

Great Ideas in Computer Architecture

Floating Point and Running a Program

Instructor: Nick Riasanovsky



“Floating Point is a peaceful, free game about using a grappling hook to swing yourself gracefully through randomly generated spaces. The only objective is to collect points, and the only thing that increases your points is swinging swiftly, smoothly and elegantly without hitting anything.”

Floating Point on Steam

store.steampowered.com › All Games › Casual

★★★★★ Rating: 10/10 - 3,138 reviews

Jun 6, 2014 - **Floating Point** is a peaceful, free

game about gracefully through randomly generated spaces

User reviews:

RECENT: **Very Positive** (54 reviews)

OVERALL: **Overwhelmingly Positive** (3,139 reviews)

Release Date: Jun 6, 2014

Popular user-defined tags for this product:

Free to Play

Casual

Indie

Relaxing

Physics

+

Review

- *Instruction formats* designed to be similar but still capable of handling all instructions

31	30	25	24	21	20	19	15	14	12	11	8	7	6	0				
funct7				rs2			rs1		funct3		rd			opcode		R-type		
imm[11:0]						rs1		funct3		rd			opcode		I-type			
imm[11:5]				rs2			rs1		funct3		imm[4:0]			opcode		S-type		
imm[12]		imm[10:5]			rs2			rs1		funct3		imm[4:1]		imm[11]		opcode		B-type
imm[31:12]										rd			opcode		U-type			
imm[20]		imm[10:1]			imm[11]		imm[19:12]			rd			opcode		J-type			

- Branches and Jumps move relative to current address
- Assembly/Disassembly: Use RISC-V Green Sheet to convert

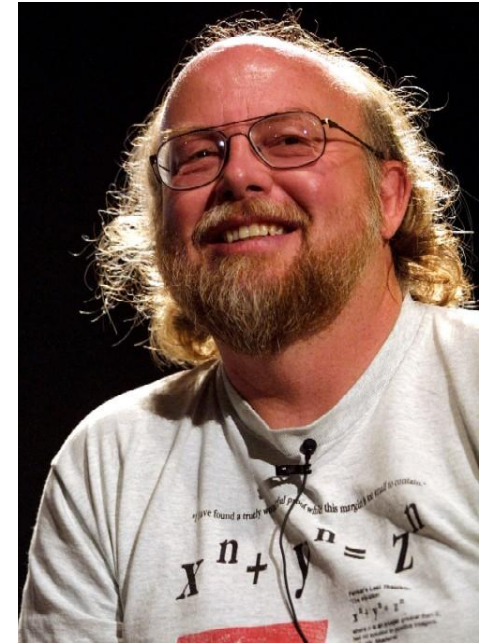
Agenda

- **Floating Point**
- Floating Point Special Cases
- Administritivia
- Floating Point Limitations
- Compiler and Assembler
- Meet the Staff
- Linker and Loader
- Bonus: FP Conversion Practice, Casting

Floating Point Quote

“95% of the folks out there are completely clueless about floating-point.”

— James Gosling
Feb. 28, 1998



James Gosling
Sun Fellow
Java Inventor

Number Representation Revisited

- Given 32 bits (a *word*), what can we represent so far?
 - Signed and Unsigned Integers
 - 4 Characters (ASCII)
 - Instructions & 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**

Goals of Floating Point

- Support a wide range of values
 - Both very small and very large
- Keep as much *precision* as possible
- Help programmer with errors in real arithmetic
 - Support $+\infty$, $-\infty$, Not-A-Number (NaN), exponent overflow and underflow

“Father” of Floating Point Standard

IEEE Standard 754 for
Binary
Floating-Point
Arithmetic

**1989
ACM Turing
Award Winner!**



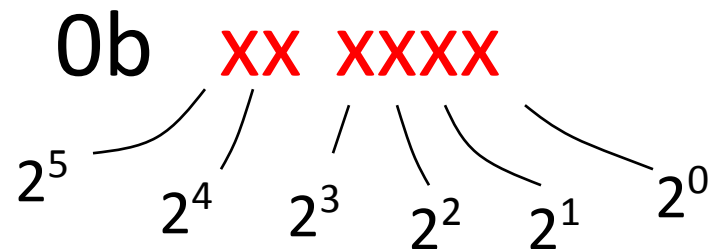
Prof. Kahan
Prof. Emeritus
UC Berkeley

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

Reasoning about Fractions

Big Idea: Why can't we represent fractions? Because our bits all represent nonnegative powers of 2.

Example 6-bit representation:



Example: $10\ 1010_{\text{two}} = 1 \times 2^5 + 1 \times 2^3 + 1 \times 2^1 = 42_{\text{ten}}$

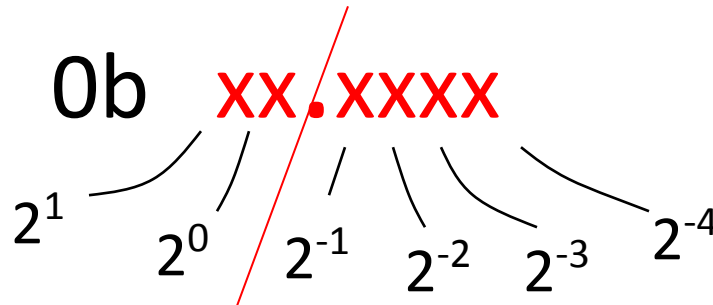
- The lowest power of 2 is 2^0 , so the ***smallest*** difference between any two numbers is $2^0 = 1$.

Example: $10\ 1011_{\text{two}} = 42 + 1 = 43_{\text{ten}}$

Representation of Fractions

New Idea: Introduce a fixed “Binary Point” that signifies boundary between negative & nonnegative powers:

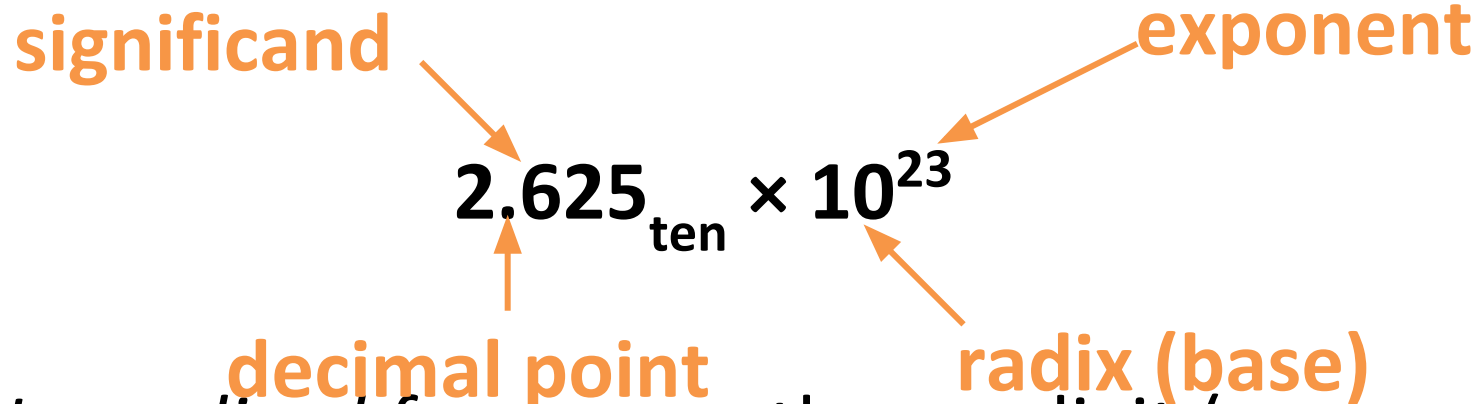
Example 6-bit
representation:



Example: $10.1010_{\text{two}} = 1 \times 2^1 + 1 \times 2^{-1} + 1 \times 2^{-3} = 2.625_{\text{ten}}$

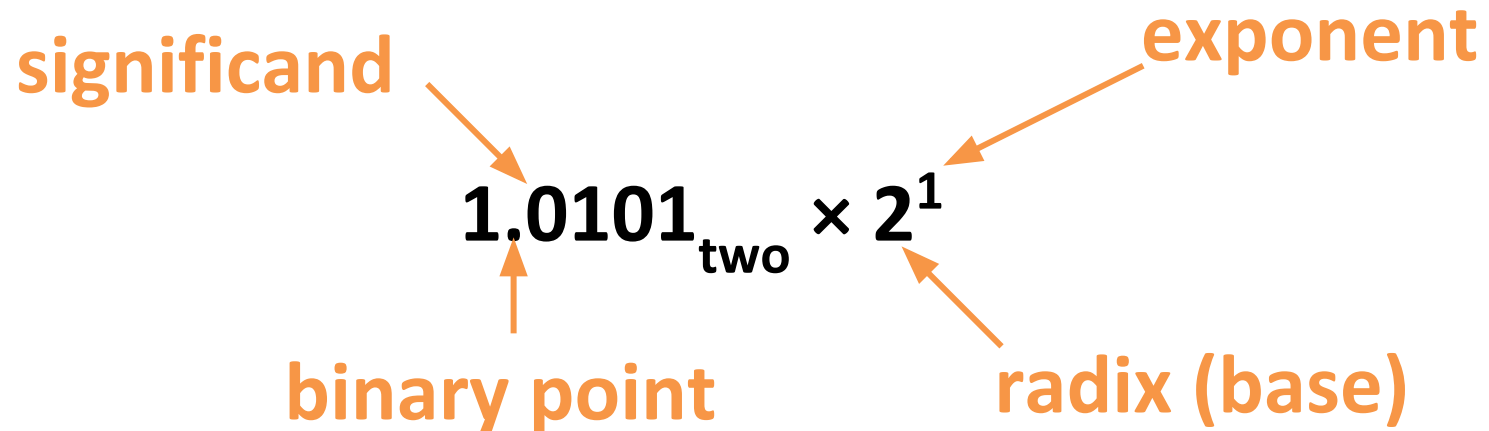
Binary point numbers that match the 6-bit format above range from 0 (00.0000_{two}) to 3.9375 (11.1111_{two})

Scientific Notation (Decimal)



- *Normalized form*: exactly one digit (non-zero) to left of decimal point (the point “floats” to be in the standard position)
- 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 (Binary)



- Computer arithmetic that supports this called **floating point** due to the “floating” of the binary point
 - Declare such variable in C as `float`

Translating To and From Scientific Notation

- Consider the number $1.011_{\text{two}} \times 2^4$
- To convert to ordinary number, shift the decimal to the right by 4
 - Result: $10110_{\text{two}} = 22_{\text{ten}}$
- For negative exponents, shift decimal to the left
 - $1.011_{\text{two}} \times 2^{-2} \Rightarrow 0.01011_{\text{two}} = 0.34375_{\text{ten}}$
- Go from ordinary number to scientific notation by shifting until in *normalized* form
 - $1101.001_{\text{two}} \Rightarrow 1.101001_{\text{two}} \times 2^3$

Floating Point Encoding

- Use normalized, Base 2 scientific notation:

$$+1.\textcolor{red}{xxx}\dots\textcolor{red}{x}_{\text{two}} \times 2^{\textcolor{blue}{yyy}\dots\textcolor{blue}{y}_{\text{two}}}$$

- Split a 32-bit word into 3 fields:



– **S** represents **Sign** (1 is negative, 0 positive)

– **Exponent** represents **y**'s

– **Significand** represents **x**'s

– Key Idea: More like Sign & Magnitude

Exponent Comparison

- Which is smaller? (i.e. closer to $-\infty$)

0 or 1×10^{-127} ?

1×10^{-126} or 1×10^{-127} ?

-1×10^{-127} or 0 ?

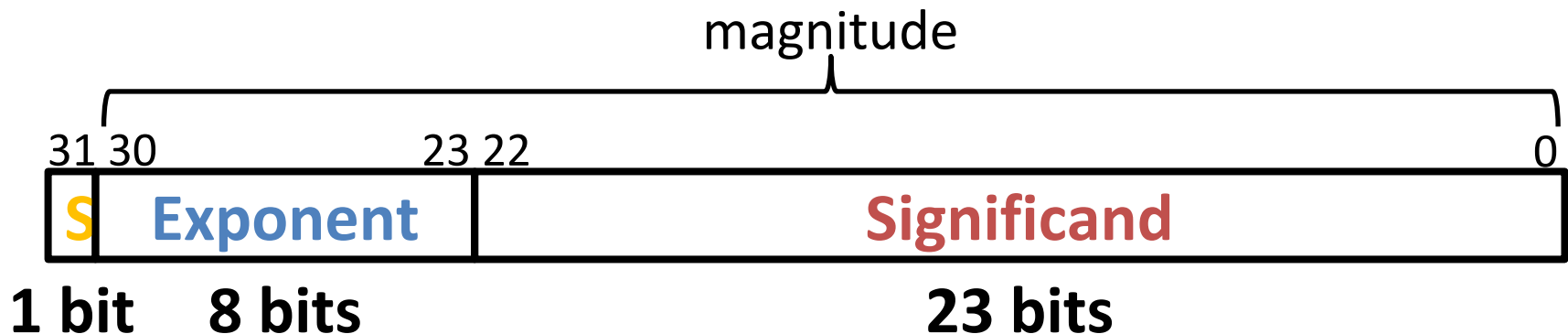
-1×10^{-126} or -1×10^{-127} ?

The Exponent Field

- Use **biased notation**
 - Read exponent as unsigned, but with *bias* of 127
 - Exp field ranges from -127 (00000000_{two}) to 128 (11111111_{two})
 - Exponent 0 is represented as $01111111_{\text{two}} = 127_{\text{ten}}$
- To encode in biased notation, add the bias (add 127) then encode in unsigned:
 - If we had 2^1 , $\text{exp} = 1 \Rightarrow 128 \Rightarrow 10000000_{\text{two}}$
 - 2^{127} : $\text{exp} = 127 \Rightarrow 254 \Rightarrow 11111110_{\text{two}}$

The Exponent Field

- Why use **biased notation** for the exponent?
 - Remember that we want floating point numbers to look small when their actual value is small
 - We don't like how in 2's complement, -1 looks bigger than 0. Bias notation preserves the linearity of value
- Recall that only the first bit denotes sign
 - Thus, floating point resembles sign and magnitude



Floating Point Encoding



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

- What does this mean?
 - Think of it as: $(1 + \text{Value of Significand})$
 - Since the Significand represents all the negative powers of 2, its total value is always < 1
 - Example: $1.0101_{\text{two}} = 1 + 2^{-2} + 2^{-4} = 1.3125$

Floating Point Encoding



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

- Note the implicit 1 in front of the Significand
 - Ex: 0011 1111 1100 0000 0000 0000 0000 0000_{two}
 - $(-1)^0 \times (1 . 1_{\text{two}}) \times 2^{(127-127)} = (1 . 1_{\text{two}}) \times 2^{(0)}$
 - $1.1_{\text{two}} = 1.5_{\text{ten}}$, NOT $0.1_{\text{two}} = 0.5_{\text{ten}}$
 - Gives us an extra bit of precision

Precision and Accuracy

Don't confuse these two terms!

Precision is a count of the number of bits in a computer word used to represent a value

Accuracy is a measure of the difference between the *actual value of a number* and its computer representation

- *High precision permits high accuracy but doesn't guarantee it. It is possible to have high precision but low accuracy.*
- **Example:** `float pi = 3.14;`
 - `pi` will be represented using all 24 bits of the significand (highly precise), but is only an approximation (not accurate)


Agenda

- Floating Point
- **Floating Point Special Cases**
- Administivia
- Floating Point Limitations
- Compiler and Assembler
- Meet the Staff
- Linker and Loader
- Bonus: FP Conversion Practice, Casting

Floating Point Numbers Summary

Exponent	Significand	Meaning
0x00	?	?
0x00	?	?
0x01 – 0xFE	anything	\pm Num
0xFF	?	?
0xFF	?	?

Representing Zero

- But wait... what happened to zero?
 - Using standard encoding 0x00000000 is $1.0 \times 2^{-127} \neq 0$
 - All because of that dang implicit 1 
 - *Special case:* Exp and Significand all zeros = 0
 - Two zeros! But at least 0x00000000 = 0 like integers

Floating Point Numbers Summary

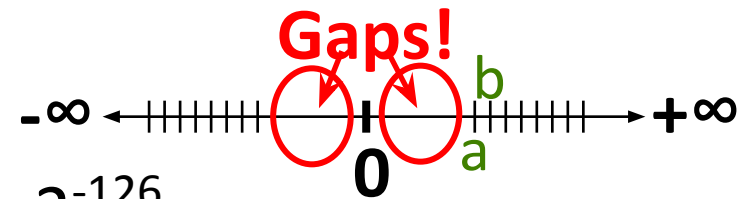
Exponent	Significand	Meaning
0x00	0	± 0
0x00	?	?
0x01 – 0xFE	anything	$\pm \text{Num}$
0xFF	?	?
0xFF	?	?

Representing Very Small Numbers

- What are the normal numbers closest to 0?
(here, normal means the exponent is **nonzero**)

– $a = 1.\textcolor{red}{0}\dots\textcolor{red}{00}_{\text{two}} \times 2^{1-127} = (1+0) \times 2^{-126} = 2^{-126}$


– $b = 1.\textcolor{red}{0}\dots\textcolor{red}{01}_{\text{two}} \times 2^{1-127} = (1+2^{-23}) \times 2^{-126} = 2^{-126} + 2^{-149}$



- The gap between 0 and a is 2^{-126}
- The gap between a and b is 2^{-149}
- We want to represent numbers between 0 and a
 - How? The implicit 1 forces the 2^{-126} term to stay :(
 - Solution: Take out the implicit 1 !
- *Special case:* $\text{Exp} = 0$, $\text{Significand} \neq 0$ are **denorm**

Denorm Numbers

- Short for “denormalized numbers”
 - No leading 1
 - Careful! **Denorm exponent bias is now 126** when $\text{Exp} = 0x00$ (intuitive reason: the floating point moves one more bit to the left of the leading bit)
- Now what do the gaps look like?
 - Smallest denorm: $\pm 0.0\dots01_{\text{two}} \times 2^{-126} = \pm 2^{-149}$
 - Largest denorm: $\pm 0.1\dots1_{\text{two}} \times 2^{-126} = \pm (2^{-126} - 2^{-149})$
 - Smallest norm: $\pm 1.0\dots0_{\text{two}} \times 2^{-126} = \pm 2^{-126}$

So much
closer to 0


No uneven gap! Increments by 2^{-149}

Floating Point Numbers Summary

Exponent	Significand	Meaning
0x00	0	± 0
0x00	non-zero	\pm Denorm Num
0x01 – 0xFE	anything	\pm Norm Num
0xFF	?	?
0xFF	?	?

Other Special Cases

- **Exp** = 0xFF, **Significand** = 0: $\pm \infty$
 - e.g. division by 0
 - Still work in comparisons
- **Exp** = 0xFF, **Significand** \neq 0: Not a Number (**NaN**)
 - e.g. square root of negative number, 0/0, $\infty - \infty$
 - NaN propagates through computations
 - In theory: **Significand** can be useful for debugging

Other Special Cases

- Largest value (besides ∞)?
 - $\text{Exp} = 0xFF$ has now been taken!
 - $\text{Exp} = 0xFE$ has largest: $1.1\dots1_{\text{two}} \times 2^{127} = 2^{128} - 2^{104}$

Floating Point Numbers Summary

Exponent	Significand	Meaning
0x00	0	± 0
0x00	non-zero	\pm Denorm Num
0x01 – 0xFE	anything	\pm Norm Num
0xFF	0	$\pm \infty$
0xFF	non-zero	NaN

Agenda

- Floating Point
- Floating Point Special Cases
- **Administrivia**
- Floating Point Limitations
- Compiler and Assembler
- Meet the Staff
- Linker and Loader
- Bonus: FP Conversion Practice, Casting

Administrivia

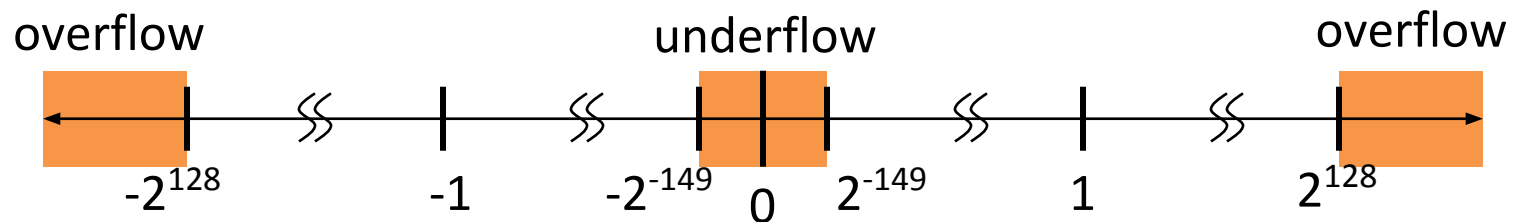
- Project 1 due tomorrow
- Project 2-1 released tomorrow, due (7/6)
- HW1 due tonight, HW2 due Monday (7/2)
- No lecture or lab on July 4th (7/4)
- Midterm is next Tuesday (7/3)
 - Covers material up to the beginning of Monday's lecture (CALL is the last MT1 topic)
 - TA-run review Session, Saturday 6/30 from 2-4PM in 306 Soda
 - Guerilla session Sunday, 2-4PM in Cory 540AB
- Check your iClicker participation on bCourses in the “Grades” tab

Agenda

- Floating Point
- Floating Point Special Cases
- Administtrivia
- **Floating Point Limitations**
- Compiler and Assembler
- Meet the Staff
- Linker and Loader
- Bonus: FP Conversion Practice, Casting

Floating Point Limitations (1/2)

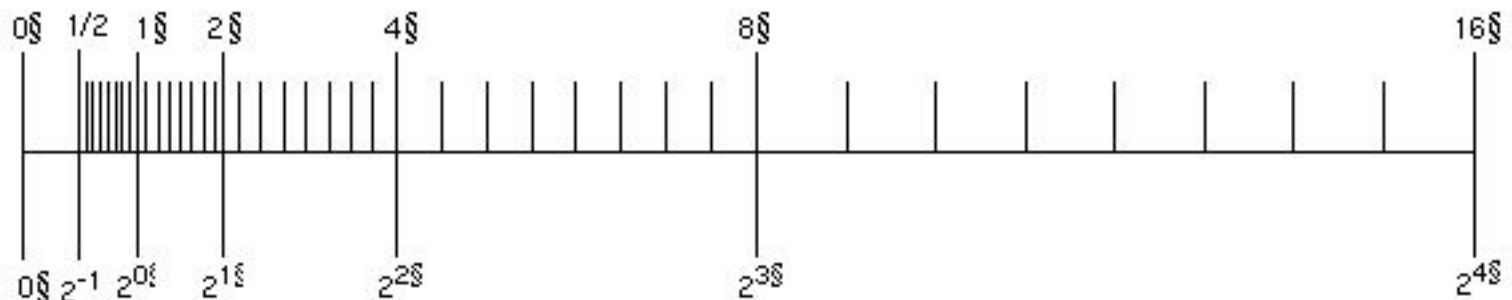
- What if result x is too large? ($\text{abs}(x) > 2^{128}$)
 - **Overflow**: Exponent is larger than can be represented
- What if result x too small? ($0 < \text{abs}(x) < 2^{-149}$)
 - **Underflow**: Negative exponent is larger than can be represented



- What if result runs off the end of the Significand?
 - **Rounding** occurs and can lead to unexpected results
 - FP has different *rounding modes*

Floating Point Gaps

- Does adding 0x00000001 always add the same value to the floating point number?
- NO—it's value depends on the exponent field
- ex: $1.0_{\text{two}} \times 2^2 = 4$ $\rightarrow +2$ $1.0_{\text{two}} \times 2^3 = 8$ $\rightarrow +4$
 $1.1_{\text{two}} \times 2^2 = 6$ $\rightarrow +2$ $1.1_{\text{two}} \times 2^3 = 12$ $\rightarrow +4$
- Thus floating points are quite different from the number representations you've learned so far



Floating Point Limitations (2/2)

- FP addition is NOT associative!
 - You can find Big and Small numbers such that:
 $\text{Small} + \text{Big} + \text{Small} \neq \text{Small} + \text{Small} + \text{Big}$
 - This is due to *rounding* errors: FP *approximates* results because it only has 23 bits for **Significand**
- Despite being seemingly “more accurate,” FP cannot represent all integers
 - e.g. $2^{24} + 2^{23} + 1 = 25165825$ (try it!)
 - Be careful when casting between `int` and `float`

Question:

Let $FP(1,2)$ = # of floats between 1 and 2

Let $FP(2,3)$ = # of floats between 2 and 3

Which of the following statements is true?

Hint: Try representing the numbers in FP

(A) $FP(1,2) > FP(2,3)$

(B) $FP(1,2) = FP(2,3)$

(C) $FP(1,2) < FP(2,3)$

(D) It depends

Question:

Let $FP(1,2)$ = # of floats between 1 and 2

Let $FP(2,3)$ = # of floats between 2 and 3

Which of the following statements is true?

Hint: Try representing the numbers in FP

(A) $FP(1,2) > FP(2,3)$

(B) $FP(1,2) = FP(2,3)$

(C) $FP(1,2) < FP(2,3)$

(D) It depends

$$1 = 1.0 \times 2^0$$

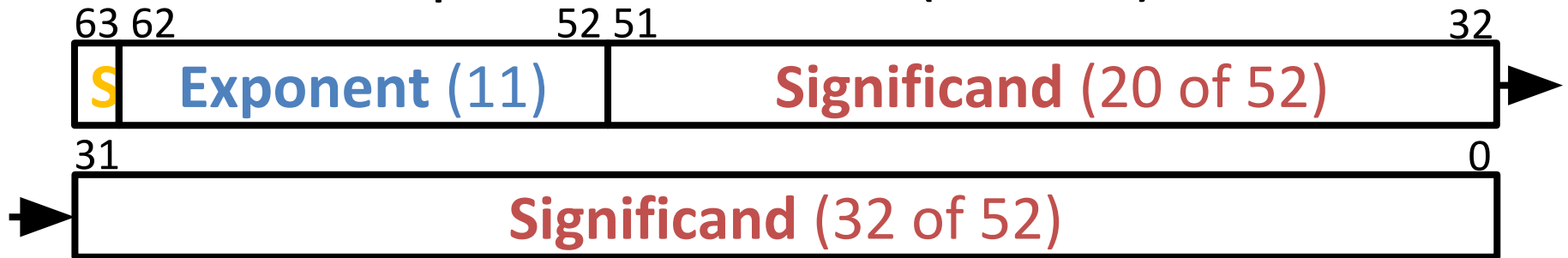
$$2 = 1.0 \times 2^1$$

$$3 = 1.1 \times 2^1$$

$$FP(1,2) \approx 2^{23}, FP(2,3) \approx 2^{22}$$

Double Precision FP Encoding

- Next multiple of word size (64 bits)

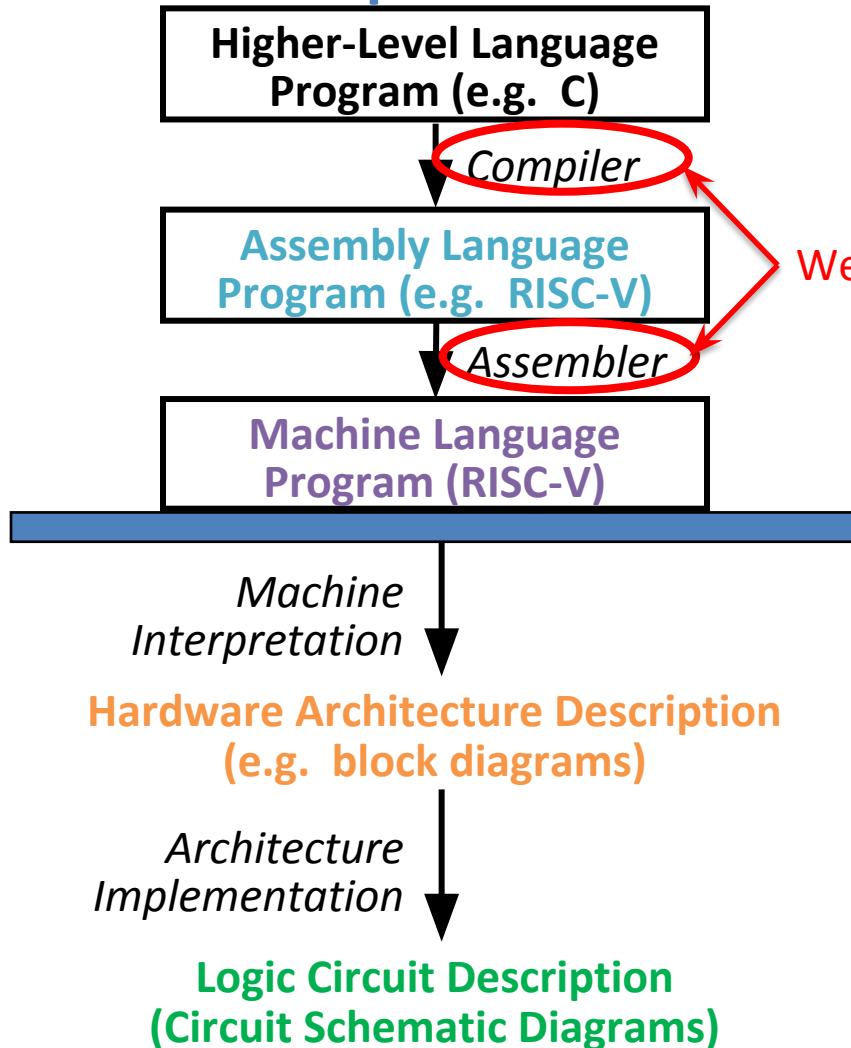


- **Double Precision** (vs. Single Precision)
 - C variable declared as `double`
 - Exponent bias is $2^{10}-1 = 1023$
 - Smallest denorm: 2^{-1074} , Largest (not ∞): $2^{1024} - 2^{971}$
 - Primary advantage is greater precision due to larger Significand

Agenda

- Floating Point
- Floating Point Special Cases
- Administtrivia
- Floating Point Limitations
- **Compiler and Assembler**
- Meet the Staff
- Linker and Loader
- Bonus: FP Conversion Practice, Casting

Great Idea #1: Levels of Representation/Interpretation

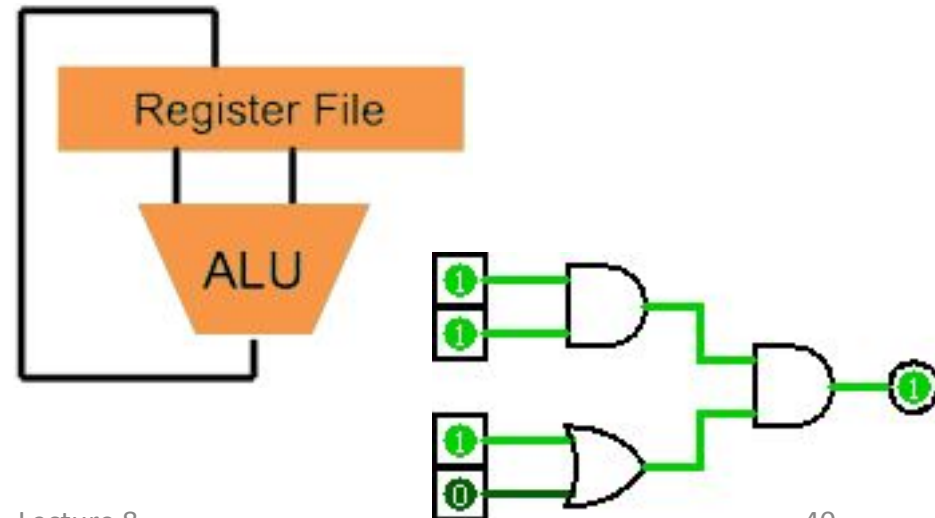


```
temp = v[k];  
v[k] = v[k+1];  
v[k+1] = temp;
```

We are here

```
lw    t0, 0(x2)  
lw    t1, 4(x2)  
sw    t1, 0(x2)  
sw    t0, 4(x2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000  
1010 1111 0101 1000 0000 1001 1100 0110  
1100 0110 1010 1111 0101 1000 0000 1001  
0101 1000 0000 1001 1100 0110 1010 1111
```



COMPILER



ASSEMBLER



LINKER



LOADER



Translation vs. Interpretation (1/3)

- How do we run a program written in a source language?
 - **Interpreter**: Directly executes a program in the source language
 - **Translator**: Converts a program from the source language to an equivalent program in another language
- Directly *interpret* a high level language when efficiency is not critical
- *Translate* to a lower level language when increased performance is desired

Translation vs. Interpretation (2/3)

- Generally easier to write an interpreter
- Interpreter closer to high-level, so can give better error messages (e.g. Python, Venus)
- Interpreter is slower ($\sim 10x$), but code is smaller ($\sim 2x$)
- Interpreter provides instruction set independence: can run on any machine

Translation vs. Interpretation (3/3)

- Translated/compiled code almost always more efficient and therefore higher performance
 - Important for many applications, particularly operating systems
- Translation/compilation helps “hide” the program “source” from the users
 - One model for creating value in the marketplace (e.g. Microsoft keeps all their source code secret)
 - Alternative model, “open source”, creates value by publishing the source code and fostering a community of developers

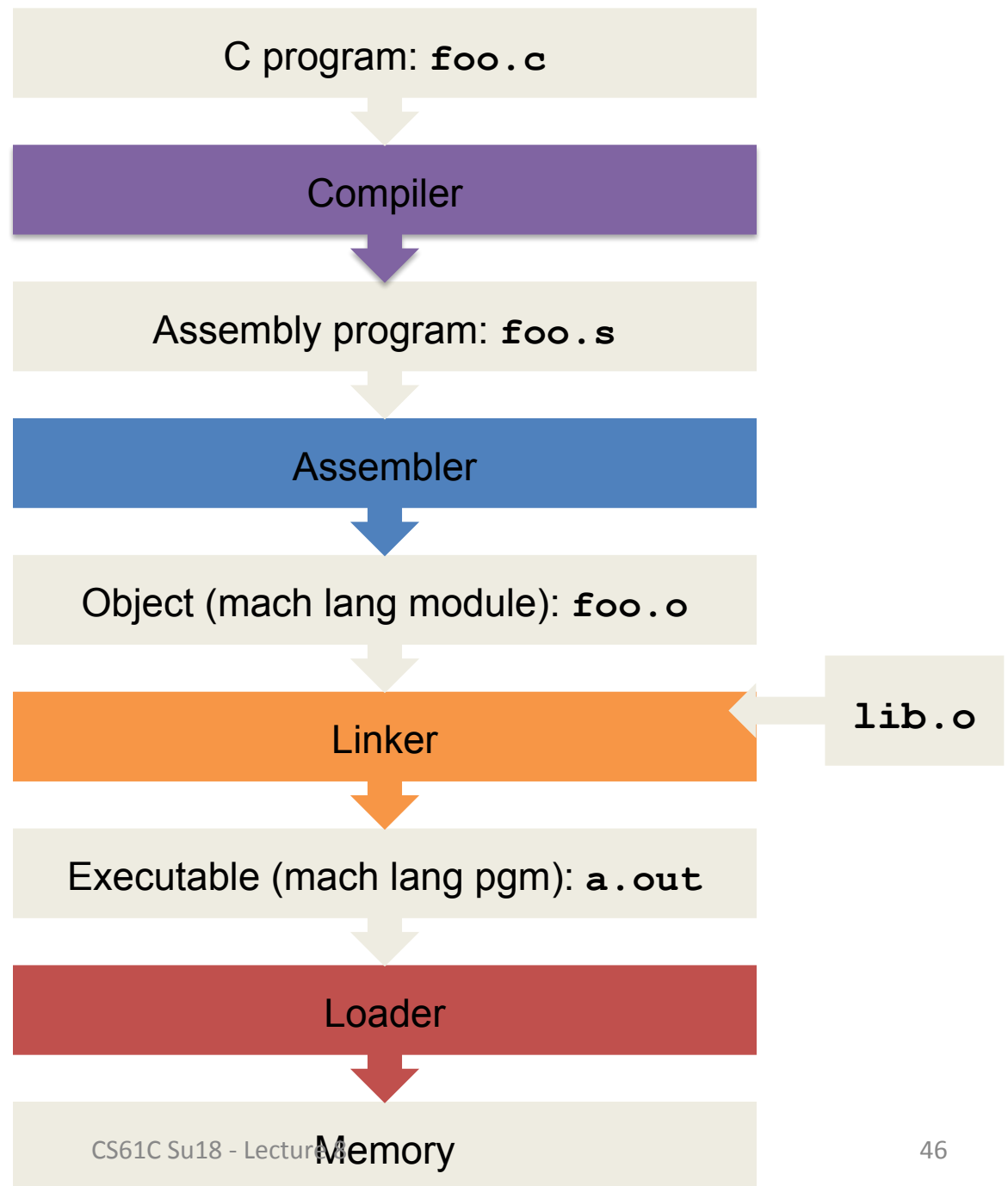
C Translation

- **Recall:** A key feature of C is that it allows you to compile files *separately*, later combining them into a single executable
- What can be accessed across files?
 - Functions
 - Static/global variables

C Translation

Steps to Starting a Program:

- 1) **C**ompiler
- 2) **A**sembler
- 3) **L**inker
- 4) **L**oader



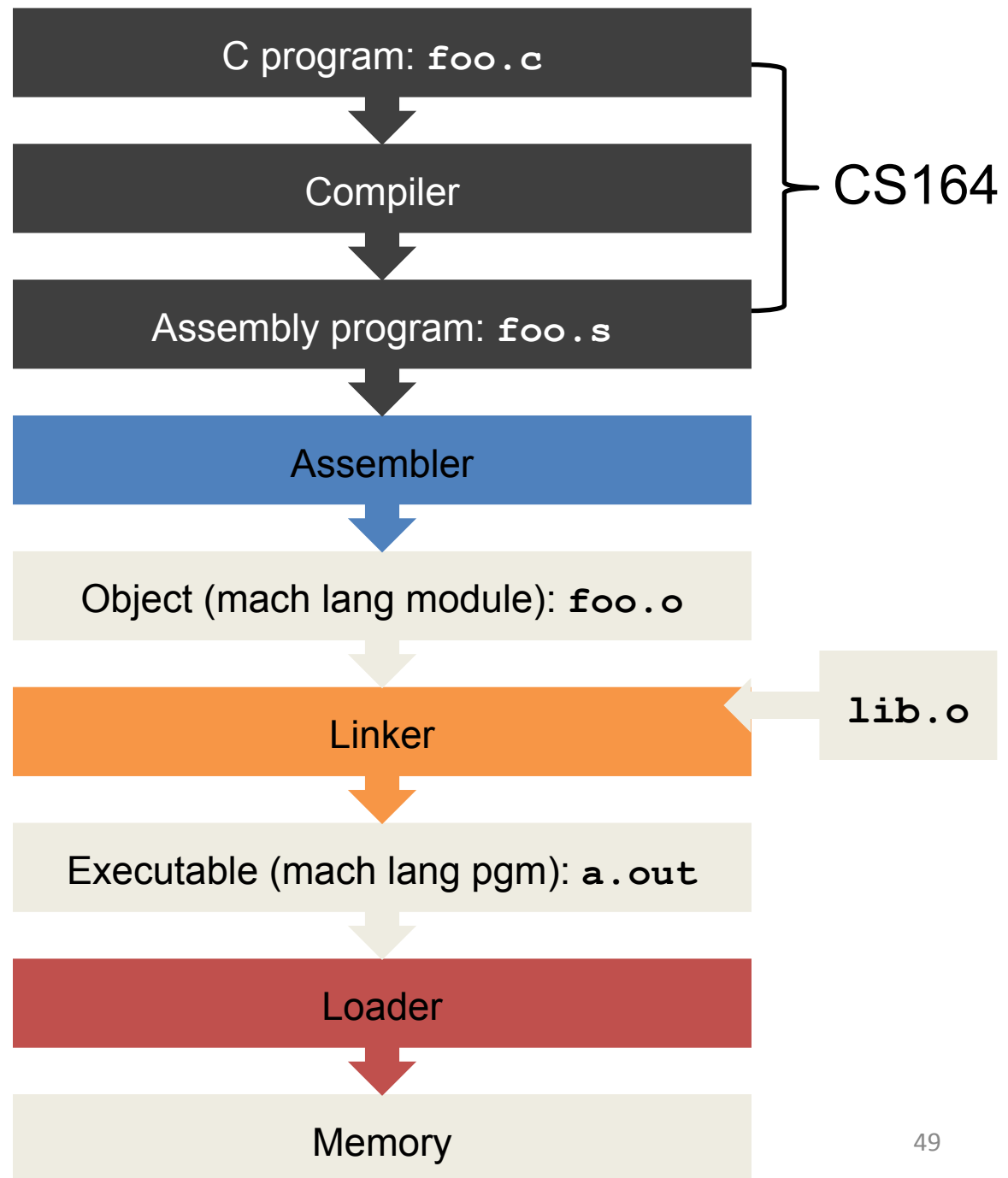
Compiler

- **Input:** Higher-level language (HLL) code (e.g. C, Java in files such as `foo.c`)
- **Output:** Assembly Language Code (e.g. `foo.s` for RISC-V)
- Note that the output may contain pseudo-instructions
- In reality, there's a preprocessor step before this to handle `#directives` but it's not very exciting

Compilers Are Non-Trivial

- There's a whole course about them – CS164
 - We won't go into further detail in this course
 - For the very curious and highly motivated:
<http://www.sigbus.info/how-i-wrote-a-self-hosting-c-compiler-in-40-days.html>
- Some examples of the task's complexity:
 - Operator precedence: $2 + 3 * 4$
 - Operator associativity: $a = b = c;$
 - Determining locally whether a program is valid
 - `if (a) { if (b) { ... /*long distance*/ ... } } } //extra bracket`

- Translation
- Compiler
- Administtrivia
- **Assembler**
- Linker
- Loader
- Example



Assembler

- **Input:** Assembly language code (e.g. `foo.s` for RISC-V)
- **Output:** Object code (True Assembly), information tables (e.g. `foo.o` for RISC-V)
 - Object file
- Reads and uses **directives**
- Replaces pseudo-instructions
- Produces machine language

Assembler Directives

(For more info, see p.B-5 and B-7 in P&H)

- Give directions to assembler, but do not produce machine instructions
 - `.text`: Subsequent items put in user text segment (machine code)
 - `.data`: Subsequent items put in user data segment (binary rep of data in source file)
 - `.globl sym`: declares `sym` global and can be referenced from other files
 - `.ascii str`: Store the string `str` in memory and null-terminates it
 - `.word w1...wn`: Store the n 32-bit quantities in successive memory words

Pseudo-instruction Replacement

Pseudo

mv t0, t1
neg t0, t1
li t0, imm
not t0, t1
beqz t0, loop
la t0, str

Real

addi t0,t1,0
sub t0, zero, t1
addi t0, zero, imm
xori t0, t1, -1
beq t0, zero, loop
lui t0, str[31:12]
addi t0, t0, str[11:0]

OR

auipc t0, str[31:12]
addi t0, t0, str[11:0]

Producing Machine Language (1/3)

- Simple Cases
 - Arithmetic and logical instructions, shifts, etc.
 - All necessary info contained in the instruction
- What about Branches and Jumps?
 - Branches and Jumps require a *relative address*
 - Once pseudo-instructions are replaced by real ones, we know by how many instructions to branch, so no problem

Producing Machine Language (2/3)

- “Forward Reference” problem
 - Branch instructions can refer to labels that are “forward” in the program:

```
      or    s0, x0, x0
L1:    slt   t0, x0, a1
      beq   t0, x0, L2
      addi  a1, a1, -1
      j     L1
L2:    add   t1, a0, a1
```

- Solution: Make two passes over the program

Two Passes Overview

- Pass 1:
 - Expands pseudo instructions encountered
 - Remember position of labels
 - Take out comments, empty lines, etc
 - Error checking
- Pass 2:
 - Use label positions to generate relative addresses (for branches and jumps)
 - Outputs the object file, a collection of instructions in binary code

Producing Machine Language (3/3)

- What about jumps to external labels?
 - Requiring knowing a final address
 - Forward or not, can't generate machine instruction without knowing the position of instructions in memory
- What about references to data?
 - `la` gets broken up into `lui` and `ori`
 - These will require the full 32-bit address of the data
- These can't be determined yet, so we create two tables...

Symbol Table

- List of “items” that may be used by other files
 - *Each* file has its own symbol table
- What are they?
 - **Labels**: function calling
 - **Data**: anything in the `.data` section; variables may be accessed across files
- Keeping track of the labels fixes the forward reference problem

Relocation Table

- List of “items” this file will need the address of later (currently undetermined)
- What are they?
 - Any external **label** jumped to: `jal` or `jalr`
 - internal
 - external (including library files)
 - Any piece of **data**
 - such as anything referenced in the `data` section

Object File Format

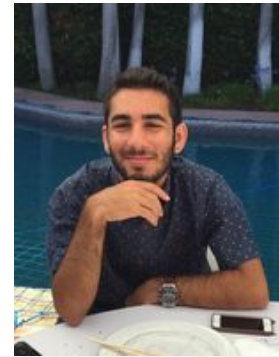
- 1) **object file header**: size and position of the other pieces of the object file
- 2) **text segment**: the machine code
- 3) **data segment**: data in the source file (binary)
- 4) **relocation table**: identifies lines of code that need to be “handled”
- 5) **symbol table**: list of this file’s labels and data that can be referenced
- 6) **debugging information**
 - A standard format is ELF (except MS)

http://www.skyfree.org/linux/references/ELF_Format.pdf

Assembler

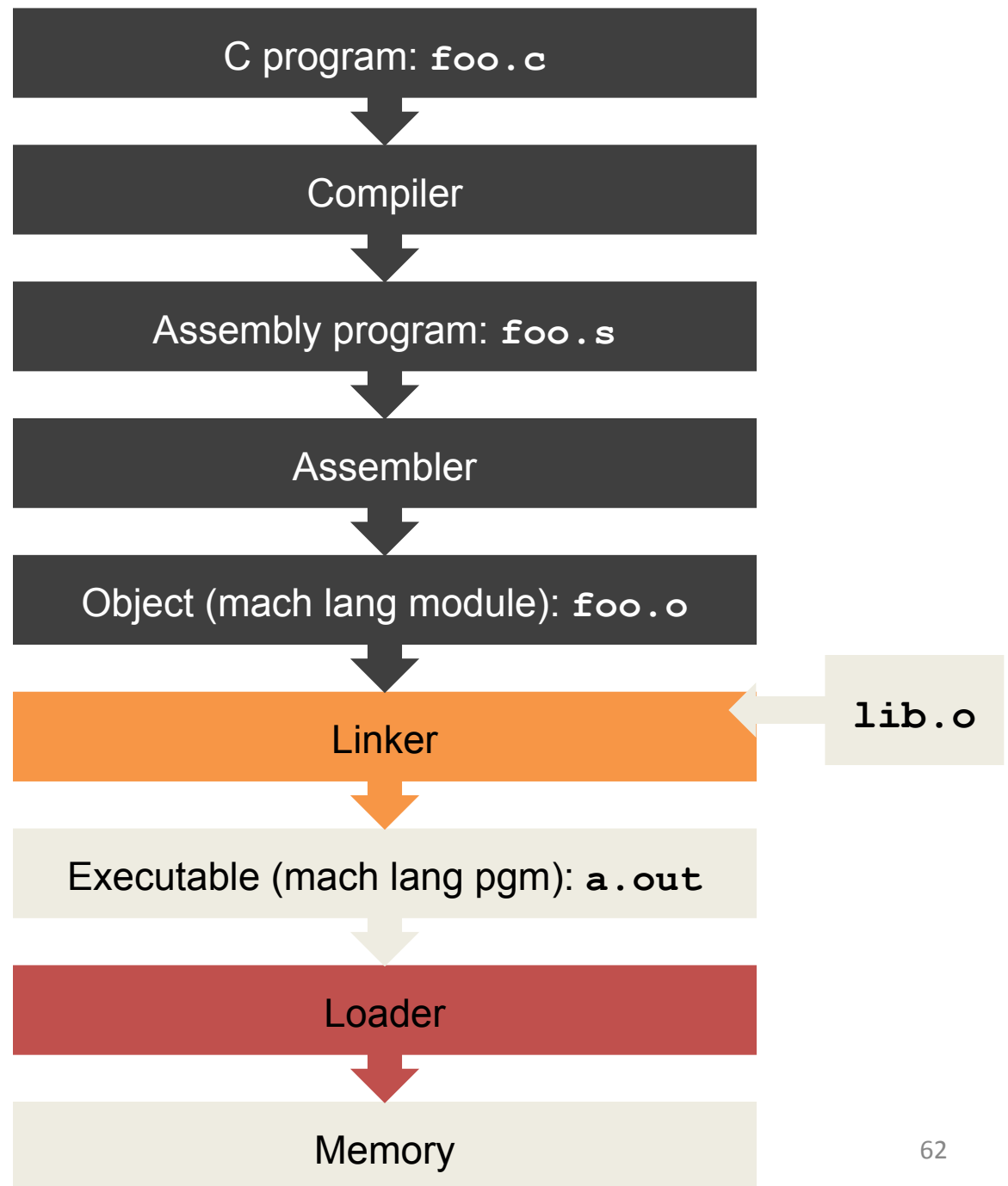
- **Input:** Assembly language code (e.g. `foo.s` for RISC-V)
- **Output:** Object code (True Assembly), information tables (e.g. `foo.o` for RISC-V)
 - Object file
- Reads and uses **directives**
- Replaces pseudo-instructions
- Produces machine language

Meet the Staff



	Emaan	Sruthi	Sean
Roadside Sign	"The end is near"	"Do not stop"	"No parking" "Parking all day"
Greatest Weakness	Unhealthy sleeping habits	Being late	Unhealthy eating habits
Favorite artist	The Weeknd	Red Hot Chili Peppers	Tyler the Creator
Favorite meme of all time	Savage Patrick	Spongebob	"E"

- Translation
- Compiler
- Administtrivia
- Assembler
- **Linker**
- Loader
- Example

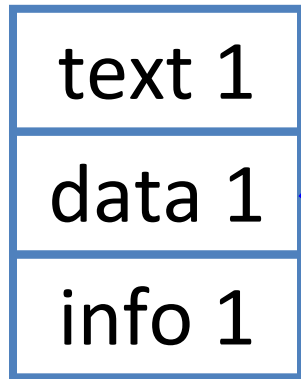


Linker (1/3)

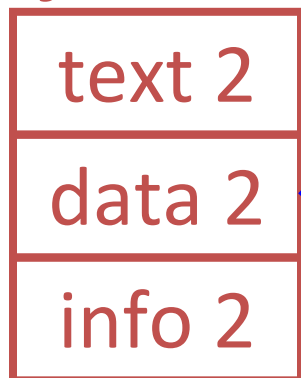
- **Input:** Object Code files, information tables (e.g. `foo.o`, `lib.o` for RISC-V)
- **Output:** Executable Code (e.g. `a.out` for RISC-V)
- Combines several object (`.o`) files into a single executable (“**linking**”)
- **Enables separate compilation of files**
 - Changes to one file do not require recompilation of whole program
 - Old name “Link Editor” from editing the “links” in jump and link instructions

Linker (2/3)

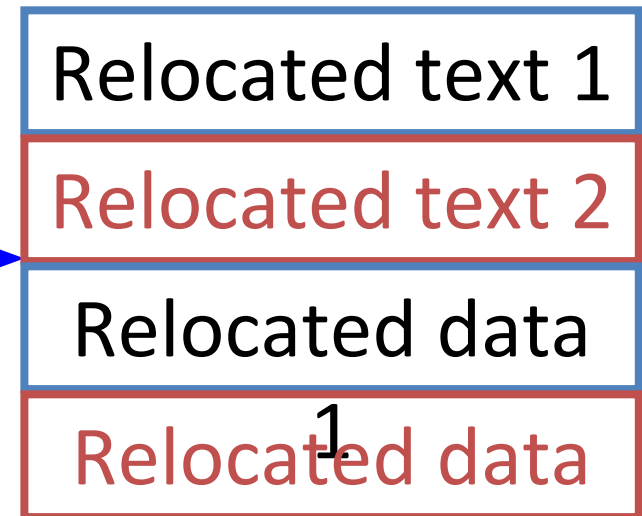
object file 1



object file 2



a.out



1
2

Linker (3/3)

- 1) Take text segment from each .o file and put them together
- 2) Take data segment from each .o file, put them together, and concatenate this onto end of text segments
- 3) Resolve References
 - Go through Relocation Table; handle each entry
 - i.e. **fill in all absolute addresses**

Three Types of Addresses

- PC-Relative Addressing (beq, bne, jal)
 - never relocate

External Function Reference (usually jal)

- always relocate

Static Data Reference (often auipc and addi)

- always relocate
- RISC-V often uses auipc rather than lui so that a big block of stuff can be further relocated as long as it is fixed relative to the pc

Absolute Addresses in RISC-V

- Which instructions need relocation editing?
 - J-format: jump/jump and link

XXXXX	jal
-------	-----

- Loads and stores to variables in static area, relative to global pointer

xxx	gp	rd	lw
xx	rs1	gp	sw

- What about conditional branches?

rs1	rs2	x	beq
-----	-----	---	-----

- PC-relative addressing preserved even if code moves

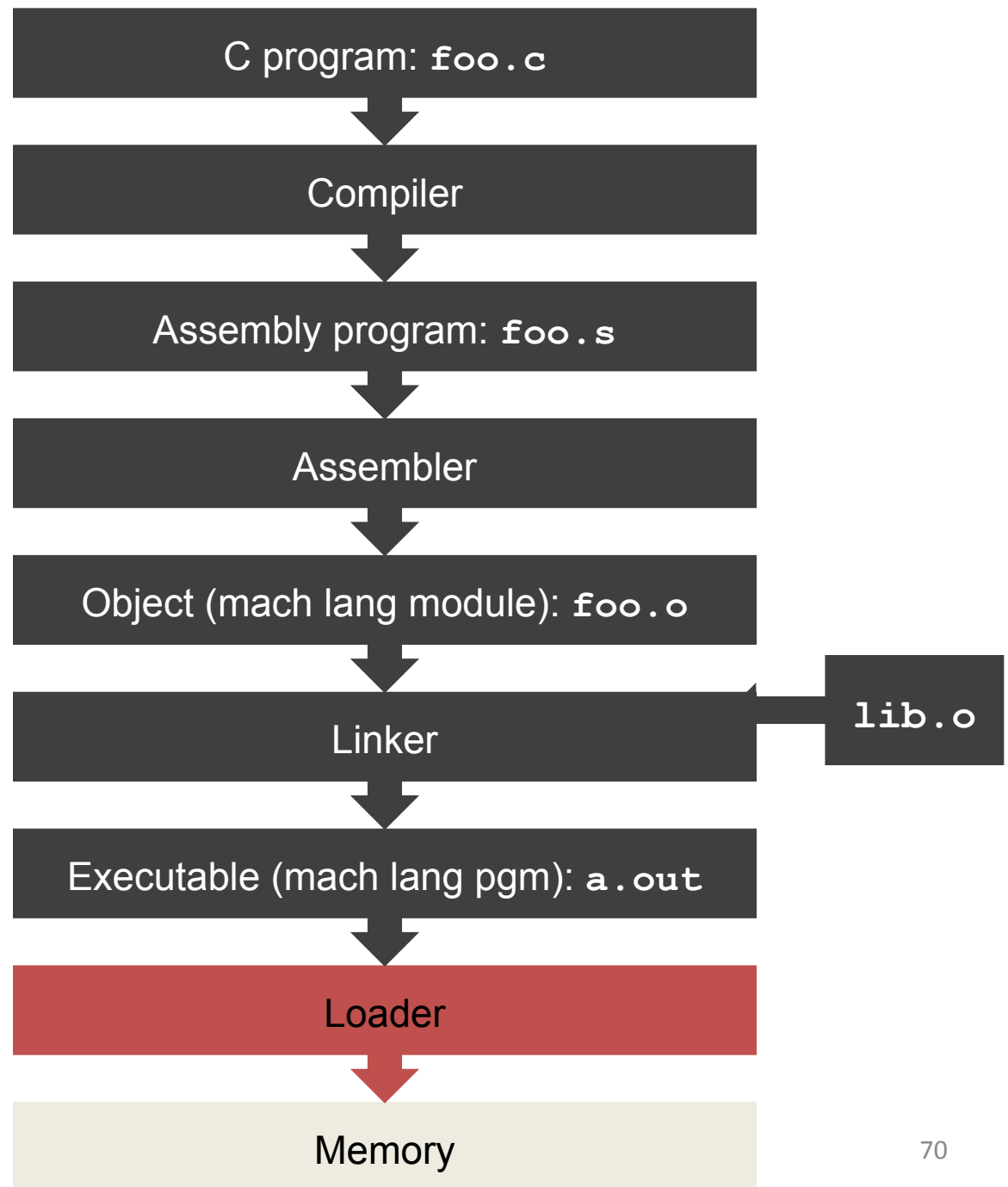
Resolving References (1/2)

- Linker assumes the first word of the first text segment is at **0x10000** for RV32.
 - More later when we study “virtual memory”
- Linker knows:
 - Length of each text and data segment
 - Ordering of text and data segments
- Linker calculates:
 - Absolute address of each label to be jumped to (internal or external) and each piece of data being referenced

Resolving References (2/2)

- To resolve references:
 - 1) Search for reference (data or label) in all “user” symbol tables
 - 2) If not found, search library files (e.g. `printf`)
 - 3) Once absolute address is determined, fill in the machine code appropriately
- Output of linker: executable file containing text and data (plus header)

- Translation
- Compiler
- Administrative
- Assembler
- Linker
- **Loader**
- Example



Loader

- **Input:** Executable Code (e.g. `a.out` for RISC-V)
- **Output:** <program is run>
- Executable files are stored on disk
- When one is run, loader's job is to load it into memory and start it running
- In reality, loader is the operating system (OS)
 - loading is one of the OS tasks

Loader

- 1) Reads executable file's header to determine size of text and data segments
- 2) Creates new address space for program large enough to hold text and data segments, along with a stack segment
<more on this later>
- 3) Copies instructions and data from executable file into the new address space

Loader

- 4) Copies arguments passed to the program onto the stack
- 5) Initializes machine registers
 - Most registers cleared, but stack pointer assigned address of 1st free stack location
- 6) Jumps to start-up routine that copies program's arguments from stack to registers and sets the PC
 - If main routine returns, start-up routine terminates program with the exit system call

IClicker Question

At what point in process are all the machine code bits determined for the following assembly instructions:

1) `add x6, x7, x8`

2) `jal x1, fprintf`

A: 1) & 2) After compilation

B: 1) After compilation, 2) After assembly

C: 1) After assembly, 2) After linking

D: 1) After assembly, 2) After loading

IClicker Question

At what point in process are all the machine code bits determined for the following assembly instructions:

1) `add x6, x7, x8`

2) `jal x1, fprintf`

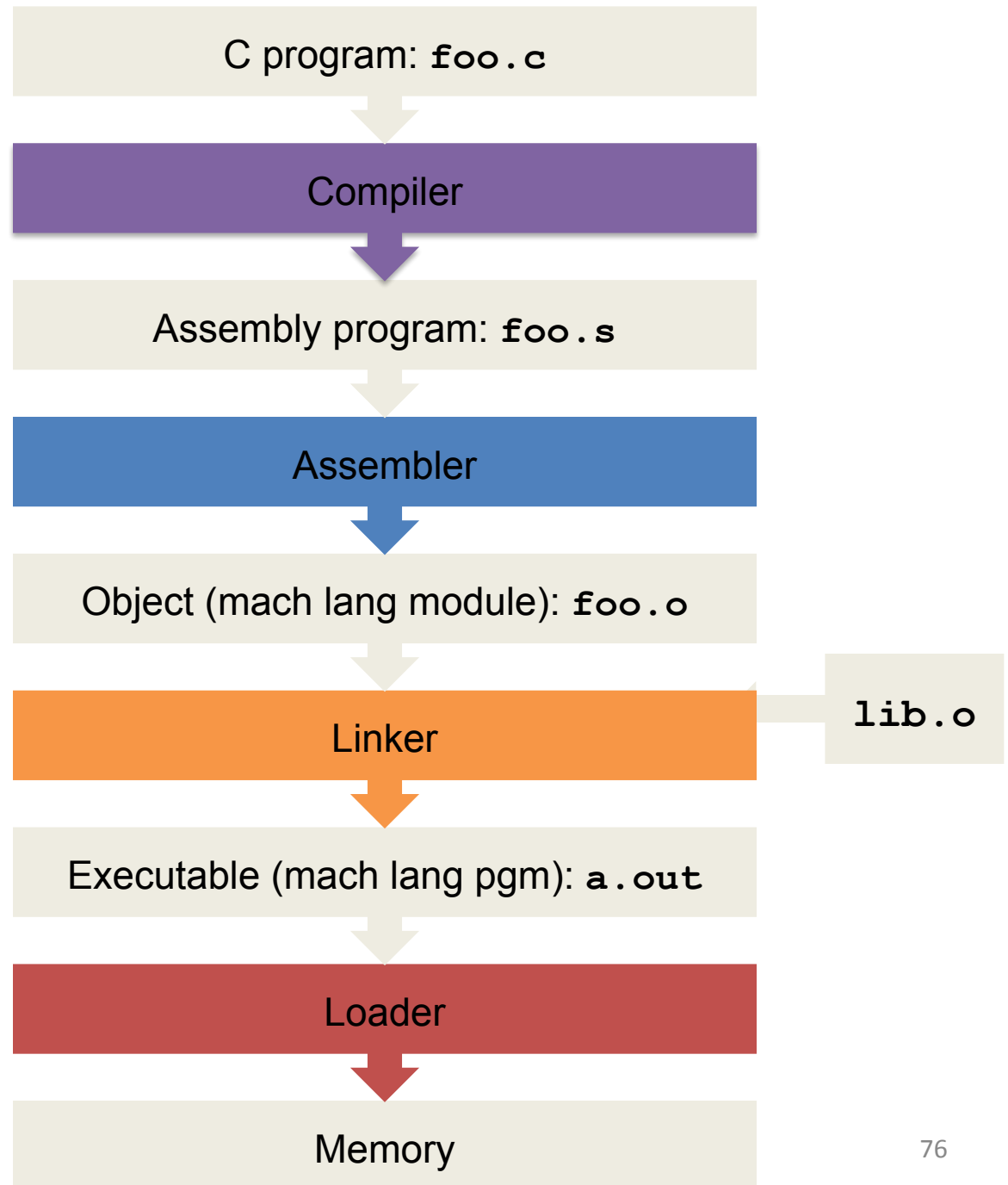
A: 1) & 2) After compilation

B: 1) After compilation, 2) After assembly

C: 1) After assembly, 2) After linking

D: 1) After assembly, 2) After loading

- Translation
- Compiler
- Administrivia
- Assembler
- Linker
- Loader
- Example



C.A.L.L. Example

```
#include <stdio.h>
int main()
{
    printf("Hello, %s\n", "world");
    return 0;
}
```

Compiled Hello.c: Hello.s

```
.text
    .align 2
    .globl main
main:
    addi sp, sp, -16
    sw    ra, 12(sp)
    lui   a0, %hi(string1)
    addi  a0, a0, %lo(string1)
    lui   a1, %hi(string2)
    addi  a1, a1, %lo(string2)
    call  printf
    lw    ra, 12(sp)
    addi  sp, sp, 16
    li    a0, 0
    ret
    .section .rodata
    .balign 4
string1:
    .string "Hello, %s!\n"
string2:
    .string "world"
```

```
# Directive: enter text section
# Directive: align code to 2^2 bytes
# Directive: declare global symbol
# label for start of main
# allocate stack frame
# save return address
# compute address of
#   string1
# compute address of
#   string2
# call function printf
# restore return address
# deallocate stack frame
# load return value 0
# return
# Directive: enter read-only data
section
# Directive: align data section to
bytes
# label for first string
# Directive: null-terminated string
# label for second string
# Directive: null-terminated string
```

Assembled Hello.s: Linkable Hello.o

```
000000000 <main>:
0:    ff010113 addi sp, sp, -16
4:    00112623 sw ra, 12(sp)
8:    00000537 lui a0, 0x0 # addr placeholder
c:    00050513 addi a0, a0, 0 # addr placeholder
10:   000005b7 lui a1, 0x0 # addr placeholder
14:   00058593 addi a1, a1, 0 # addr placeholder
18:   00000097 auipc ra, 0x0 # addr placeholder
1c:   000080e7 jalr ra # addr placeholder
20:   00c12083 lw ra, 12(sp)
24:   01010113 addi sp, sp, 16
28:   00000513 addi a0, a0, 0
2c:   00008067 jalr ra
```

Linked Hello.o: a.out

```
000101b0 <main>:
  101b0: ff010113 addi sp, sp, -16
  101b4: 00112623 sw    ra, 12(sp)
  101b8: 00021537 lui    a0, 0x21
  101bc: a1050513 addi   a0, a0, -1520 # 20a10
<string1>
  101c0: 000215b7 lui    a1, 0x21
  101c4: a1c58593 addi   a1, a1, -1508 # 20a1c
<string2>
  101c8: 288000ef jal    ra, 10450      # <printf>
  101cc: 00c12083 lw     ra, 12(sp)
  101d0: 01010113 addi   sp, sp, 16
  101d4: 000000513 addi   a0, 0, 0
  101d8: 000008067 jalr   ra
```


Summary (1/2)

- Floating point approximates real numbers:
 - Largest magnitude: $2^{128} - 2^{104}$ (**Exp** = 0xFE)
 - Smallest magnitude: 2^{-149} (denorm)
 - Also has encodings for 0, $\pm\infty$, NaN



Summary (2/2)

- **Compiler** converts a single HLL file into a single assembly file
 $.c \rightarrow .s$
- **Assembler** removes pseudo-instructions, converts what it can to machine language, and creates a checklist for linker (relocation table) $.s \rightarrow .o$
 - Resolves addresses by making 2 passes (for internal forward references)
- **Linker** combines several object files and resolves absolute addresses
 $.o \rightarrow .out$
 - Enable separate compilation and use of libraries
- **Loader** loads executable into memory and begins execution

BONUS SLIDES

The following material includes examples of floating point questions that are exam-worthy.

Though we may not have enough time to get to them in lecture, they have been prepared in a way that should be easily readable.

Question: Suppose we have the following floats in C:

$$\text{Big} = 2^{60}, \text{Tiny} = 2^{-15}, \text{BigNeg} = -\text{Big}$$

What will the following conditionals evaluate to?

- 1) $(\text{Big} * \text{Tiny}) * \text{BigNeg} == (\text{Big} * \text{BigNeg}) * \text{Tiny}$
- 2) $(\text{Big} + \text{Tiny}) + \text{BigNeg} == (\text{Big} + \text{BigNeg}) + \text{Tiny}$

	1	2
(A)	F	F
(B)	F	T
(C)	T	F
(D)	T	T

In general:

- (1) is TRUE as long as $\text{Big} * \text{BigNeg}$ doesn't overflow.
 (2) evaluates to $0 \neq \text{Tiny}$, which is FALSE as long as Tiny is at least 2^{24} times smaller than Big.

Agenda

- Performance
- Administtrivia
- Floating Point
- Floating Point Special Cases
- Floating Point Limitations
- **Bonus: FP Conversion Practice**
- Bonus: FP Casting Concerns

Example: Convert FP to Decimal

0	0110 1000	101 0101 0100 0011 0100 0010
---	-----------	------------------------------

- Sign: 0 means positive

- Exponent:

- $0110\ 1000_{\text{two}} = 104_{\text{ten}}$
 - Bias adjustment: $104 - 127 = -23$

- Significand:

$$\begin{aligned} &1 . 10101010100001101000010 \\ &= 1 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4} + 1 \times 2^{-5} + \dots \\ &= 1 + 2^{-1} + 2^{-3} + 2^{-5} + 2^{-7} + 2^{-9} + 2^{-14} + 2^{-15} + 2^{-17} + 2^{-22} \\ &= 1.0 + 0.666115 \end{aligned}$$

- Represents: $1.666115_{\text{ten}} \times 2^{-23} \approx 1.986 \times 10^{-7}$

Example: Scientific Notation to FP

-2.340625 x 10¹

1. Denormalize: -23.40625

2. Convert integer part:

$$23 = 16 + 4 + 2 + 1 = 10111_{\text{two}}$$

3. Convert fractional part:

$$.40625 = .25 + .125 + .03125 = 0.01101_{\text{two}}$$

4. Put parts together and normalize:

$$10111.01101 = 1.011101101 \times 2^4$$

5. Convert exponent: $4 + 127 = 10000011_{\text{two}}$

6.

1	1000 0011	011 1011 0100 0000 0000 0000
---	-----------	------------------------------

Example: Convert to FP

1/3

$$= 0.33333\dots_{\text{ten}}$$

$$= 0.25 + 0.0625 + 0.015625 + 0.00390625 + \dots$$

$$= 1/4 + 1/16 + 1/64 + 1/256 + \dots$$

$$= 2^{-2} + 2^{-4} + 2^{-6} + 2^{-8} + \dots$$

$$= 0.0101010101\dots_{\text{two}} \times 2^0$$

$$= 1.0101010101\dots_{\text{two}} \times 2^{-2}$$

— Sign: 0

— Exponent = $-2 + 127 = 125 = 01111101_{\text{two}}$

— Significand = $0101010101\dots_{\text{two}}$

0	0111 1101	010 1010 1010 1010 1010 1010
---	-----------	------------------------------

Agenda

- Performance
- Administtrivia
- Floating Point
- Floating Point Special Cases
- Floating Point Limitations
- Bonus: FP Conversion Practice
- **Bonus: FP Casting Concerns**

Casting floats to ints and vice versa

`(int) floating_point_expression`

Coerces and converts it to the *nearest* integer
(C uses truncation)

```
i = (int) (3.14159 * f);
```

`(float) integer_expression`

Converts integer to *nearest* floating point

```
f = f + (float) i;
```

float \rightarrow int \rightarrow float

```
if (f == (float)((int) f)) {  
    printf("true");  
}
```

- Will not always print “true”
 - Small floating point numbers (< 1) don’t have integer representations
- For other numbers, often will be rounding errors

int \rightarrow float \rightarrow int

```
if (i == (int) ((float) i)) {  
    printf("true");  
}
```

- Will not always print “true”
 - Many large values of integers don’t have exact floating point representations
- What about double?
 - Significand is now 52 bits, which can hold all of 32-bit integer, so will always print “true”