System Programming

05. Floating Point (ch 2.4)

2019. Fall

Instructor: Joonho Kwon

jhkwon@pusan.ac.kr

Data Science Lab @ PNU

datalab

data science laboratory

These slides are based on your text book on the slides prepared by Dan Grossman, Ruth Anderson.

Roadmap

- Memory & data
- Integers & floats
- x86 assembly
- Procedures & stacks
- Executables
- Arrays & structs
- Memory & caches
- Processes
- Virtual memory
- Memory allocation
- Java vs. C

C:

```c
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```java
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

Assembly language:

```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:

Windows 8 Mac

Computer system:


Number Representation Revisited

- What can we represent so far?
  - Signed and Unsigned Integers
  - Characters (ASCII)
  - Addresses
- How do we encode the following:
  - Real numbers (*e.g.* 3.14159)
  - Very large numbers (*e.g.* 6.02×10^{23})
  - Very small numbers (*e.g.* 6.626×10^{-34})
  - Special numbers (*e.g.* ∞, NaN)

Floating Point

Floating point topics

- **Fractional binary numbers**
- IEEE floating-point standard
- Floating-point operations and roundir
- Floating-point in C

- There are many more details that we won't cover
  - It's a 58-page standard…

Floating Point Summary

- As with integers, floats suffer from the fixed number of bits available to represent them
  - Can get overflow/underflow, just like ints
  - Some “simple fractions” have no exact representation (e.g., 0.2)
  - Can also lose precision, unlike ints
    - “Every operation gets a slightly wrong result”
- Floating point arithmetic not associative or distributive
  - Mathematically equivalent ways of writing an expression may compute different results

- Never test floating point values for equality!
- Careful when converting between ints and floats!

Representation of Fractions

- "Binary Point," like decimal point, signifies boundary between integer and fractional parts:

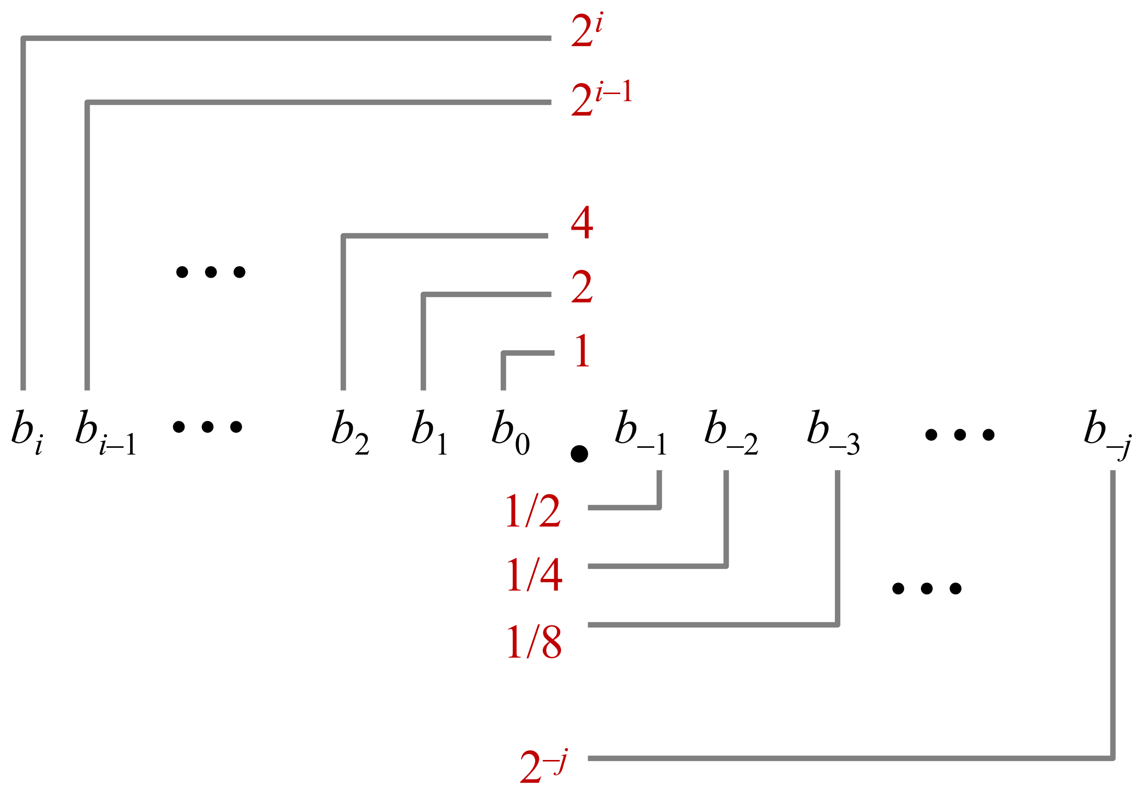
Example 6-bit representation: **xx.yyyy**

2^{1} 2^{0} 2^{-1} 2^{-2} 2^{-3} 2^{-4}

- <u>Example</u>: $10.1010_2 = 1 \times 2^1 + 1 \times 2^{-1} + 1 \times 2^{-3} = 2.625_{10}$

- Binary point numbers that match the 6-bit format above range from 0 (00.0000_2) to 3.9375 (11.1111_2)

Fractional Binary Numbers



- Representation
  - Bits to right of "binary point" represent fractional powers of 2
  - Represents rational number:

$$\sum_{k=-j}^{i} b_k \cdot 2^k$$

Fractional Binary Numbers

- Value Representation
  - 5 and 3/4 101.11_2
  - 2 and 7/8 10.111_2
  - 47/64 0.101111_2
- Observations
  - Shift left = multiply by power of 2
  - Shift right = divide by power of 2
  - Numbers of the form $0.111111\ldots_2$ are just below 1.0
    - $1/2 + 1/4 + 1/8 + \ldots + 1/2^i + \ldots \rightarrow 1.0$
    - Use notation $1.0 - \varepsilon$

Limits of Representation

- Limitations:
  - Even given an arbitrary number of bits, can only **exactly** represent numbers of the form $x * 2^y$ (y can be negative)
  - Other rational numbers have repeating bit representations

Value: **Binary Representation:**

- $1/3 = 0.333333\ldots_{10} =$ $0.01010101[01]\ldots_2$
- $1/5 =$ $0.001100110011[0011]\ldots_2$
- $1/10 =$ $0.0001100110011[0011]\ldots_2$

Fixed Point Representation

- Implied binary point. Two example schemes:

  #1: the binary point is between bits 2 and 3

  $b_7\ b_6\ b_5\ b_4\ b_3$ [.] $b_2\ b_1\ b_0$

  #2: the binary point is between bits 4 and 5

  $b_7\ b_6\ b_5$ [.] $b_4\ b_3\ b_2\ b_1\ b_0$

- Wherever we put the binary point, with fixed point representations there is a trade off between the amount of range and precision we have
- Fixed point = fixed *range* and fixed *precision*
  - range: difference between largest and smallest numbers possible
  - precision: smallest possible difference between any two numbers
- Hard to pick how much you need of each!

Floating Point Representation

- Analogous to scientific notation
  - In Decimal:
    - Not 12000000, but 1.2×10^7 In C: 1.2e7
    - Not 0.0000012, but 1.2×10^{-6} In C: 1.2e-6
  - In Binary:
    - Not 11000.000, but 1.1×2^4
    - Not 0.000101, but 1.01×2^{-4}

- We have to divvy up the bits we have (e.g., 32) among:
  - the sign (1 bit)
  - the significand
  - the exponent

Scientific Notation Translation

- Convert from scientific notation to binary point
  - Perform the multiplication by shifting the decimal until the exponent disappears
    - <u>Example</u>: $1.011_2 \times 2^4 = 10110_2 = 22_{10}$
    - <u>Example</u>: $1.011_2 \times 2^{-2} = 0.01011_2 = 0.34375_{10}$
- Convert from binary point to *normalized* scientific notation
  - Distribute out exponents until binary point is to the right of a single digit
    - <u>Example</u>: $1101.001_2 = 1.101001_2 \times 2^3$
- **Practice:** Convert 11.375_{10} to binary scientific notation
- **Practice:** Convert 1/5 to binary

Floating point topics

- Fractional binary numbers
- **IEEE floating-point standard**
- Floating-point operations and rounding
- Floating-point in C

- There are many more details that we won't cover
  - It's a 58-page standard…

IEEE Floating Point

- IEEE 754
  - Established in 1985 as uniform standard for floating point arithmetic
  - Main idea: make numerically sensitive programs portable
  - Specifies two things: representation and result of floating operations
  - Now supported by all major CPUs

- Driven by numerical concerns
  - **Scientists**/numerical analysts want them to be as **real** as possible
  - **Engineers** want them to be **easy to implement** and **fast**
  - In the end:
    - Scientists mostly won out
    - Nice standards for rounding, overflow, underflow, but...
    - Hard to make fast in hardware
    - **Float operations can be an order of magnitude slower than integer ops**

Floating Point Representation

- Numerical form:

$$V_{10} = (-1)^{s} * M * 2^{E}$$

  - Sign bit ***s*** determines whether number is negative or positive
  - Significand (mantissa) ***M*** normally a fractional value in range [1.0,2.0)
  - Exponent ***E*** weights value by a (possibly negative) power of two

Floating Point Representation

- Numerical form:

$$V_{10} = (-1)^{s} * M * 2^{E}$$

  - Sign bit ***s*** determines whether number is negative or positive
  - Significand (mantissa) ***M*** normally a fractional value in range [1.0,2.0)
  - Exponent ***E*** weights value by a (possibly negative) power of two
- Representation in memory:
  - MSB **`s`** is sign bit ***s***
  - **`exp`** field encodes ***E*** (but is *not equal* to E)
  - **`frac`** field encodes ***M*** (but is *not equal* to M)

`s`	`exp`	`frac`

Precisions

Finite representation means not all values can be represented exactly. Some will be approximated.

- Single precision: 32 bits

s	exp	frac
1	8-bits	23-bits

- Double precision: 64 bits

s	exp	frac
1	11-bits	52-bits

- Extended precision: 80 bits (Intel only)

s	exp	frac
1	15-bits	63 or 64-bits

Normalization and Special Values

$v = (-1)^s \boldsymbol{M}\, 2^E$

s	exp	frac

- "Normalized" = **M** has the form 1.xxxxx
  - As in scientific notation, but in binary
  - 0.011×2^5 and 1.1×2^3 represent the same number, but the latter makes better use of the available bits
  - Since we know the mantissa starts with a 1, we don't bother to store it

- **How do we represent 0.0? Or special or undefined values like 1.0/0.0?**

Normalized Values

$v = (-1)^s \, M \, 2^E$

s	exp	frac

- "Normalized" = **M** has the form 1.xxxxx
  - As in scientific notation, but in binary
  - 0.011×2^5 and 1.1×2^3 represent the same number, but the latter makes better use of the available bits
  - Since we know the mantissa starts with a 1, we don't bother to store it
- Special values:

zero: **s** == 0 **exp** == 00…0 **frac** == 00…0

+ ∞, - ∞ : **exp** == 11…1 **frac** == 00…0

$1.0/0.0 = -1.0/-0.0 = +\infty$, $1.0/-0.0 = -1.0/0.0 = -\infty$

NaN ("Not a Number"): **exp** == 11…1 **frac** != 00…0

Results from operations with undefined result: sqrt(-1), $\infty - \infty$, $\infty * 0$, etc.

Note: **exp=11…1 and exp=00…0 are reserved**, limiting exp range…

Normalized Values

$v = (-1)^s \, \boldsymbol{M} \, 2^{\boldsymbol{E}}$

s	exp	frac

- When: exp ≠ 000…0 and exp ≠ 111…1
- Exponent coded as a ***biased*** value: ***E* = *Exp* – *Bias***
  - *Exp*: unsigned value of exp field
  - *Bias* = 2^{k-1} - 1, where *k* is number of exponent bits
    - Single precision: 127 (Exp: 1…254, E: -126…127)
    - Double precision: 1023 (Exp: 1…2046, E: -1022…1023)

- Significand coded with implied leading 1: ***M*** = $1.xxx…x_2$
  - xxx…x: bits of frac field
  - Minimum when frac=000…0 (M = 1.0)
  - Maximum when frac=111…1 (M = 2.0 – ε)
  - Get extra leading bit for "free"

Normalized Encoding Example

$v = (-1)^s\, M\, 2^E$
$E = Exp - Bias$

- Value: `float F = 15213.0;`
  - $15213_{10} = 11101101101101_2$
    $= 1.1101101101101_2 \times 2^{13}$ (normalized form)

- Significand
  $M = 1.\underline{1101101101101}_2$
  frac= $\underline{1101101101101}0000000000_2$

- Exponent
  $E = 13$
  $Bias = 127$
  $Exp = 140 = 10001100_2$

- Result:

s	exp	frac
0	10001100	11011011011010000000000

Question

- What is the correct value encoded by the following floating point number?
  - 0b 0 10000000 11000000000000000000000

A. + 0.75

B. + 1.5

C. + 2.75

D. + 3.5

Denormalized Values

$v = (-1)^s \, M \, 2^E$
$E = 1 - Bias$

- Condition: exp = 000…0

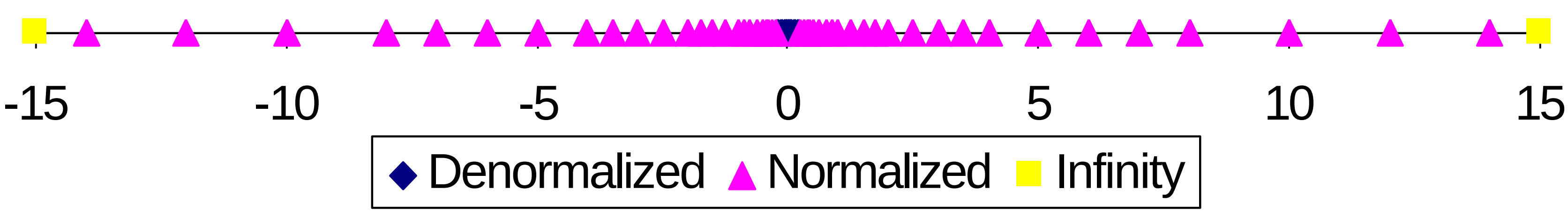
- Exponent value: $E = 1 - \text{Bias}$ (instead of $E = 0 - Bias$)
- Significand coded with implied leading 0: $M = 0.\text{xxx}\ldots\text{x}_2$
  - `xxx…x`: bits of `frac`
- Cases
  - `exp` = 000…0, `frac` = 000…0
    - Represents zero value
    - Note distinct values: +0 and –0 (why?)
  - `exp` = 000…0, `frac` ≠ 000…0
    - Numbers closest to 0.0
    - Equispaced

Distribution of Values

- 6-bit IEEE-like format
  - e = 3 exponent bits
  - f = 2 fraction bits
  - Bias is $2^{3-1}-1 = 3$

s	exp	frac
1	3	2

- Notice how the distribution gets denser toward zero.



Floating point topics

- Fractional binary numbers
- IEEE floating-point standard
- **Floating-point operations** and rounding
- Floating-point in C

- There are many more details that we won't cover
  - It's a 58-page standard…

Floating Point Operations

- Unlike the representation for integers, the representation for floating-point numbers is <u>not exact</u>

Floating Point Operations: Basic Idea

$v = (-1)^s \, M \, 2^E$

s	exp	frac

- $\mathtt{x} +_{\mathtt{f}} \mathtt{y} = \mathtt{Round}(\mathtt{x} + \mathtt{y})$
- $\mathtt{x} \times_{\mathtt{f}} \mathtt{y} = \mathtt{Round}(\mathtt{x} \times \mathtt{y})$

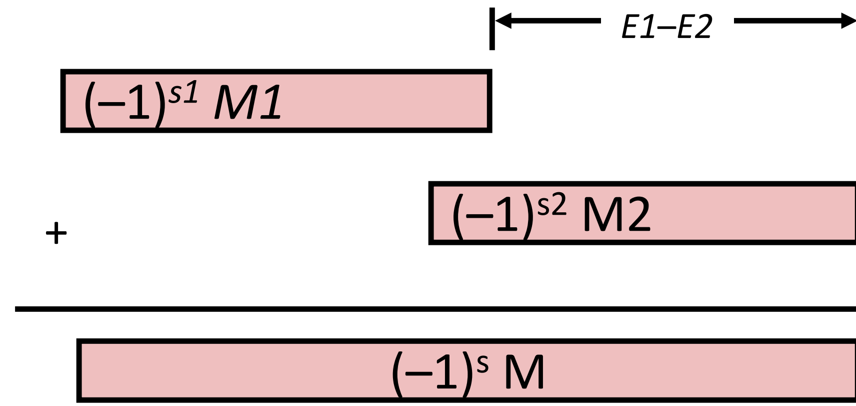
- Basic idea
  - First **compute exact result**
  - Then, **round** the result to make it fit into desired precision:
    - Possibly overflow if exponent too large
    - Possibly drop least-significant bits of significand to fit into frac

Floating Point Addition

Line up the binary points

$(-1)^{s1}\ M1\ \ 2^{E1}\ \ +\ \ (-1)^{s2}\ M2\ \ 2^{E2}$

Assume $E1 > E2$



- Exact Result: $(-1)^s\ M\ \ 2^E$
  - Sign *s*, significand *M*:
    - Result of signed align & add
  - Exponent *E*: *E1*
- Fixing
  - If $M \geq 2$, shift *M* right, increment *E*
  - if $M < 1$, shift *M* left *k* positions, decrement *E* by *k*
  - Overflow if *E* out of range
  - Round *M* to fit `frac` precision

Floating Point Multiplication

$(-1)^{s1}\, M1\;\; 2^{E1} \;\; * \;\; (-1)^{s2}\, M2\;\; 2^{E2}$

- Exact Result: $(-1)^{s}\, M\;\; 2^{E}$
  - Sign *s*: $s1$ ^ $s2$
  - Significand *M*: $M1 * M2$
  - Exponent *E*: $E1 + E2$

- Fixing
  - If $M \geq 2$, shift *M* right, increment *E*
  - If *E* out of range, overflow
  - Round *M* to fit `frac` precision

Rounding

- Rounding Modes (illustrate with $ rounding)

	\$1.40	\$1.60	\$1.50	\$2.50	–\$1.50
Towards zero	\$1	\$1	\$1	\$2	–\$1
Round down (−∞)	\$1	\$1	\$1	\$2	–\$2
Round up (+∞)	\$2	\$2	\$2	\$3	–\$1
Nearest Even (default)	\$1	\$2	\$2	\$2	–\$2

- Round-to-even avoids statistical bias in repeated rounding.
  - Rounds up about half the time, down about half the time.
  - Default rounding mode for IEEE floating-point

Mathematical Properties of FP Operations

- Exponent overflow yields +∞ or -∞
- Floats with value +∞, -∞, and NaN can be used in operations
  - Result usually still +∞, -∞, or NaN; but not always intuitive
- Floating point operations do not work like real math, due to rounding!!
  - Not associative: `(3.14+1e100)-1e100 != 3.14+(1e100-1e100)`

    0 **3.14**
  - Not distributive: `100*(0.1+0.2) != 100*0.1+100*0.2`

    30.000000000000003553 **30**
  - Not cumulative
    - Repeatedly adding a very small number to a large one may do nothing

Floating Point Topics

- Fractional binary numbers
- IEEE floating-point standard
- Floating-point operations and rounding
- **Floating-point in C**

- There are many more details that we won't cover
  - It's a 58-page standard…

Floating Point in C

!!!

- C offers two (well, 3) levels of precision

`float`	`1.0f`	single precision (32-bit)
`double`	`1.0`	double precision (64-bit)
`long double`	`1.0L`	*("double double" or quadruple)* precision (64-128 bits)

- `#include <math.h>` to get `INFINITY` and `NAN` constants
- Equality (==) comparisons between floating point numbers are tricky, and often return unexpected results, so just avoid them!
- Instead use abs(f1 – f2) < 2^{-20} or some other threshold

Floating Point Conversions in C

!!!

- Casting between `int`, `float`, and `double` **changes** the bit representation
  - **`int` → `float`**
    - May be rounded (not enough bits in mantissa: 23)
    - Overflow impossible
  - **`int` or `float` → `double`**
    - Exact conversion (all 32-bit `int`s representable; 52-bit frac)
  - **`long` → `double`**
    - Depends on word size (32-bit is exact, 64-bit may be rounded)
  - **`double` or `float` → `int`**
    - Truncates fractional part (rounded toward zero)
    - “Not defined” when out of range or NaN: generally sets to `Tmin` (even if the value is a very big positive)

Number Representation Really Matters

- **1991:** Patriot missile targeting error
  - clock skew due to conversion from integer to floating point
- **1996:** Ariane 5 rocket exploded ($1 billion)
  - overflow converting 64-bit floating point to 16-bit integer
- **2000:** Y2K problem
  - limited (decimal) representation: overflow, wrap-around
- **2038:** Unix epoch rollover
  - Unix epoch = seconds since 12am, January 1, 1970
  - signed 32-bit integer representation rolls over to TMin in 2038
- **Other related bugs:**
  - 1982: Vancouver Stock Exchange 10% error in less than 2 years
  - 1994: Intel Pentium FDIV (floating point division) HW bug ($475 million)
  - 1997: USS Yorktown "smart" warship stranded: divide by zero
  - 1998: Mars Climate Orbiter crashed: unit mismatch ($193 million)

Floating Point and the Programmer

```
include <stdio.h>

int main(int argc, char* argv[]) {
  int a = 33554435;
  printf("a = %d\n(float) a = %f
          \n\n", a, (float) a);

  float f1 = 1.0;
  float f2 = 0.0;
  int i;
  for (i = 0; i < 10; i++)
      f2 += 1.0/10.0;

  printf("0x%08x  0x%08x\n", *(int*)&f1, *(int*)&f2);
  printf("f1 = %10.9f\n", f1);
  printf("f2 = %10.9f\n\n", f2);

  f1 = 1E30;
  f2 = 1E-30;
  float f3 = f1 + f2;
  printf("f1 == f3? %s\n", f1 == f3 ? "yes" : "no" );
  return 0;
}
```

```
$ ./a.out
a = 33554435
(float) a = 33554436.000000
0x3f800000  0x3f800001
f1 = 1.000000000
f2 = 1.000000119

f1 == f3? yes
```

Floating Point Summary

- As with integers, floats suffer from the fixed number of bits available to represent them
  - Can get overflow/underflow, just like ints
  - Some “simple fractions” have no exact representation (e.g., 0.2)
  - Can also lose precision, unlike ints
    - “Every operation gets a slightly wrong result”
- Floating point arithmetic not associative or distributive
  - Mathematically equivalent ways of writing an expression may compute different results

- Never test floating point values for equality!
- Careful when converting between ints and floats!

Floating Point Puzzles

s	exp	frac
1 bit	8 bits	23 bits

s	exp	frac
1 bit	11 bits	52 bits

- For each of the following C expressions, either:
  - Argue that it is true for all argument values
  - Explain why not true

```
int x = …;
float f = …;
double d = …;
double d2 = …;
```

Assume neither **d** nor **f** is NaN

- `x == (int)(float) x`
- `x == (int)(double) x`
- `f == (float)(double) f`
- `d == (double)(float) d`
- `f == -(-f);`
- `2/3 == 2/3.0`
- `(d+d2)-d == d2`

Q&A