Computer Systems

CS107

Cynthia Lee

Today's Topics

LECTURE:

- › Floating point!
- › (if we have time)
  - Preview of Friday's assembly language lecture

Real Numbers and Approximation

MATH TIME!

Some preliminary observations on approximation

- We know that some non-integer numbers can be represented precisely by a finite-length decimal value
  - $1/5_{10} = 0.2_{10}$
- …and some can't (in base 10)
  - π
  - $1/3_{10} = .3333333\overline{3}_{10}$

- But this isn't consistent across different bases
  - $1/3_{10} = 0.1_{3}$
  - $1/5_{10} = 0.012012\overline{012}_{3}$

- Uncountably many real numbers we could represent!
- And of course even with `int`(countable) we needed to make choices about which integers make the cut and which don't, when *we have only a limited storage space*

Real number types: `float`, `double`

- If you're deciding how to divide up a large number of possible bit patterns into numbers they can represent, there are any number of ways you cold do this
  - What are your priorities?
  - Do you want to leave space for very big ones? Or very small ones?
  - Do you prefer positive or negative?

Thought experiment: how we could represent real numbers

1. Take our integer representation
2. Decide how much granularity we want to represent
3. Move the decimal point (binary point?) over a corresponding number of places

- For example: let's say we want to be able to measure in units of $1/64^{th}$
  - 32-bit binary number example (10 and $1/64^{th}$):
    - 0000 0000 0000 0000 0000 0010 10.00 0001

Real number types: `float`, `double`

- `float` is 4 bytes = 32 bits
  - 2^{32} = 4,294,967,296 possible values
  - (note: obviously same number of possible values as 32-bit int)
- `double` is 8 bytes = 64 bits
  - 2^{64} = 18,446,744,073,709,551,616 possible
- This is the same number of different possible representable values as int and long, of course
  - › But floating point range goes much higher than max int:
  - › Max 32-bit int: $2^{31} - 1 = 2{,}147{,}483{,}647 \approx 2.1 \times \mathbf{10^{9}}$
  - › Max 32-bit float: $(2 - 2^{-23}) \times 2^{127} \approx 3.4 \times \mathbf{10^{38}}$

The "mini-float"

NOT A REAL TYPE IN C, BUT WILL SHOW US THE PRINCIPLES THAT WILL SCALE UP TO THE ACTUAL BIT SIZES FLOAT AND DOUBLE

IEEE format, squished down to example-size "mini-float"

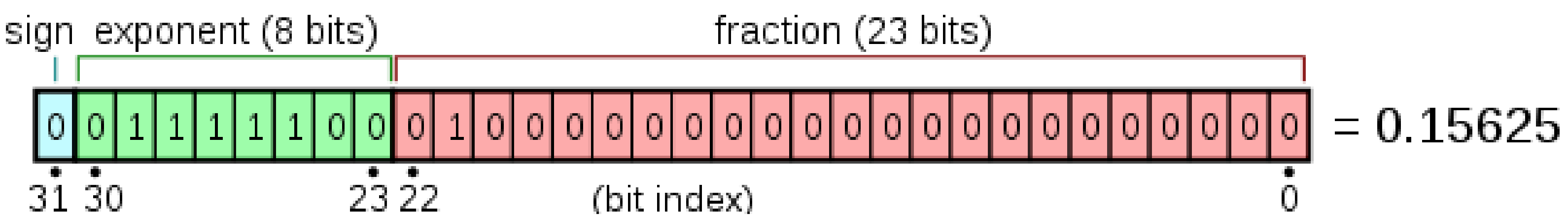
Sign	Exponent				Mantissa		

- Sign bit is 1 for negative, 0 for positive
  - Operates completely independently, *unlike* 2's complement int
- The rest of the number is in something like scientific notation:
  - (+/-) M * 2^{Exp}
    - Mantissa provides the significant digits of the number
      - *Note:* also called the coefficient or the significand
    - Exponent is used to scale the number up or down to very large or very small values
  - Both exponent and mantissa are binary numbers, 4 bits and 3 bits, respectively
    - *but not in normal 2's complement form!*

IEEE format, squished down to example-size “mini-float”

Sign	Exponent				Mantissa		

- Compare to the bit distribution for 32-bit `float`:



"Mini-float": the mantissa

- A 3-bit binary number representing a number after an implicit 1.
  - What??
- Yes, it looks like this: 1.*[mantissa digits]*
- Examples:
  - Mantissa = $100_2 \rightarrow 1.100_2$
    - $= 1 + ½ = 1.5_{10}$
  - Mantissa = $110_2 \rightarrow 1.110_2$
    - $= 1 + ½ + ¼ = 1.75_{10}$

Sign	Exponent				Mantissa		

“Mini-float”: the exponent

- A 4-bit binary number representing the exponent:
  - `0000` — `reserved`
  - `0001` — `-6`
  - `0010` — `-5`
  - `0011` — `-4`
  - `0100` — `-3`
  - `0101` — `-2`
  - `0110` — `-1`
  - `0111` — `0`
  - `1000` — `1`
  - `1001` — `2`
  - `1010` — `3`
  - `1011` — `4`
  - `1100` — `5`
  - `1101` — `6`
  - `1110` — `7`
  - `1111` — `reserved`

Sign	Exponent				Mantissa		

Lets do an example!

Sign	Exponent				Mantissa		
0	0	0	0	1	0	1	0

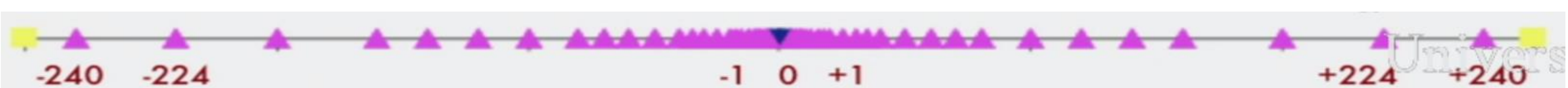
- This number is:
  A. Greater than 0
  B. Less than 0
  C. Help!
- Extra credit: > 1 or < -1 or in between?

Lets do an example!

+/-	Exponent				Mantissa			Value
0	0	0	0	1	0	1	0	$10/8 * 2^{-6} = 10/512$
0	0	0	0	1	0	1	1	$11/8 * 2^{-6} = 11/512$
0	0	0	0	1	1	0	0	$12/8 * 2^{-6} = 12/512$
0	0	0	0	1	1	0	1	$13/8 * 2^{-6} = 13/512$
0	0	0	0	1	1	1	0	$14/8 * 2^{-6} = 14/512$
0	0	0	0	1	1	1	1	$15/8 * 2^{-6} = 15/512$
0	0	0	1	0	0	0	0	$1 * 2^{-5} = 16/512$
0	0	0	1	0	0	0	1	$9/8 * 2^{-6} = 18/512$

Lets do an example!

+/-	Exponent				Mantissa			Value
0	0	0	1	0	0	0	0	$1 * 2^{-5} = 16/512$
0	0	0	1	0	0	0	1	$9/8 * 2^{-6} = 18/512$
0	1	1	1	0	0	1	0	$10/8 * 2^{7} = 160$
0	1	1	1	0	0	1	1	$11/8 * 2^{7} = 176$
0	1	1	1	0	1	0	0	$12/8 * 2^{7} = 192$
0	1	1	1	0	1	0	1	$13/8 * 2^{7} = 208$
0	1	1	1	0	1	1	0	$14/8 * 2^{7} = 224$
0	1	1	1	0	1	1	1	$15/8 * 2^{7} = 240$



About those reserved exponents…

Denormalized and Special cases

Reserved exponent values: 0000 and 1111

0000 exponent:

- If the mantissa is all zeros: +/-0

Sign	Exponent				Mantissa		
any	0	0	0	0	0	0	0

- If the mantissa is nonzero: denormalized floats

Sign	Exponent				Mantissa
any	0	0	0	0	any nonzero

1111 exponent:

- If the mantissa is all zeros: +/-INF (infinity)

Sign	Exponent				Mantissa		
any	1	1	1	1	0	0	0

- If the mantissa is nonzero: NaN (not a number)

Sign	Exponent				Mantissa
any	1	1	1	1	any nonzero

MORE about: the mantissa

- A 3-bit binary number representing a number <u>after an implicit 1.</u>
  - What??
- Yes, it looks like this: 1.*[mantissa digits]*
- Examples:
  - Mantissa = $100_2 \rightarrow 1.100_2$
    - $= 1 + ½ = 1.5_{10}$
  - Mantissa = $110_2 \rightarrow 1.110_2$
    - $= 1 + ½ + ¼ = 1.75_{10}$
- **If the exponent is all zeros, then there is no implicit leading 1**
  - The exponent is implicitly smallest possible exponent (2^{-6} for mini-float)
  - This allows us to eke out a few numbers that are even (closer to zero) than would otherwise be possible
  - Called “denormalized” or “denorm” floats

Sign	Exponent				Mantissa		

Comparing float and int

ADVANTAGES AND DISADVANTAGES

Comparing float and int

- 32-bit integer (type **`int`**):
  - -2,147,483,648 to 2147483647
  - Every integer in that range can be represented
- 64-bit integer (type **`long`**):
  - −9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
    - (Even Psy will have a hard time overflowing this!)

- 32-bit floating point (type **`float`**):
  - ~1.7 x10^{-38} to ~3.4 x10^{38}
  - Not all numbers in the range can be represented (obviously—uncountable)
    - Not even all integers in the range can be represented!
    - Gaps can get quite large!! (larger the exponent, larger the gap between successive mantissa values)
- 64-bit floating point (type **`double`**):
  - ~9 x10^{-307} to ~2 x10^{308}

Doing arithmetic in float

FUN TIMES

Adding two mini-floats

- Your bank account balance on your 12th birthday was \$128.00
- Your bank stores this amount as a mini-float:

Sign	Exponent				Mantissa		

- Each week of your childhood, starting at age 12, you deposited your weekly allowance of \$8:

Sign	Exponent				Mantissa		

- **What was your account balance on the day you turned 18?**

A. About \$10000
B. About \$5000
C. About \$2500
D. About \$100
E. Other

Doing arithmetic in float

FUN TIMES

Adding two mini-floats

- Your bank account balance on your 12th birthday was $128.00
- **Your bank stores this amount as a mini-float:**

Sign	Exponent				Mantissa		

- Each week, starting at age 12, you deposited your weekly allowance of $14:

Sign	Exponent				Mantissa		

- **What was your account balance on the day you turned 18?**

A. About $10000
B. About $5000
C. About $2500
D. About $100
E. Other

Instruction Set Architectures

SOME CONTEXT AND TERMINOLOGY

Instruction Set Architecture

- **The ISA defines:**
  - Operations that the processor can execute
  - Data transfer operations + how to access data
  - Control mechanisms like branch, jump (think loops and if-else)
  - Contract between programmer/compiler and hardware
- **Layer of abstraction:**
  - Above:
    - Programmer/compiler can write code for the ISA
    - New programming languages can be built on top of the ISA as long as the compiler will do the translation
  - Below:
    - New hardware can implement the ISA
    - Can have even potentially radical changes in hardware implementation
    - Have to “do” the same thing from programmer point of view
- **ISAs have incredible inertia!**
  - Legacy support is a huge issue for x86-64

Two major categories of Instruction Set Architectures

- **CISC:**
  - **Complex** instruction set computers
    - e.g., x86 **(CS107 studies this)**
  - Have special instructions for each thing you might want to do
  - Can write code with fewer instructions, because each instruction is very expressive
- **RISC:**
  - **Reduced** instruction set computers
    - e.g., MIPS
  - Have only a very tiny number of instructions, optimize the heck out of them in the hardware
  - Code may need to be longer because you have to go roundabout ways of achieving what you wanted