\) gives \(\lfloor u / 2^k \rfloor\)

<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

```
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
  - $\text{x} \gg k$ gives $\left\lfloor \frac{x}{2^k} \right\rfloor$
  - Uses arithmetic shift

Operands:

$$x \div 2^k$$

Result:

RoundDown($\frac{x}{2^k}$)

<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>
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?