CS 61C: Great Ideas in Computer Architecture
Floating Point Arithmetic

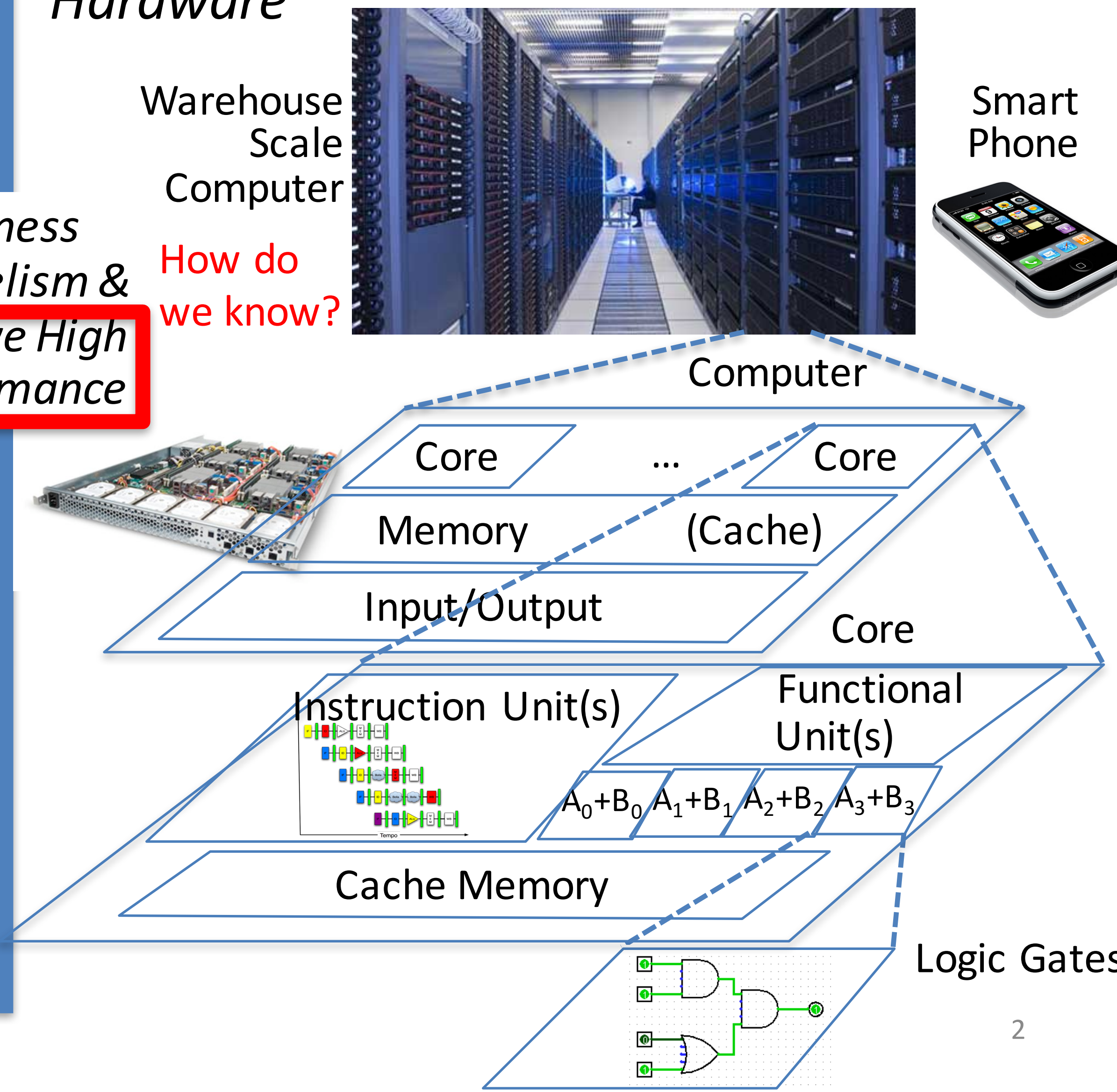
Instructors:
Vladimir Stojanovic & Nicholas Weaver
http://inst.eecs.berkeley.edu/~cs61c/

New-School Machine Structures (It's a bit more complicated!)

Software | *Hardware*

- Parallel Requests
  - Assigned to computer
  - e.g., Search "Katz"
- Parallel Threads
  - Assigned to core
  - e.g., Lookup, Ads
- Parallel Instructions
  - >1 instruction @ one time
  - e.g., 5 pipelined instructions
- Parallel Data
  - >1 data item @ one time
  - e.g., Add of 4 pairs of words
- Hardware descriptions
  - All gates @ one time
- Programming Languages

Harness Parallelism &
Achieve High Performance



Review of Numbers

- Computers are made to deal with numbers
- What can we represent in N bits?
  - 2^N things, and no more! They could be…
  - Unsigned integers:

    0 to $2^N - 1$

    (for N=32, $2^N - 1 = 4{,}294{,}967{,}295$)
  - Signed Integers (Two's Complement)

    $-2^{(N-1)}$ to $2^{(N-1)} - 1$

    (for N=32, $2^{(N-1)} = 2{,}147{,}483{,}648$)

What about other numbers?

1. Very large numbers? (seconds/millennium)
   ⇒ $31,556,926,000_{ten}$ ($3.1556926_{10} \times 10^{10}$)
2. Very small numbers? (Bohr radius)
   ⇒ 0.0000000000529177_{ten} ($5.29177_{10} \times 10^{-11}$)
3. Numbers with <u>both</u> integer & fractional parts?
   ⇒ 1.5

First consider #3.

…our solution will also help with #1 and #2.

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}

$$\mathbf{10.1010_{two} = 1x2^1 + 1x2^{-1} + 1x2^{-3} = 2.625_{ten}}$$

If we assume "fixed binary point", range of 6-bit representations with this format:

0 to 3.9375 (almost 4)

Fractional Powers of 2

i	2^{-i}		
0	1.0	1	
1	0.5		1/2
2	0.25	1/4	
3	0.125	1/8	
4	0.0625	1/16	
5	0.03125	1/32	
6	0.015625		
7	0.0078125		
8	0.00390625		
9	0.001953125		
10	0.0009765625		
11	0.00048828125		
12	0.000244140625		
13	0.0001220703125		
14	0.00006103515625		
15	0.000030517578125		

Representation of Fractions with Fixed Pt.

What about addition and multiplication?

Addition is straightforward:

```
  01.100   1.5ten
+ 00.100   0.5ten
  ------
  10.000   2.0ten
```

Multiplication a bit more complex:

```
    01.100   1.5ten
    00.100   0.5ten
    ------
    00 000
   000 00
  0110 0
 00000
00000
----------
0000110000
```

Where's the answer, `0.11`? (need to remember where point is)

Representation of Fractions

So far, in our examples we used a "fixed" binary point. What we really want is to "float" the binary point. Why?

Floating binary point most effective use of our limited bits (and thus more accuracy in our number representation):

example: put 0.1640625_{ten} into binary. Represent with 5-bits choosing where to put the binary point.

… 000000.001010100000…

Store these bits and keep track of the binary point 2 places to the left of the MSB

Any other solution would lose accuracy!

With floating-point rep., each numeral carries an exponent field recording the whereabouts of its binary point.

The binary point can be outside the stored bits, so very large and small numbers can be represented.

Scientific Notation (in Decimal)

mantissa → $6.02_{ten} \times 10^{23}$ ← exponent

decimal point ↑ radix (base)

- Normalized form: no leading 0s
  (exactly one digit to left of decimal point)
- Alternatives to representing 1/1,000,000,000
  - Normalized: 1.0×10^{-9}
  - Not normalized: $0.1 \times 10^{-8}, 10.0 \times 10^{-10}$

Scientific Notation (in Binary)

mantissa → $1.01_{two} \times 2^{-1}$ ← exponent

↑ "binary point"   ↖ radix (base)

- Computer arithmetic that supports it called <u>floating point</u>, because it represents numbers where the binary point is not fixed, as it is for integers
  - Declare such variable in C as `float`
    - `double` for double precision.

Floating-Point Representation (1/2)

- Normal format: $+1.xxx\ldots x_{two} * 2^{yyy\ldots y_{two}}$
- Multiple of Word Size (32 bits)

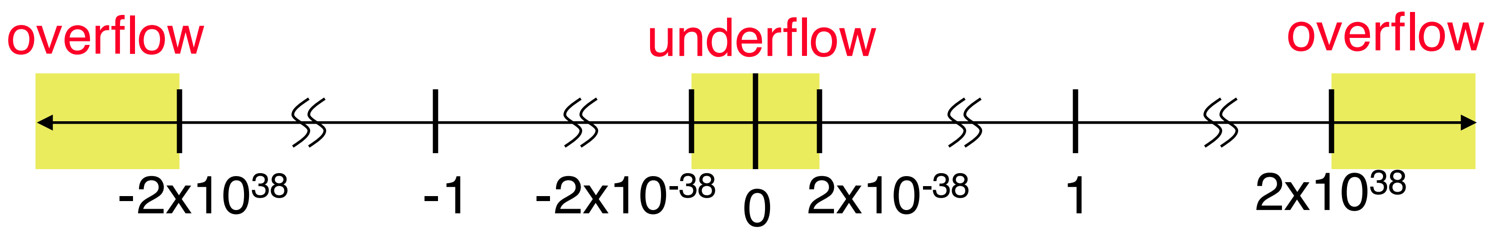
31	30 ... 23	22 ... 0
S	Exponent	Significand
1 bit	8 bits	23 bits

- S represents Sign
  Exponent represents y's
  Significand represents x's

- Represent numbers as small as $2.0_{ten} \times 10^{-38}$ to as large as $2.0_{ten} \times 10^{38}$

Floating-Point Representation (2/2)

- **What if result too large?**
  - ($> 2.0 \times 10^{38}$, $< -2.0 \times 10^{38}$)
  - **Overflow!** ⇒ Exponent larger than represented in 8-bit Exponent field
- **What if result too small?**
  - (>0 & $< 2.0 \times 10^{-38}$, <0 & $> -2.0 \times 10^{-38}$)
  - **Underflow!** ⇒ Negative exponent larger than represented in 8-bit Exponent field



- **What would help reduce chances of overflow and/or underflow?**

IEEE 754 Floating-Point Standard (1/3)

Single Precision (Double Precision similar):

31	30 ... 23	22 ... 0
S	Exponent	Significand
1 bit	8 bits	23 bits

- Sign bit: 1 means negative
  0 means positive
- Significand in *sign-magnitude* format (not 2's complement)
  - To pack more bits, leading 1 implicit for normalized numbers
  - 1 + 23 bits single, 1 + 52 bits double
  - always true: $0 < \text{Significand} < 1$ (for normalized numbers)
- Note: 0 has no leading 1, so reserve exponent value 0 just for number 0

IEEE 754 Floating Point Standard (2/3)

- IEEE 754 uses "biased exponent" representation
  - Designers wanted FP numbers to be used even if no FP hardware; e.g., sort records with FP numbers using integer compares
  - Wanted bigger (integer) exponent field to represent bigger numbers
  - 2's complement poses a problem (because negative numbers look bigger)
    - Use just magnitude and offset by half the range

IEEE 754 Floating Point Standard (3/3)

- Called Biased Notation, where bias is number subtracted to get final number
  - IEEE 754 uses bias of 127 for single prec.
  - Subtract 127 from Exponent field to get actual value for exponent

- Summary (single precision):

S (31)	Exponent (30–23)	Significand (22–0)
1 bit	8 bits	23 bits

- $(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$

  - Double precision identical, except with exponent bias of 1023 (half, quad similar)

“Father” of the Floating point standard

IEEE Standard 754 for Binary Floating-Point Arithmetic.

1989
ACM Turing
Award Winner!

Prof. Kahan

www.cs.berkeley.edu/~wkahan/ieee754status/754story.html

Clickers

- Guess this Floating Point number:

1 1000 0000 1000 0000 0000 0000 0000 000

A: $-1x\ 2^{128}$

B: $+1x\ 2^{-128}$

C: $-1x\ 2^{1}$

D: $+1.5x\ 2^{-1}$

E: $-1.5x\ 2^{1}$

Administrivia

- Project 3-2 extended until 03/20 @ 23:59:59

- Guerrilla Session: Caches/ Proj 3-2 OH
  - Sat 3/19 1 - 3 PM @ 521 Cory

Representation for ± ∞

- In FP, divide by 0 should produce ± ∞, not overflow.
- Why?
  - OK to do further computations with ∞ E.g., X/0 > Y may be a valid comparison
- IEEE 754 represents ± ∞
  - Most positive exponent reserved for ∞
  - Significands all zeroes

Representation for 0

- **Represent 0?**
  - exponent all zeroes
  - significand all zeroes
  - What about sign? Both cases valid

```
+0: 0 00000000 00000000000000000000000
-0: 1 00000000 00000000000000000000000
```

Special Numbers

- What have we defined so far? (Single Precision)

Exponent	Significand	Object
0	0	0
0	nonzero	???
1-254	anything	+/- fl. pt. #
255	0	+/- ∞
255	nonzero	???

- Professor Kahan had clever ideas:
  - Wanted to use Exp=0,255 & Sig!=0

Representation for Not a Number

- What do I get if I calculate `sqrt(-4.0)` or `0/0`?
  - If ∞ not an error, these shouldn't be either
  - Called Not a Number (NaN)
  - Exponent = 255, Significand nonzero
- Why is this useful?
  - Hope NaNs help with debugging?
  - They contaminate: op(NaN, X) = NaN
  - Can use the significand to identify which!

Representation for Denorms (1/2)

- **Problem: There's a gap among representable FP numbers around 0**
  - Smallest representable pos num:

    $a = 1.0\ldots_{two} * 2^{-126} = 2^{-126}$

  - Second smallest representable pos num:

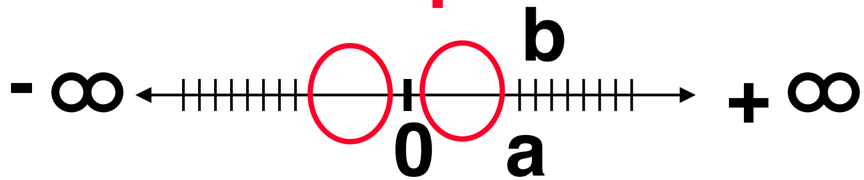
    $b = 1.000\ldots\ldots1_{two} * 2^{-126}$
    $= (1 + 0.00\ldots1_{two}) * 2^{-126}$
    $= (1 + 2^{-23}) * 2^{-126}$
    $= 2^{-126} + 2^{-149}$

$a - 0 = 2^{-126}$

$b - a = 2^{-149}$

Normalization and implicit 1 is to blame!

Gaps!

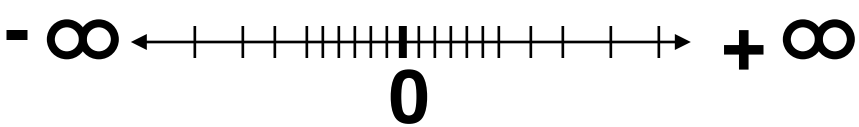


Representation for Denorms (2/2)

- **Solution:**
  - We still haven't used Exponent = 0, Significand nonzero
  - DEnormalized number: no (implied) leading 1, implicit exponent = -126.
  - Smallest representable pos num:

    $a = 2^{-149}$
  - Second smallest representable pos num:

    $b = 2^{-148}$



Special Numbers Summary

- Reserve exponents, significands:

Exponent	Significand	Object
0	0	0
0	nonzero	Denorm
1-254	anything	+/- fl. pt. #
255	0	+/- ∞
255	nonzero	NaN

www.h-schmidt.net/FloatApplet/IEEE754.html

Conclusion

Exponent tells Significand how much (2^i) to count by (…, 1/4, 1/2, 1, 2, …)

- Floating Point lets us:
  - Represent numbers containing both integer and fractional parts; makes efficient use of available bits.
  - Store approximate values for very large and very small #s.

Can store NaN, ± ∞

- IEEE 754 Floating-Point Standard is most widely accepted attempt to standardize interpretation of such numbers (Every desktop or server computer sold since ~1997 follows these conventions)

- Summary (single precision):

31	30 … 23	22 … 0
S	Exponent	Significand
1 bit	8 bits	23 bits

- $(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$

- Double precision identical, except with exponent bias of 1023 (half, quad similar)