Number Systems and Number Representation

Q: Why do computer programmers confuse Christmas and Halloween?
A: Because 25 Dec = 31 Oct

Goals of this Lecture

Help you learn (or refresh your memory) about:
- The binary, hexadecimal, and octal number systems
- Finite representation of unsigned integers
- Finite representation of signed integers
- Finite representation of rational (floating-point) numbers

Why?
- A power programmer must know number systems and data representation to fully understand C’s primitive data types

Primitives values and the operations on them

Agenda

Number Systems
Finite representation of unsigned integers
Finite representation of signed integers
Finite representation of rational (floating-point) numbers

The Decimal Number System

Name
- “decem” (Latin) ⇒ ten

Characteristics
- Ten symbols
  - 0 1 2 3 4 5 6 7 8 9
- Positional
  - $2945 = (2 \times 10^3) + (9 \times 10^2) + (4 \times 10^1) + (5 \times 10^0)$

(Most) people use the decimal number system

Why?

The Binary Number System

binary
 adjective: being in a state of one of two mutually exclusive conditions such as on or off, true or false, molten or frozen, presence or absence of a signal.
From Late Latin binarius (“consisting of two”).

Characteristics
- Two symbols
  - 0 1
- Positional
  - $1010_2 \neq 1100_2$

Most (digital) computers use the binary number system

Terminology
- Bit: a binary digit
- Byte: (typically) 8 bits
- Nibble (or nybble): 4 bits

Decimal-Binary Equivalence

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

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>10000</td>
</tr>
<tr>
<td>17</td>
<td>10001</td>
</tr>
<tr>
<td>18</td>
<td>10010</td>
</tr>
<tr>
<td>19</td>
<td>10011</td>
</tr>
<tr>
<td>20</td>
<td>10100</td>
</tr>
<tr>
<td>21</td>
<td>10101</td>
</tr>
<tr>
<td>22</td>
<td>10110</td>
</tr>
<tr>
<td>23</td>
<td>10111</td>
</tr>
<tr>
<td>24</td>
<td>11000</td>
</tr>
<tr>
<td>25</td>
<td>11001</td>
</tr>
<tr>
<td>26</td>
<td>11010</td>
</tr>
<tr>
<td>27</td>
<td>11011</td>
</tr>
<tr>
<td>28</td>
<td>11100</td>
</tr>
<tr>
<td>29</td>
<td>11101</td>
</tr>
<tr>
<td>30</td>
<td>11110</td>
</tr>
<tr>
<td>31</td>
<td>11111</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Decimal-Binary Conversion

Binary to decimal: expand using positional notation

\[100101_2 = (1\times2^5)+(0\times2^4)+(0\times2^3)+(1\times2^2)+(0\times2^1)+(1\times2^0)\]
\[= 32 + 0 + 0 + 4 + 0 + 1\]
\[= 37\]

Least-significant bit (lsb)
Most-significant bit (msb)

Integer-Binary Conversion

Integer to binary: do the reverse

- Determine largest power of 2 ≤ number; write template

\[37 = (\_\times2^5)+(\_\times2^4)+(\_\times2^3)+(\_\times2^2)+(\_\times2^1)+(\_\times2^0)\]

- Fill in template

\[37 = (1\times2^5)+(0\times2^4)+(0\times2^3)+(1\times2^2)+(0\times2^1)+(1\times2^0)\]
\[-32\]
\[-4\]
\[-1\]
\[1\]
\[0\]
\[100101_2\]

Integer to binary shortcut

- Repeatedly divide by 2, consider remainder

\[37 / 2 = 18 \text{ R } 1\]
\[18 / 2 = 9 \text{ R } 0\]
\[9 / 2 = 4 \text{ R } 1\]
\[4 / 2 = 2 \text{ R } 0\]
\[2 / 2 = 1 \text{ R } 0\]
\[1 / 2 = 0 \text{ R } 1\]

Read from bottom to top: \(100101_2\)

The Hexadecimal Number System

Name
- "hexa" (Greek) ⇒ six
- "decent" (Latin) ⇒ ten

Characteristics
- Sixteen symbols
  - \(0 1 2 3 4 5 6 7 8 9 A B C D E F\)
- Positional
  - \(A13D_{16} \neq 3DA1_{16}\)

Computer programmers often use hexadecimal
- In C: 0x prefix (0xA13D, etc.)

Decimal-Hexadecimal Equivalence

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Decimal</th>
<th>Hex</th>
<th>Decimal</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>16</td>
<td>10</td>
<td>32</td>
<td>20</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>17</td>
<td>11</td>
<td>33</td>
<td>21</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>18</td>
<td>12</td>
<td>34</td>
<td>22</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>19</td>
<td>13</td>
<td>35</td>
<td>23</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>20</td>
<td>14</td>
<td>36</td>
<td>24</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>21</td>
<td>15</td>
<td>37</td>
<td>25</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>22</td>
<td>16</td>
<td>38</td>
<td>26</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>23</td>
<td>17</td>
<td>39</td>
<td>27</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>24</td>
<td>18</td>
<td>40</td>
<td>28</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>25</td>
<td>19</td>
<td>41</td>
<td>29</td>
</tr>
<tr>
<td>10</td>
<td>A</td>
<td>26</td>
<td>1A</td>
<td>42</td>
<td>2A</td>
</tr>
<tr>
<td>11</td>
<td>B</td>
<td>27</td>
<td>1B</td>
<td>43</td>
<td>2B</td>
</tr>
<tr>
<td>12</td>
<td>C</td>
<td>28</td>
<td>1C</td>
<td>44</td>
<td>2C</td>
</tr>
<tr>
<td>13</td>
<td>D</td>
<td>29</td>
<td>1D</td>
<td>45</td>
<td>2D</td>
</tr>
<tr>
<td>14</td>
<td>E</td>
<td>30</td>
<td>1E</td>
<td>46</td>
<td>2E</td>
</tr>
<tr>
<td>15</td>
<td>F</td>
<td>31</td>
<td>1F</td>
<td>47</td>
<td>2F</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
### Integer-Hexadecimal Conversion

**Hexadecimal to integer: expand using positional notation**

\[
25_{16} = (2\times16^1) + (5\times16^0) \\
= 32 + 5 \\
= 37
\]

**Integer to hexadecimal: use the shortcut**

\[
\begin{align*}
37 & \div 16 = 2 \text{ R } 5 \\
2 & \div 16 = 0 \text{ R } 2
\end{align*}
\]

Read from bottom to top: \(25_{16}\)

---

### Binary-Hexadecimal Conversion

**Observation:** \(16^1 = 2^4\)

- Every 1 hexadecimal digit corresponds to 4 binary digits

**Binary to hexadecimal**

\[
\begin{align*}
1010000100111101 & \quad \text{A 1 3 D}_8 \\
\end{align*}
\]

Digit count in binary number not a multiple of 4 ⇒ pad with zeros on left

**Hexadecimal to binary**

\[
\begin{align*}
\text{A 1 3 D}_8 & \quad 1010000100111101_B \\
\end{align*}
\]

Discard leading zeros from binary number if appropriate

Is it clear why programmers often use hexadecimal?

---

### iClicker Question

Q: Convert binary 101010 into decimal and hex

A. 21 decimal, 1A hex
B. 42 decimal, 2A hex
C. 48 decimal, 32 hex
D. 55 decimal, 4G hex

---

### The Octal Number System

**Name**

- “octo” (Latin) ⇒ eight

**Characteristics**

- Eight symbols:
  - 0 1 2 3 4 5 6 7
- Positional:
  - \(1743_8 \neq 7314_8\)

Computer programmers often use octal (so does Mickey!)

- In C: 0 prefix (01743, etc.)

Why?

---

### Integral Types in Java vs. C

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unsigned types</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>char</td>
<td>// 16 bits</td>
<td>unsigned char /* 8 bits */</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned short</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned (int)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned long</td>
</tr>
<tr>
<td><strong>Signed types</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>byte</td>
<td>// 8 bits</td>
<td>signed char /* Note 2 */</td>
</tr>
<tr>
<td>int</td>
<td>// 16 bits</td>
<td>(signed) short</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(signed) int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(signed) long</td>
</tr>
<tr>
<td>long</td>
<td>// 64 bits</td>
<td></td>
</tr>
<tr>
<td><strong>Floating-point types</strong></td>
<td></td>
<td>float</td>
</tr>
<tr>
<td></td>
<td></td>
<td>double</td>
</tr>
<tr>
<td></td>
<td></td>
<td>long double</td>
</tr>
</tbody>
</table>

1. Not guaranteed by C, but on coursela, char = 8 bits, short = 16 bits, int = 32 bits, long = 64 bits, float = 32 bits, double = 64 bits
2. Not guaranteed by C, but on coursela, char is signed

To understand C, must consider representation of both unsigned and signed integers
Representing Unsigned Integers

Mathematics
• Range is 0 to $\infty$

Computer programming
• Range limited by computer’s word size
• Word size is $n$ bits $\Rightarrow$ range is 0 to $2^n - 1$
• Exceed range $\Rightarrow$ overflow

Typical computers today
• $n = 32$ or 64, so range is 0 to $2^{32} - 1$ or $2^{64} - 1$ (huge!)

Pretend computer
• $n = 4$, so range is 0 to $2^4 - 1$ (15)

Hereafter, assume word size = 4
• All points generalize to word size = 64, word size $= n$

Adding Unsigned Integers

Addition

<table>
<thead>
<tr>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
</tr>
<tr>
<td>+ 10</td>
</tr>
<tr>
<td>--</td>
</tr>
<tr>
<td>13</td>
</tr>
</tbody>
</table>

Start at right column
Proceed leftward
Carry 1 when necessary

Start at right column
Proceed leftward
Carry 1 when necessary

Beware of overflow

How would you detect overflow programatically?

Results are mod $2^4$

Subtracting Unsigned Integers

Subtraction

<table>
<thead>
<tr>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
</tr>
<tr>
<td>- 7</td>
</tr>
<tr>
<td>--</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

Start at right column
Proceed leftward
Borrow when necessary

Start at right column
Proceed leftward
Borrow when necessary

Beware of overflow

How would you detect overflow programatically?

Results are mod $2^4$

Shifting Unsigned Integers

Bitwise right shift ($>>$ in C): fill on left with zeros

$10 >> 1 \Rightarrow 5$

$1010<sub>b</sub> \ 0101<sub>b</sub>$

$10 >> 2 \Rightarrow 2$

$1010<sub>b</sub> \ 0010<sub>b</sub>$

Bitwise left shift ($<<$ in C): fill on right with zeros

$5 << 1 \Rightarrow 10$

$0101<sub>b</sub> \ 1010<sub>b</sub>$

$3 << 2 \Rightarrow 12$

$0011<sub>b</sub> \ 1100<sub>b</sub>$

What is the effect arithmetically?
(No fair looking ahead)

What is the effect arithmetically?
(No fair looking ahead)

Results are mod $2^4$

Other Operations on Unsigned Ints

Bitwise NOT (~ in C)
• Flip each bit

$\neg 10 \Rightarrow 5$

$1010<sub>b</sub> \ 0101<sub>b</sub>$

Bitwise AND (& in C)
• Logical AND corresponding bits

$10 \ & 7 \ & 0111<sub>b</sub>$

Useful for setting selected bits to 0
Other Operations on Unsigned Ints

Bitwise OR: (| in C)
- Logical OR corresponding bits
  - Useful for setting selected bits to 1

Bitwise exclusive OR (^ in C)
- Logical exclusive OR corresponding bits
  - x ^ x sets all bits to 0

Aside: Using Bitwise Ops for Arith
Can use <<, >>, and & to do some arithmetic efficiently
- Fast way to multiply by a power of 2
  x * 2^y == x << y
  - 3*4 = 3*2^2 = 3<<2 ⇒ 12
- Fast way to divide unsigned by power of 2
  x / 2^y == x >> y
  - 13/4 = 13/2^2 = 13>>2 ⇒ 3
- Fast way to mod by a power of 2
  x % 2^y == x & (2^y-1)
  - 13%4 = 13%2^2 = 13&(2^2-1) = 13&3 ⇒ 1

Many compilers will do these transformations automatically!

Aside: Example C Program
```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    unsigned int n;
    unsigned int count = 0;
    printf("Enter an unsigned integer: ");
    if (scanf("%u", &n) != 1)
    {
        fprintf(stderr, "Error: Expect unsigned int.\n");
        exit(EXIT_FAILURE);
    }
    while (n > 0)
    {
        count += (n & 1);
        n = n >> 1;
    }
    printf("%u\n", count);
    return 0;
}
```

What does it write?

How could you express this more succinctly?

Agenda

Number Systems
Finite representation of unsigned integers

Finite representation of signed integers
Finite representation of rational (floating-point) numbers

Sign-Magnitude

<table>
<thead>
<tr>
<th>Integer</th>
<th>Rep</th>
</tr>
</thead>
<tbody>
<tr>
<td>-7</td>
<td>1111</td>
</tr>
<tr>
<td>-6</td>
<td>1110</td>
</tr>
<tr>
<td>-5</td>
<td>1101</td>
</tr>
<tr>
<td>-4</td>
<td>1100</td>
</tr>
<tr>
<td>-3</td>
<td>1011</td>
</tr>
<tr>
<td>-2</td>
<td>1010</td>
</tr>
<tr>
<td>-1</td>
<td>1001</td>
</tr>
<tr>
<td>0</td>
<td>1000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
</tr>
</tbody>
</table>

Definition
High-order bit indicates sign
- 0 ⇒ positive
- 1 ⇒ negative

Remaining bits indicate magnitude
- $1101_2 = -101_2 = -5$
- $0101_2 = 101_2 = 5$
Sign-Magnitude (cont.)

<table>
<thead>
<tr>
<th>Integer</th>
<th>Rep</th>
</tr>
</thead>
<tbody>
<tr>
<td>-7</td>
<td>1111</td>
</tr>
<tr>
<td>-6</td>
<td>1110</td>
</tr>
<tr>
<td>-5</td>
<td>1101</td>
</tr>
<tr>
<td>-4</td>
<td>1100</td>
</tr>
<tr>
<td>-3</td>
<td>1011</td>
</tr>
<tr>
<td>-2</td>
<td>1010</td>
</tr>
<tr>
<td>-1</td>
<td>1001</td>
</tr>
<tr>
<td>0</td>
<td>0100</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
</tr>
</tbody>
</table>

Computing negative
- \( \text{neg} \) = flip high order bit of \( x \)
- \( \text{neg}(0101_{\text{B}}) = 1101_{\text{B}} \)
- \( \text{neg}(1101_{\text{B}}) = 0101_{\text{B}} \)

Pros and cons
- easy for people to understand
- symmetric
- two representations of zero
- need different algorithms to add signed and unsigned numbers

Ones’ Complement

<table>
<thead>
<tr>
<th>Integer</th>
<th>Rep</th>
</tr>
</thead>
<tbody>
<tr>
<td>-8</td>
<td>1000</td>
</tr>
<tr>
<td>-7</td>
<td>1001</td>
</tr>
<tr>
<td>-6</td>
<td>1010</td>
</tr>
<tr>
<td>-5</td>
<td>1011</td>
</tr>
<tr>
<td>-4</td>
<td>1100</td>
</tr>
<tr>
<td>-3</td>
<td>1101</td>
</tr>
<tr>
<td>-2</td>
<td>1110</td>
</tr>
<tr>
<td>-1</td>
<td>1111</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
</tr>
</tbody>
</table>

Definition
- High-order bit has weight -7
- \( \text{neg}(0101_{\text{B}}) = 1101_{\text{B}} \)
- \( \text{neg}(1101_{\text{B}}) = 0101_{\text{B}} \)

Computing negative
- \( \text{neg} \) = \( \neg x \)
- \( \text{neg}(0101_{\text{B}}) = 1010_{\text{B}} \)
- \( \text{neg}(1010_{\text{B}}) = 0101_{\text{B}} \)

Similar pros and cons to sign-magnitude

Two’s Complement

<table>
<thead>
<tr>
<th>Integer</th>
<th>Rep</th>
</tr>
</thead>
<tbody>
<tr>
<td>-8</td>
<td>1000</td>
</tr>
<tr>
<td>-7</td>
<td>1001</td>
</tr>
<tr>
<td>-6</td>
<td>1010</td>
</tr>
<tr>
<td>-5</td>
<td>1011</td>
</tr>
<tr>
<td>-4</td>
<td>1100</td>
</tr>
<tr>
<td>-3</td>
<td>1101</td>
</tr>
<tr>
<td>-2</td>
<td>1110</td>
</tr>
<tr>
<td>-1</td>
<td>1111</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
</tr>
</tbody>
</table>

Definition
- High-order bit has weight -8
- \( \text{neg}(0101_{\text{B}}) = 1010_{\text{B}} + 1 = 1011_{\text{B}} \)
- \( \text{neg}(1011_{\text{B}}) = 0100_{\text{B}} + 1 = 0101_{\text{B}} \)

Computing negative
- \( \text{neg}(\neg x + 1) \)
- \( \text{neg}(\text{onescomp}(x) + 1) \)
- \( \text{neg}(0101_{\text{B}}) = 1010_{\text{B}} \)
- \( \text{neg}(1010_{\text{B}}) = 0101_{\text{B}} \)

Pros and cons
- not symmetric
- one representation of zero
- same algorithm adds unsigned numbers or signed numbers

Almost all computers today use two’s complement to represent signed integers
- Arithmetic is easy!

Hereafter, assume two’s complement.
**Adding Signed Integers**

<table>
<thead>
<tr>
<th>pos + pos</th>
<th>pos + pos (overflow)</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>111</td>
</tr>
<tr>
<td>3 + 3</td>
<td>+ 1 + 0011</td>
</tr>
<tr>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>6</td>
<td>-8</td>
</tr>
<tr>
<td>0110B</td>
<td>1000B</td>
</tr>
</tbody>
</table>

**Negating Signed Ints: Math**

**Question:** Why does two’s comp arithmetic work?

**Answer:** \([-b] \mod 2^4 = [\text{twoscomp}(b)] \mod 2^4\)

\[
[-b] \mod 2^4 = [2^4 - b] \mod 2^4 = [2^4 - 1 - b + 1] \mod 2^4 = [(2^4 - 1 - b) + 1] \mod 2^4 = [\text{onescomp}(b) + 1] \mod 2^4 = [\text{twoscomp}(b)] \mod 2^4
\]

See Bryant & O’Hallaron book for much more info

**Subtracting Signed Integers**

<table>
<thead>
<tr>
<th>Performing subtraction with borrow or Compute two’s comp and add</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>- 4</td>
</tr>
<tr>
<td>--</td>
</tr>
<tr>
<td>-1</td>
</tr>
</tbody>
</table>

<p>| -5               | 1011B           |</p>
<table>
<thead>
<tr>
<th>-2</th>
<th>-0100B</th>
</tr>
</thead>
<tbody>
<tr>
<td>-7</td>
<td>1010B</td>
</tr>
</tbody>
</table>

How would you detect overflow programmatically?

**Subtracting Signed Integers: Math**

And so:\n
\([a - b] \mod 2^4 = [a + \text{twoscomp}(b)] \mod 2^4\)

\[
[a - b] \mod 2^4 = [a + 2^4 - b] \mod 2^4 = [a + 2^4 - 1 - b + 1] \mod 2^4 = [a + (2^4 - 1 - b) + 1] \mod 2^4 = [a + \text{onescomp}(b) + 1] \mod 2^4 = [a + \text{twoscomp}(b)] \mod 2^4
\]

See Bryant & O’Hallaron book for much more info

**Shifting Signed Integers**

**Bitwise left shift (<< in C): fill on right with zeros**

3 << 1 = 6
0011B → 0110B

-3 << 1 = -6
1101B → -1010B

**Bitwise arithmetic right shift: fill on left with sign bit**

6 >> 1 = 3
0110B → 0011B

-6 >> 1 = -3
1010B → 1101B

Results are mod 2^4

* The C language does not provide an arithmetic right shift operator, although the hardware (machine language) does.

**Shifting Signed Integers (cont.)**

**Bitwise logical right shift: fill on left with zeros**

6 >> 1 = 3
0110B → 0011B

-6 >> 1 = 5
1010B → 0101B

What is the effect arithmetically???

In C, right shift (>>) could be logical or arithmetic:

- Not specified by C90 standard
- Compiler designer decides (typically it’s logical right shift)

Best to avoid shifting signed integers
Other Operations on Signed Ints

- Bitwise NOT (~ in C)
  - Same as with unsigned ints
- Bitwise AND (& in C)
  - Same as with unsigned ints
- Bitwise OR: (| in C)
  - Same as with unsigned ints
- Bitwise exclusive OR (^ in C)
  - Same as with unsigned ints

Best to avoid with signed integers

Agenda

- Number Systems
- Finite representation of unsigned integers
- Finite representation of signed integers
- Finite representation of rational (floating-point) numbers

Rational Numbers

Mathematics
- A rational number is one that can be expressed as the ratio of two integers
- Unbounded range and precision

Computer science
- Finite range and precision
- Approximate using floating point number

Floating Point Numbers

Like scientific notation: e.g., \( c \) is \( 2.99792458 \times 10^8 \text{ m/s} \)

This has the form \((\text{multiplier}) \times (\text{base})^{\text{power}}\)

In the computer,
- Multiplier is called mantissa
- Base is almost always 2
- Power is called exponent

IEEE Floating Point Representation

Common finite representation: IEEE floating point
- More precisely: ISO/IEEE 754 standard

Using 32 bits (type float in C):
- 1 bit: sign (0 = positive, 1 = negative)
- 8 bits: exponent + 127
- 23 bits: binary fraction of the form \( 1.bbbbbbbbbbbbbbbbbbbbb \)

Using 64 bits (type double in C):
- 1 bit: sign (0 = positive, 1 = negative)
- 11 bits: exponent + 1023
- 52 bits: binary fraction of the form \( 1.bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb \)

Floating Point Example

Sign (1 bit):
- 1 \( \Rightarrow \) negative

Exponent (8 bits):
- 10000011 = 131
- 131 − 127 = 4

Fraction (23 bits): also called “mantissa”
- 1.101101100000000000000000
- \( 1 + (1 \times 2^{-1}) + (0 \times 2^{-2}) + (1 \times 2^{-3}) + (1 \times 2^{-4}) + (0 \times 2^{-5}) + (1 \times 2^{-6}) + (1 \times 2^{-7}) = 1.7109375 \)

Number:
- \(-1.7109375 \times 2^4 = -27.375 \)
When was floating-point invented?

**Answer:** long before computers!

**mantissa**

noun  
decimal part of a logarithm, 1865, from Latin mantissa “a worthless addition, makeweight,” perhaps a Gaulish word introduced into Latin via Etruscan (cf. Old Irish met, Welsh naint “size”).

---

**Floating Point Consequences**

"Machine epsilon": smallest positive number you can add to 1.0 and get something other than 1.0

For float: $\varepsilon \approx 10^{-7}$
- No such number as 1.000000001
- Rule of thumb: "almost 7 digits of precision"

For double: $\varepsilon \approx 2 \times 10^{-16}$
- Rule of thumb: "not quite 16 digits of precision"

These are all relative numbers

---

**Floating Point Consequences, cont.**

Decimal number system can represent only some rational numbers with finite digit count
  - Example: 1/3 *cannot* be represented

Binary number system can represent only some rational numbers with finite digit count
  - Example: 1/5 *cannot* be represented

Beware of **roundoff error**
  - Error resulting from inexact representation
  - Can accumulate
  - Be careful when comparing two floating-point numbers for equality

---

**iClicker Question**

Q: What does the following code print?

```c
double sum = 0.0;
int i;
for (i = 0; i < 10; i++)
    sum += 0.1;
if (sum == 1.0)
    printf("All good!\n");
else
    printf("Yikes!\n");
```

A. All good!  
B. Yikes!  
C. Code crashes  
D. Code enters an infinite loop

---

**Summary**

The binary, hexadecimal, and octal number systems

Finite representation of unsigned integers
Finite representation of signed integers
Finite representation of rational (floating-point) numbers

Essential for proper understanding of
  - C primitive data types
  - Assembly language
  - Machine language