Announcement

- Locations for lab hours are finalized (check course webpage).
- Lab 1 due this Friday (with the 2-day extension, that will be Sunday night).

Floating Points (Cont)

B&O Readings: 2.4
CSE 361: Introduction to Systems Software

Instructor:
I-Ting Angelina Lee

Today: Floating Point

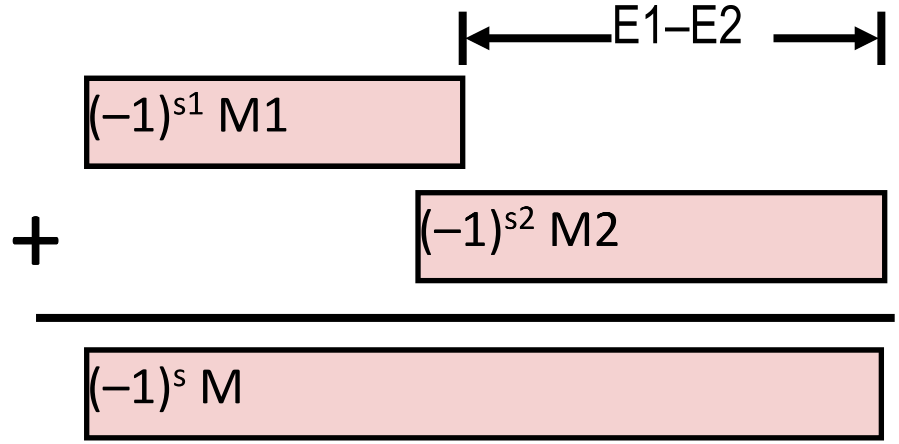
- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Floating point operations and rounding
- Floating point in C

Floating Point Multiplication

- $(-1)^{s1}\ M1\ \ 2^{E1}\ \ \ \mathrm{x}\ \ \ (-1)^{s2}\ M2\ \ 2^{E2}$
- **Exact Result:** $(-1)^{s}\ M\ \ 2^{E}$
  - Sign s: s1 ^ s2
  - Mantissa M: M1 x M2
  - Exponent E: E1 + E2
- **Fixing**
  - If M ≥ 2, shift M right, increment E
  - If E out of range, overflow
  - Round M to fit `frac` precision
- **Implementation**
  - Biggest chore is multiplying the Mantissas

Floating Point Addition

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



- **Exact Result:** $(-1)^{s}\ M\ \ 2^{E}$
  - Sign s, mantissa M:
    - Result of signed align & add
  - Exponent E: E1

- **Fixing**
  - If M ≥ 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 Operations: Basic Idea

$V = (-1)^{s} \cdot M \cdot 2^{E}$

s	exp	frac
	8	23

- **`x +f y = Round(x + y)`**
  - E could be very different
  - the binary point needs to line up
- **`x ×f y = Round(x × y)`**
  - need to ensure that the resulting exponent is still in range

- **Basic idea**
  - First compute exact result
  - Make it fit into desired precision
    - Possibly overflow if exponent too large
    - Possibly round to fit into `frac`

IEEE Rounding Modes

- **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 nearest Even:**
  - When more than halfway, round up; when less than halfway, round down.
  - When exactly halfway between two possible values, round it so that least significant digit is even
  - The default rounding mode.
  - Why? So that we don't introduce statistical bias.
  - All others are statistically biased

Rounding Binary Numbers

- **When exactly halfway between two possible values**
  - Round so that least significant digit is even
- **Binary Fractional Numbers**
  - "Even" when least significant bit is `0`
  - "Half way" when bits to right of rounding position = `100`…$_2$
- **Examples**
  - Round to nearest 1/4 (2 bits right of binary point)

Value	Binary	Rounded	Action	Rounded Value
2 3/32	10.00011_2	10.00_2	(<1/2—down)	2
2 3/16	10.00110_2	10.01_2	(>1/2—up)	2 1/4
2 7/8	10.11100_2	11.00_2	(1/2—up)	3
2 5/8	10.10100_2	10.10_2	(1/2—down)	2 1/2

Mathematical Properties of FP Add

- **Commutative?** Yes
- **Associative?** No
  - Overflow and inexactness of rounding
  - `(3.14+1e10)-1e10 = 0, 3.14+(1e10-1e10) = 3.14`
- **0 is additive identity?** Yes
- **Every element has additive inverse?** Almost
  - Yes, except for infinities & NaNs
- **Monotonicity** Almost
  - $a \geq b \Rightarrow a+c \geq b+c$?
    - Except for infinities & NaNs

Mathematical Properties of FP Mult

- **Multiplication Commutative?** Yes
- **Multiplication is Associative?** No
  - Possibility of overflow, inexactness of rounding
  - Ex: `(1e20*1e20)*1e-20= inf, 1e20*(1e20*1e-20)= 1e20`
- **1 is multiplicative identity?** Yes
- **Multiplication distributes over addition?** No
  - Possibility of overflow, inexactness of rounding
  - `1e20*(1e20-1e20)= 0.0, 1e20*1e20 – 1e20*1e20 = NaN`
- **Monotonicity** Almost
  - $a \geq b$ & $c \geq 0 \Rightarrow a * c \geq b * c$?
    - Except for infinities & NaNs

Recap: What We Learned Thus Far

- **Due to inexactness of rounding and possibility of overflow, floating point operations are NOT associative. Although they are commutative and generally maintains monotonicity (except when you have +/- infinity / NaN involved).**

- **Watch out:**
  - When reordering the order of floating point operations, you may not get the same result!
  - Compiler never does this!

Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Rounding, addition, multiplication
- **Floating point in C**

Float and Double in C

- **Designed to represent reals and fractional numbers**
- **Still fixed width, like the integer data types:**
  - float: 4 bytes
  - double: 8 bytes

What Does This Code Print?

```
float f = 0.3;
printf("%.20f\n", f);
printf("%.20f\n", 0.1+0.2);
```

A) 0.3 for both
B) Not exactly 0.3, but two printouts show the same values
C) Not exactly 0.3, and two printouts differ

```
0.30000001192092895508
0.30000000000000004441
```

Answer: C)

By default, real constants have type double, which has better precision (unless suffixed with `f` or `F`).

Floating Point in C

- **C Guarantees Two Levels**
  - **`float`** single precision
  - **`double`** double precision
- **Conversions/Casting**
  - Casting between **`int`**, **`float`**, and **`double`** changes bit representation
  - **`double`/`float`** → **`int`**
    - Truncates fractional part
    - Like rounding toward zero
    - Not defined when out of range or NaN: Generally sets to TMin
  - **`int`** → **`double`**
    - Exact conversion, as long as `int` has ≤ 53 bit word size
  - **`int`** → **`float`**
    - Will round according to rounding mode

Floating Point Puzzles

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

```
int x = …;
long l = …;
float f = …;
double d = …;
```

Assume neither `d` nor `f` is NaN

```
A. x == (int)(float) x
B. x == (int)(double) x
C. f == (float)(double) f
D. l == (long)(double) l
E. d == (double)(float) d
F. f == -(-f);
G. 2/3 == 2/3.0
H. d < 0.0 => ((d*2) < 0.0)
I. d > f => -f > -d
J. d * d >= 0.0
K. (f+d)-f == d
```

Floating Point Puzzles Answers

A. No, float has 23 frac bits, and x has 32 bits
B. Yes, double has greater precision and range (52 frac bits)
C. Yes, since double has a wider range and prec.
D. No, long has 64 bits and double has only 52 frac bits
E. No, can lose precision / overflow to infinity
F. Yes, it simply negates the sign bit
G. No, result of 2/3 would be int so you get 0, 2/3.0 will be a double
H. Yes, even if it overflows, it will overflow to –inf
I. Yes, monotonicity, but also, note that we won't lost anything; unlike int, the negative and positive range representable by float or double are the same
J. Yes, though it may overflow to +INF, but that's still > 0
K. No, floating ops are not associative; if f is a really large number and d is really small, f+d will be rounded to about f, and you may get 0 on the left.

Machine-Level Programming I: Basics

B&O Readings: 2.1, 3.1-3.5
CSE 361: Introduction to Systems Software

Instructor:
I-Ting Angelina Lee

Byte-Oriented Memory Organization

- **Programs refer to data by address**
  - Conceptually, envision it as a very large array of bytes
    - In reality, it's not, but can think of it that way
  - An address is like an index into that array
    - and, a ***pointer*** variable stores an address

- **Note: system provides private address spaces to each "process"**
  - Think of a process as a program being executed
  - So, a program can clobber its own data, but not that of others

Machine Words

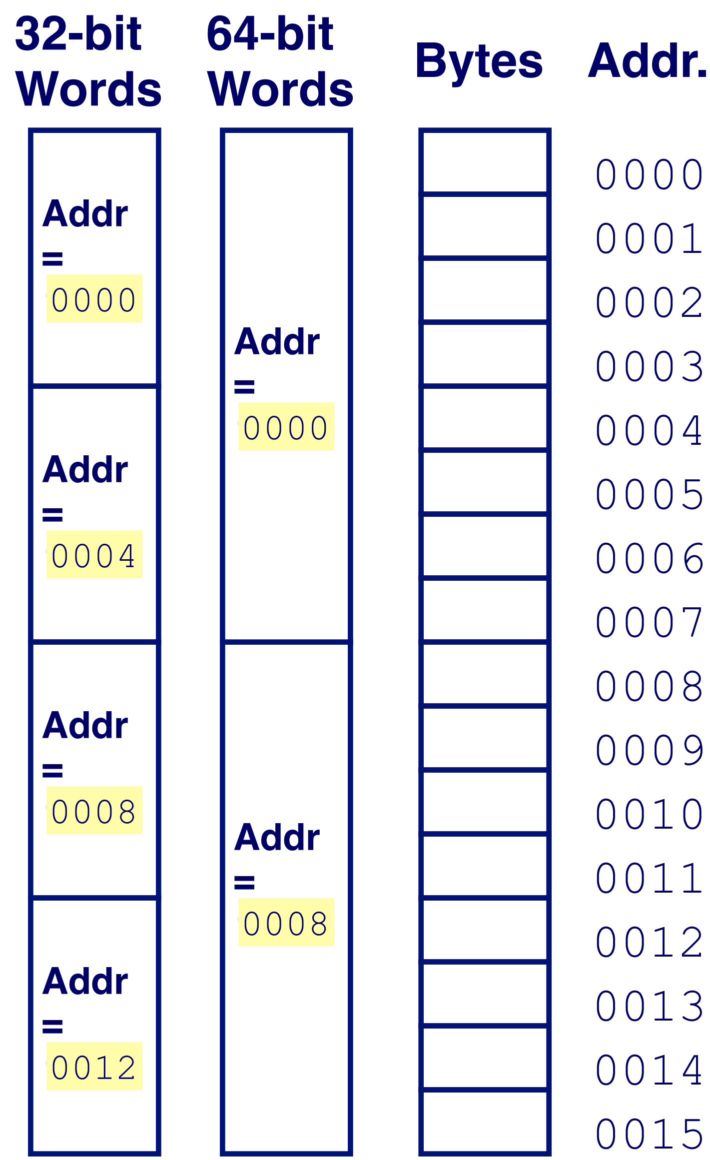
- **Any given computer has a "Word Size"**
  - Nominal size of integer-valued data
  - Or, the size of an address
  - Until recently, most machines used 32 bits (4 bytes) as word size
    - Limits addresses to 4GB (2^{32} bytes)
  - These days, most machines have 64-bit word size
    - Potentially, could have 18 PB (petabytes) of addressable memory
    - That's 18.4 X 10^{15}
  - Machines support multiple data formats
    - Fractions or multiples of word size
    - Always integral number of bytes

Word-Oriented Memory Organization

- **Addresses Specify Byte Locations**
  - Address of first byte in word
  - Addresses of successive words differ by 4 (32-bit word) or 8 (64-bit word)

- **Pointers in C:**

```
int x = 100;
int *y = &x; //y store addr of x
*y = *y + 1; //x is now 101
```



Byte Ordering

Question: how are the bytes within a multi-byte word ordered in memory?

- Example: Variable x has 4-byte value of 0x01234567
- Address given by &x is 0x100

Big Endian

A)

		0x100	0x101	0x102	0x103		
		01	23	45	67		

Least significant byte has the highest address

Ex: Sun, MacPPC, Internet

Little Endian

B)

		0x100	0x101	0x102	0x103		
		67	45	23	01		

Least significant byte has the lowest address

Ex: x86, ARM running Android, iOS, and Windows

C) Both are valid

Answer: Both