Operations and Arithmetic

Floating point representation
Operations in C

Have the data, what now?

- Boolean operations
- Logical operations
- Arithmetic operations
Boolean Algebra

Algebraic representation of logic

- Encode “True” as 1 and “False” as 0
- Operators & | ~ ^

AND (&)

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

A&B = 1 when both A=1 and B=1

OR (|)

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

A|B = 1 when either A=1 or B=1

NOT (~)

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

~A = 1 when A=0

XOR/EXCLUSIVE-OR (^)

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

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

- 3 –
In C

Operators the same (&, |, ~, ^)

- Apply to any “integral” data type
  - long, int, short, char
  - View arguments as bit vectors
  - Arguments applied bit-wise

- Examples

```
01101001 & 01010101 = 01000001
01101001 | 01010101 = 01111101
01101001 ^ 01010101 = 00111100
01010101 ~ 01010101 = 10101010
```
Practice problem

\[ 0x69 \& 0x55 \]

\[ 0x69 \^ 0x55 \]

\[ 0x69 \mid 0x55 \]

\[ \sim 0x55 \]
Practice problem

0x69 & 0x55

01101001
01010101
01000001 = 0x41

0x69 | 0x55

01101001
01010101
01111101 = 0x7D

0x69 ^ 0x55

01101001
01010101
00111100 = 0x3C

~0x55

01010101
10101010 = 0xAA
Shift Operations

Left Shift: \( x \ll y \)
- Shift bit-vector \( x \) left \( y \) positions
  - Throw away extra bits on left
  - Fill with 0’s on right

Right Shift: \( x \gg y \)
- 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 left
  - Recall two’s complement integer representation
  - Perform division by 2 via shift

<table>
<thead>
<tr>
<th>Argument x</th>
<th>01100010</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x \ll 3 )</td>
<td>00010000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Argument x</th>
<th>10100010</th>
</tr>
</thead>
<tbody>
<tr>
<td>Log. ( x \gg 2 )</td>
<td>00101000</td>
</tr>
<tr>
<td>Arith. ( x \gg 2 )</td>
<td>11101000</td>
</tr>
</tbody>
</table>
### Practice problem

<table>
<thead>
<tr>
<th>$x$</th>
<th>$x &lt;&lt; 3$</th>
<th>$x &gt;&gt; 2$ (Logical)</th>
<th>$x &gt;&gt; 2$ (Arithmetic)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xf0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xf0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0f</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0f</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xcc</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xcc</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x55</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x55</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Practice problem

<table>
<thead>
<tr>
<th>x</th>
<th>x&lt;&lt;3</th>
<th>x&gt;&gt;2  (Logical)</th>
<th>x&gt;&gt;2  (Arithmetic)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xf0</td>
<td>0x80</td>
<td>0x3c</td>
<td>0xfc</td>
</tr>
<tr>
<td>0x0f</td>
<td>0x78</td>
<td>0x03</td>
<td>0x03</td>
</tr>
<tr>
<td>0xc0</td>
<td>0x60</td>
<td>0x33</td>
<td>0xf3</td>
</tr>
<tr>
<td>0x55</td>
<td>0xa8</td>
<td>0x15</td>
<td>0x15</td>
</tr>
</tbody>
</table>
Logic Operations in C

Operations always return 0 or 1

Comparison operators
- >, >=, <, <=, ==, !=

Logical Operators
- &&, ||, !
  - Logical AND, Logical OR, Logical negation
  - 0 is “False”, anything nonzero is “True”

Examples (char data type)
- !0x41  -->  0x00
- !0x00  -->  0x01
- !!0x41  -->  0x01

What are the values of:
- 0x69  ||  0x55
- 0x69  |  0x55
- What does this expression do?  (p && *p)
Logical vs. Bitwise operations

Watch out

- Logical operators versus bitwise boolean operators
- && versus &
- || versus |
- == versus =

But on Nov. 5, 2003, Larry McVoy noticed that there was a code change in the CVS copy that did not have a pointer to a record of approval. Investigation showed that the change had never been approved and, stranger yet, that this change did not appear in the primary BitKeeper repository at all. Further investigation determined that someone had apparently broken in (electronically) to the CVS server and inserted this change.

What did the change do? This is where it gets really interesting. The change modified the code of a Linux function called wait4, which a program could use to wait for something to happen. Specifically, it added these two lines of code:

```c
if ((options == (__WCLONE|__WALL)) && (current->uid == 0))
    retval = -EINVAL;
```

Using Bitwise and Logical operations

```c
int x, y;

For any processor, independent of the size of an integer, write C expressions without any "=" signs that are true if:

- x and y have any non-zero bits in common in their low order byte
- x has any 1 bits at higher positions than the low order 8 bits
- x is zero
- x == y
```
Using Bitwise and Logical operations

```c
int x, y;

For any processor, independent of the size of an integer, write C expressions without any "=" signs that are true if:

- x and y have any non-zero bits in common in their low order byte
  ```c
  0xff & (x & y)
  ```
- x has any 1 bits at higher positions than the low order 8 bits
  ```c
  ~0xff & x
  ```
  ```c
  (x & 0xff)^x
  ```
  ```c
  (x >> 8)
  ```
- x is zero
  ```c
  !x
  ```
- x == y
  ```c
  !(x^y)
  ```
```
Arithmetic operations

Signed/unsigned

- Addition and subtraction
- Multiplication
- Division
Unsigned addition

Suppose we have a computer with 4-bit words

What is the unsigned value of 7 + 7?
- $0111 + 0111$

What about 9 + 9?
- $1001 + 1001$

With $w$ bits, unsigned addition is regular addition, modulo $2^w$
- Bits beyond $w$ are discarded
Unsigned addition

With 32 bits, unsigned addition is modulo what?

What is the value of 0xc0000000 + 0x70004444 ?

```c
#include <stdio.h>
unsigned int sum(unsigned int a, unsigned int b)
{
    return a+b;
}
main () {
    unsigned int i=0xc0000000;
    unsigned int j=0x70004444;
    printf("%x\n",sum(i,j));
}

Output: 30004444
```
Two’s-Complement Addition

Two’s-complement numbers have a range of

\[-2^{w-1} \leq x, y \leq 2^{w-1} - 1\]

Their sum has the range

\[-2^w \leq x + y \leq 2^w - 2\]

When actual represented result is truncated, it is not modular as unsigned addition

- However, the bit representation for signed and unsigned addition is the same
Two’s-Complement Addition

Since we are dealing with signed numbers, we can have negative overflow or positive overflow.

\[
x + y = \begin{cases} 
  x + y - 2^w, & 2^{w-1} \leq x + y \\
  x + y, & -2^{w-1} \leq x + y < 2^{w-1} \\
  x + y + 2^w, & x + y < -2^{w-1}
\end{cases}
\]
## Example (w=4)

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x + y</th>
<th>$x + \frac{y}{4}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>-8</td>
<td>-5</td>
<td>-13</td>
<td>3</td>
</tr>
<tr>
<td>[1000]</td>
<td>[1011]</td>
<td>[10011]</td>
<td>[0011]</td>
</tr>
<tr>
<td>-8</td>
<td>-8</td>
<td>-16</td>
<td>0</td>
</tr>
<tr>
<td>[1000]</td>
<td>[1000]</td>
<td>[10000]</td>
<td>[0000]</td>
</tr>
<tr>
<td>-8</td>
<td>5</td>
<td>-3</td>
<td>-3</td>
</tr>
<tr>
<td>[1000]</td>
<td>[0101]</td>
<td>[1101]</td>
<td>[1101]</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>[0010]</td>
<td>[0101]</td>
<td>[0111]</td>
<td>[0111]</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>10</td>
<td>-6</td>
</tr>
<tr>
<td>[0101]</td>
<td>[0101]</td>
<td>[1010]</td>
<td>[1010]</td>
</tr>
</tbody>
</table>

### Constraints:
- Case 1: $x + y < -2^{w-1}$
- Case 2: $-2^{w-1} \leq x + y < 2^{w-1}$
- Case 3: $2^{w-1} \leq x + y$
- Case 4: $x + y - 2^w, x + y, x + y + 2^w$
Unsigned Multiplication

For unsigned numbers: $0 \leq x, y \leq 2^{w-1} - 1$

- Thus, $x$ and $y$ are $w$-bit numbers

The product $x \times y$: $0 \leq x \times y \leq (2^{w-1} - 1)^2$

- Thus, product can require $2w$ bits

Only the low $w$ bits are used

- The high order bits may overflow

This makes unsigned multiplication modular

$x \times_w^u y = (x \times y) \mod 2^w$
Two’s-Complement Multiplication

Same problem as unsigned

- The result of multiplying two $w$-bit numbers could be as large as $2^w$ bits

The bit-level representation for two’s-complement and unsigned is identical

- This simplifies the integer multiplier

As before, the interpretation of this value is based on signed vs. unsigned

Maintaining exact results

- Need to keep expanding word size with each product computed
- Must be done in software, if needed
  - e.g., by “arbitrary precision” arithmetic packages
Security issues with multiplication

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);
```

```c
malloc(ele_cnt * 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:

\[
\begin{align*}
\text{ele\_cnt} & = 2^{20} + 1 \\
\text{ele\_size} & = 4096 = 2^{12} \\
\text{Allocation} & = 2^{32} + 4096
\end{align*}
\]

How can I make this function secure?
Multiplication by Powers of Two

What happens if you shift a binary number left one bit?

What if you shift it left $N$ bits?

$\begin{align*}
00001000_2 \ll 2 &= 00100000_2 \\
(8_{10}) \ll 2 &= (32_{10}) \\
11111000_2 \ll 2 &= 11100000_2 \\
(-8_{10}) \ll 2 &= (-32_{10})
\end{align*}$

Examples

$\begin{align*}
u \ll 3 &= u \times 8 \\
(u \ll 5) - (u \ll 3) &= u \times 24
\end{align*}$

- Most machines shift and add faster than multiply
  - Compiler may generate this code automatically
Dividing by Powers of Two (unsigned)

For unsigned numbers, performed via logical right shifts

Quotient of unsigned division by power of 2

- \( u \gg k \) gives \( \lfloor u / 2^k \rfloor \)
- Rounds towards 0

**Operands:**

\[
\begin{array}{c}
\text{u} \\
\text{/} \\
\text{2}^k \\
\hline
\text{u / 2}^k
\end{array}
\]

**Division:**

\[
\begin{array}{c}
\text{u / 2}^k
\end{array}
\]

**Result:**

\[
\lfloor u / 2^k \rfloor
\]

<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>3B 6D</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>x &gt;&gt; 1</td>
<td>7606.5</td>
<td>1D B6</td>
<td>00011101 10110110</td>
</tr>
<tr>
<td>x &gt;&gt; 4</td>
<td>950.8125</td>
<td>03 B6</td>
<td>00000111 10110110</td>
</tr>
<tr>
<td>x &gt;&gt; 8</td>
<td>59.4257813</td>
<td>00 3B</td>
<td>000000000 00111011</td>
</tr>
</tbody>
</table>
Dividing by Powers of Two (signed)

For signed numbers, performed via arithmetic right shifts

Quotient of signed division by power of 2

- \( x >> k \) gives \( \lfloor x / 2^k \rfloor \)
- Rounds away from 0!

Operands:

\[
\begin{array}{c}
\text{x} \\
/ 2^k \\
\text{x} / 2^k
\end{array}
\]

Division:

\[
\begin{array}{c}
\text{x} \\
/ 2^k \\
\text{x} / 2^k
\end{array}
\]

Result:

\[
\text{RoundDown}(x / 2^k)
\]

<table>
<thead>
<tr>
<th>y</th>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>-15213</td>
<td>-15213</td>
<td>C4</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>y &gt;&gt; 1</td>
<td>-7606.5</td>
<td>-7607</td>
<td>E2</td>
<td>11100010 01001001</td>
</tr>
<tr>
<td>y &gt;&gt; 4</td>
<td>-950.8125</td>
<td>-951</td>
<td>FC</td>
<td>11111100 01001001</td>
</tr>
<tr>
<td>y &gt;&gt; 8</td>
<td>-59.4257813</td>
<td>-60</td>
<td>FF</td>
<td>11111111 11000100</td>
</tr>
</tbody>
</table>
Why rounding matters

German parliament (1992)
- 5% law before vote allowed to count for a party
- Rounding of 4.97% to 5% allows Green party vote to count

Vancouver stock exchange (1982)
- Index initialized to 1000, falls to 520 in 22 months
- Updates to index value truncated result instead of rounding
- Value should have been 1098
Exam practice

2.1, 2.3, 2.4  hex binary decimal
2.5, 2.7      endian
2.8, 2.12     bit ops
2.14, 2.15    logical ops
2.16          shifts
2.17, 2.19    2s complement
2.21          implicit casting signed unsigned cmp
2.22, 2.23    2s complement sign xtend
2.25, 2.26    casting security problem
2.28          unsigned additive inverse
2.29          2s complement addition cases
2.33          2s complement additive inverse
2.37          xdr vulnerability fix
2.38, 2.40    shift add to multiply
2.43          rce shift add multiply
2.59, 2.61    bit/logical ops in C
Floating point representation and operations
Fractional Binary Numbers

In Base 10, a decimal point for representing non-integer values

- $125.35$ is $1*10^2 + 2*10^1 + 5*10^0 + 3*10^{-1} + 5*10^{-2}$

In Base 2, a binary point

- $b_n b_{n-1} ... b_1 b_0 . b_{-1} b_{-2} ... b_{-m}$
- $b = \sum 2^i \cdot b_i, \ i = -m \ldots n$
- Example: $101.11_2$ is
  - $1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2}$
  - $4 + 0 + 1 + \frac{1}{2} + \frac{1}{4} = 5\frac{3}{4}$

Accuracy is a problem

- Numbers such as $1/5$ or $1/3$ must be approximated
  - This is true also with decimal
Fractional binary number example

Convert the following binary numbers

$10.111_2$

$1.0111_2$

$1011.101_2$
Floating Point

Integer data type

- 32-bit unsigned integers limited to whole numbers from 0 to just over 4 billion
  - What about large numbers (e.g. national debt, bank bailout bill, Avogadro’s number, Google...the number)?
- 64-bit unsigned integers up to over 9 quintillion
  - What about small numbers and fractions (e.g. 1/2 or π)?

Requires a different interpretation of the bits!

- New data types in C
  - float (32-bit IEEE floating point format)
  - double (64-bit IEEE floating point format)
- 32-bit int and float both represent $2^{32}$ distinct values!
  - Trade-off range and precision
  - e.g. to support large numbers (> $2^{32}$) and fractions, float can not represent every integer between 0 and $2^{32}$!
Floating Point overview

Problem: how can we represent very large or very small numbers with a compact representation?

- Current way with int
  - $5 \times 2^{100}$ as 1010000....0000000000000? (103 bits)
  - Not very compact, but can represent all integers in between

- Another
  - $5 \times 2^{100}$ as 101 01100100 (i.e. $x=101$ and $y=01100100$)? (11 bits)
  - Compact, but does not represent all integers in between

Basis for IEEE Standard 754, “IEEE Floating Point”

- Supported in most modern CPUs via floating-point unit
- Encodes rational numbers in the form $(M \times 2^E)$
  - Large numbers have positive exponent $E$
  - Small numbers have negative exponent $E$
  - Rounding can lead to errors
IEEE Floating-Point

Specifically, IEEE FP represents numbers in the form

\[ V = (-1)^s \times M \times 2^E \]

Three fields

- \( s \) is sign bit: 1 == negative, 0 == positive
- \( M \) is the significand, a fractional number
- \( E \) is the, possibly negative, exponent
IEEE Floating Point Encoding

- $s$ is sign bit
- $\text{exp}$ field encodes $E$
- $\text{frac}$ field encodes $M$

- Sizes
  - Single precision: 8 exp bits, 23 frac bits (32 bits total)
    - C type float
  - Double precision: 11 exp bits, 52 frac bits (64 bits total)
    - C type double
  - Extended precision: 15 exp bits, 63 frac bits
    - Only found in Intel-compatible machines
    - Stored in 80 bits (1 bit wasted)
IEEE Floating-Point

Depending on the exp value, the bits are interpreted differently

- **Normalized (most numbers):** exp is neither all 0’s nor all 1’s
  - E is \((\text{exp} - \text{Bias})\)
    - E is in biased form:
      - Bias=127 for single precision
      - Bias=1023 for double precision
    - Allows for negative exponents
  - M is \(1 + \frac{\text{frac}}{}\)

- **Denormalized (numbers close to 0):** exp is all 0’s
  - E is 1-Bias
    - Not set to –Bias in order to ensure smooth transition from Normalized
  - M is frac
    - Can represent 0 exactly
    - IEEE FP handles +0 and -0 differently

- **Special values:** exp is all 1’s
  - If frac == 0, then we have ±∞, e.g., divide by 0
  - If frac != 0, we have NaN (Not a Number), e.g., sqrt(-1)
Encodings form a continuum

Why two regions?

- Allows 0 to be represented
- Allows for smooth transition near 0
- Encoding allows magnitude comparison to be done via integer unit
Normalized Encoding Example

Using 32-bit float

Value
- float f = 15213.0; /* exp=8 bits, frac=23 bits */
- \[ 15213_{10} = 11101101101101_2 \]
  - \[ = 1.1101101101101_2 \times 2^{13} \text{ (normalized form)} \]

Significand
- \[ M = 1.1101101101101_2 \]
- \[ \text{frac} = 1101101101101000000000000_2 \]

Exponent
- \[ E = 13 \]
- \[ \text{Bias} = 127 \]
- \[ \text{Exp} = 140 = 10001100_2 \]

Floating Point Representation:

<table>
<thead>
<tr>
<th>Hex</th>
<th>4</th>
<th>6</th>
<th>6</th>
<th>D</th>
<th>B</th>
<th>4</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>0100 0110 0110 1101 1011 0100 0000 0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>140:</td>
<td>100 0110 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15213:</td>
<td>1110 1101 1011 01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

http://thefengs.com/wuchang/courses/cs201/class/05/normalized_float.c
Denormalized Encoding Example

Using 32-bit float Value

```c
float f = 7.347e-39;  /* 7.347*10^{-39} */
```

http://thefengs.com/wuchang/courses/cs201/class/05/denormalized_float.c
Distribution of Values

7-bit IEEE-like format

- e = 4 exponent bits
- f = 3 fraction bits
- Bias is 7 (Bias is always set to half the range of exponent – 1)
### 7-bit IEEE FP format (Bias=7)

<table>
<thead>
<tr>
<th>Value</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>frac</th>
<th>E</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0000 000</td>
<td>-6</td>
<td>0000 000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0 0000 001</td>
<td>-6</td>
<td>0000 001</td>
<td>1/8*1/64 = 1/512</td>
<td></td>
</tr>
<tr>
<td>0 0000 010</td>
<td>-6</td>
<td>0000 010</td>
<td>2/8*1/64 = 2/512</td>
<td></td>
</tr>
<tr>
<td>0 0000 110</td>
<td>-6</td>
<td>0000 110</td>
<td>6/8*1/64 = 6/512</td>
<td></td>
</tr>
<tr>
<td>0 0000 111</td>
<td>-6</td>
<td>0000 111</td>
<td>7/8*1/64 = 7/512</td>
<td></td>
</tr>
<tr>
<td>0 0001 000</td>
<td>-6</td>
<td>0001 000</td>
<td>8/8*1/64 = 8/512</td>
<td></td>
</tr>
<tr>
<td>0 0001 001</td>
<td>-6</td>
<td>0001 001</td>
<td>9/8*1/64 = 9/512</td>
<td></td>
</tr>
<tr>
<td>0 0110 110</td>
<td>-1</td>
<td>0110 110</td>
<td>14/8*1/2 = 14/16</td>
<td></td>
</tr>
<tr>
<td>0 0110 111</td>
<td>-1</td>
<td>0110 111</td>
<td>15/8*1/2 = 15/16</td>
<td></td>
</tr>
<tr>
<td>0 0111 000</td>
<td>0</td>
<td>0111 000</td>
<td>8/8*1 = 1</td>
<td></td>
</tr>
<tr>
<td>0 0111 001</td>
<td>0</td>
<td>0111 001</td>
<td>9/8*1 = 9/8</td>
<td></td>
</tr>
<tr>
<td>0 0111 010</td>
<td>0</td>
<td>0111 010</td>
<td>10/8*1 = 10/8</td>
<td></td>
</tr>
<tr>
<td>0 1110 110</td>
<td>7</td>
<td>1110 110</td>
<td>14/8*128 = 224</td>
<td></td>
</tr>
<tr>
<td>0 1110 111</td>
<td>7</td>
<td>1110 111</td>
<td>15/8*128 = 240</td>
<td></td>
</tr>
<tr>
<td>0 1111 000</td>
<td>n/a</td>
<td>1111 000</td>
<td>inf</td>
<td></td>
</tr>
</tbody>
</table>
Distribution of Values

Number distribution gets denser toward zero
6-bit IEEE-like format

- \( e = 3 \) exponent bits
- \( f = 2 \) fraction bits
- Bias is 3

![Diagram of 6-bit IEEE-like format with s, exp, and frac sections.](chart)

-1 -0.5 0 0.5 1

- Denormalized ▲ Normalized ■ Infinity
Practice problem 2.47

Consider a 5-bit IEEE floating point representation
- 1 sign bit, 2 exponent bits, 2 fraction bits, Bias = 1

Fill in the following table

<table>
<thead>
<tr>
<th>Bits</th>
<th>exp</th>
<th>E</th>
<th>frac</th>
<th>M</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 00 00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 00 11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 01 00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 01 10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Practice problem 2.47

Consider a 5-bit IEEE floating point representation
- 1 sign bit, 2 exponent bits, 2 fraction bits, Bias = 1

Fill in the following table

<table>
<thead>
<tr>
<th>Bits</th>
<th>exp</th>
<th>E</th>
<th>frac</th>
<th>M</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 00 00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0 00 11</td>
<td>0</td>
<td>0</td>
<td>¾</td>
<td>¾</td>
<td>¾</td>
</tr>
<tr>
<td>0 01 00</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0 01 10</td>
<td>1</td>
<td>0</td>
<td>½</td>
<td>1 ½</td>
<td>1 ½</td>
</tr>
</tbody>
</table>
Floating Point Operations

FP addition is

- Commutative: $x + y = y + x$
- NOT associative: $(x + y) + z \neq x + (y + z)$
  - $(3.14 + 1e10) - 1e10 = 0.0$, due to rounding
  - $3.14 + (1e10 - 1e10) = 3.14$
- Very important for scientific and compiler programmers

FP multiplication

- Is not associative
- Does not distribute over addition
  - $1e20 \times (1e20 - 1e20) = 0.0$
  - $1e20 \times 1e20 - 1e20 \times 1e20 = \text{NaN}$
- Again, very important for scientific and compiler programmers
Approximations and estimations

Famous floating point errors

- Patriot missile (rounding error from inaccurate representation of 1/10 in time calculations)
  - 28 killed due to failure in intercepting Scud missile (2/25/1991)
- Ariane 5 (floating point cast to integer for efficiency caused overflow trap)
- Microsoft's sqrt estimator...
Floating Point in C

C guarantees two levels
- `float` single precision
- `double` double precision

Casting between data types (not pointer types)
- Casting between int, float, and double results in (sometimes inexact) conversions to the new representation
- `float` to `int`
  - Not defined when beyond range of int
  - Generally saturates to TMin or TMax
- `double` to `int`
  - Same as with float, but, fractional part also truncated (53-bit to 32-bit)
- `int` to `double`
  - Exact conversion
- `int` to `float`
  - Will round
Floating Point Puzzles

int x = ...;
float f = ...;
double d = ...;

Assume neither d nor f is NAN

• x == (int)(float) x
  No: 24 bit significand

• x == (int)(double) x
  Yes: 53 bit significand

• f == (float)(double) f
  Yes: increases precision

• d == (float) d
  No: loses precision

• f == -( -f);
  Yes: Just change sign bit

• 2/3 == 2/3.0
  No: 2/3 == 0

• d < 0.0 ⇒ ((d*2) < 0.0)
  Yes (Note use of \(-\infty\))

• d > f ⇒ -f > -d
  Yes!

• d * d >= 0.0
  Yes! (Note use of \(+\infty\))

• (d+f)-d == f
  No: Not associative
Wait a minute...

Recall
- \( x \equiv (\text{int})(\text{float}) \ x \) No: 24 bit significand

Compiled with gcc –O2, this is true!

Example with \( x = 2147483647 \).

What’s going on?
- See B&O 2.4.6
- x86 uses 80-bit floating point registers
- Optimized code does not return intermediate results into memory
  - Keeps case in 80-bit register
- Non-optimized code returns results into memory
  - 32 bits for intermediate float

http://thefengs.com/wuchang/courses/cs201/class/05/cast_noround.c
Practice problem 2.49

For a floating point format with a k-bit exponent and an n-bit fraction, give a formula for the smallest positive integer that cannot be represented exactly (because it would require an n+1 bit fraction to be exact)
Practice problem 2.49

For a floating point format with a k-bit exponent and an n-bit fraction, give a formula for the smallest positive integer that cannot be represented exactly (because it would require an n+1 bit fraction to be exact)

- What is the smallest n+1 bit integer?
  - \(2^{(n+1)}\)
    - Can this be represented exactly?
    - Yes. \(s=0, \text{exp}=\text{Bias}+n+1, \text{frac}=0\)
    - \(E=n+1, M=1, V=2^{(n+1)}\)

- What is the next largest n+1 bit integer?
  - \(2^{(n+1)}+1\)
    - Can this be represented exactly?
    - No. Need an extra bit in the fraction.
Exam practice

2.45 fractional binary numbers
2.47 5 bit floats
2.48 32 bit floats
2.54 float casts
2.87 half-precision float
2.90 float parsing in C
Extra
Why rounding matters

Well-known errors in currency exchange

- **Direct conversion inaccuracy**

<table>
<thead>
<tr>
<th>BEF</th>
<th>quasi-exact amount in EUR</th>
<th>EUR after rounding</th>
<th>Difference</th>
<th>Percentage of difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0,025302043</td>
<td>0,03</td>
<td>0,004698</td>
<td>16%</td>
</tr>
<tr>
<td>2</td>
<td>0,050604086</td>
<td>0,05</td>
<td>-0,0006</td>
<td>-1%</td>
</tr>
<tr>
<td>3</td>
<td>0,075906129</td>
<td>0,08</td>
<td>0,004094</td>
<td>5%</td>
</tr>
</tbody>
</table>

- **Reconversion errors going to and from currency**

<table>
<thead>
<tr>
<th></th>
<th>From EUR to BEF and back using conversion factor 1 EUR = 38,45 BEF</th>
</tr>
</thead>
<tbody>
<tr>
<td>101 EUR x 38,45 BEF/EUR = 3883,45 BEF; rounded 3883 BEF</td>
<td></td>
</tr>
<tr>
<td>3883 BEF / 38,45 BEF/EURO = 100.99 EUR. 0,01 EUR is missing</td>
<td></td>
</tr>
</tbody>
</table>

- **Totaling errors (compounded rounding errors)**

<table>
<thead>
<tr>
<th></th>
<th>BEF</th>
<th>EURO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Net amount</td>
<td>1000</td>
<td>25,05</td>
</tr>
<tr>
<td>VAT 21%</td>
<td>210</td>
<td>5,26</td>
</tr>
<tr>
<td>Total</td>
<td>1210</td>
<td>30,32 of 30,31</td>
</tr>
</tbody>
</table>