CS 61C

Discussion 4: Floating Point, CALL

John Yang
Summer 2019

Announcements

- Check In: tinyurl.com/john61c
- Project 2 being released soon!
- Homework 3 due 7/10 (Today!)

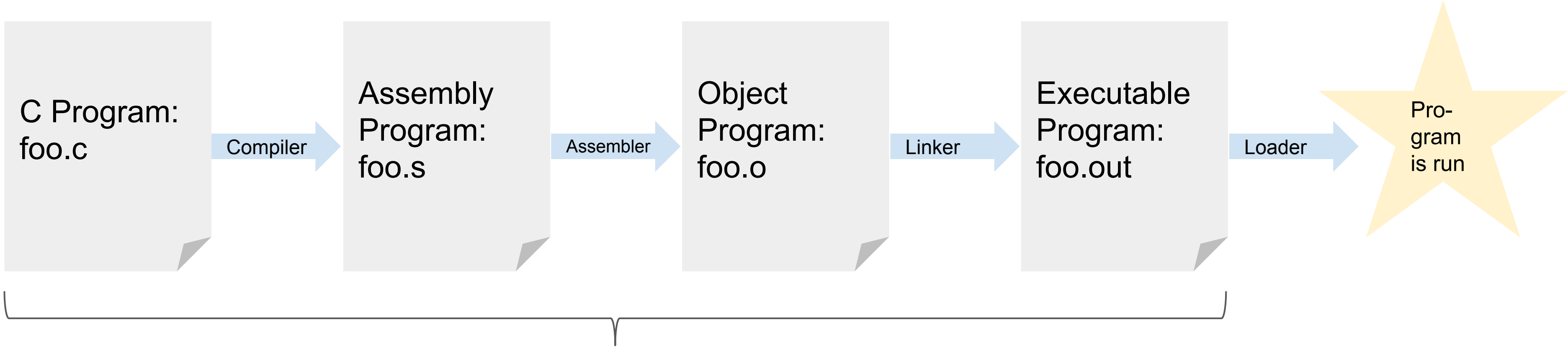
Today’s Goal

- Learn about the semantics behind representing floating point numbers
- Study the process of how human readable code is converted into machine instructions AKA Compiler, Assembler, Linker, Loader (CALL)

CALL

CALL Stack

Compiler, **A**ssembler, **L**inker, **L**oader



C Program: foo.c

Compiler

Assembly Program: foo.s

Assembler

Object Program: foo.o

Linker

Executable Program: foo.out

Loader

Program is run

All of this occurs whenever we did “`gcc foo.c`”

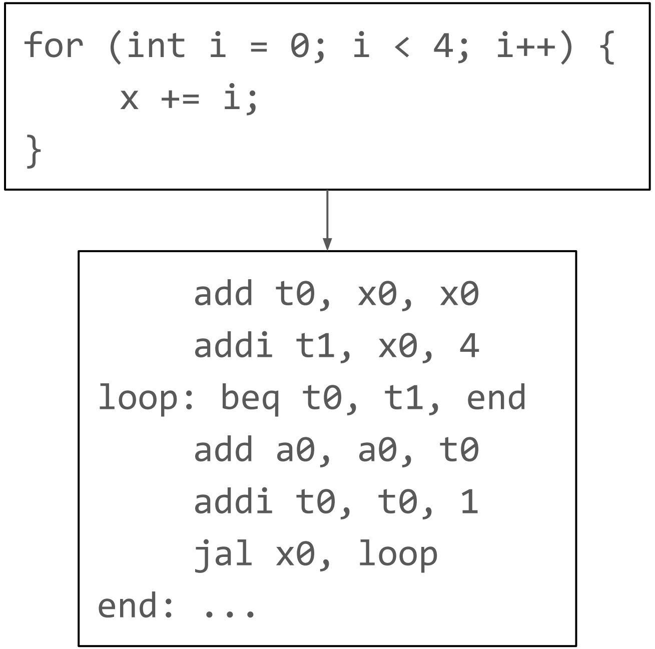
Compiler

Purpose: Convert C Code into RISC-V Instructions

Input: High Level Language Code (i.e. foo.c)

Output: Assembly Language Code (i.e. foo.s)

- Output may contain pseudo-instructions (understood by assembler, not machine)
    - I.e. `j label => jal x0 label`
- Multiple Responsibilities:
    - Lexer: Turn character input into tokens
    - Parser: Convert tokens into Abstract Syntax Tree (Think Scheme / Lisp!), identify program structure
    - Semantic Analysis (Compile Time Errors)
    - Code Optimization (Branches + Jumps)



```
for (int i = 0; i < 4; i++) {
    x += i;
}
```

```
      add t0, x0, x0
      addi t1, x0, 4
loop: beq t0, t1, end
      add a0, a0, t0
      addi t0, t0, 1
      jal x0, loop
end: ...
```

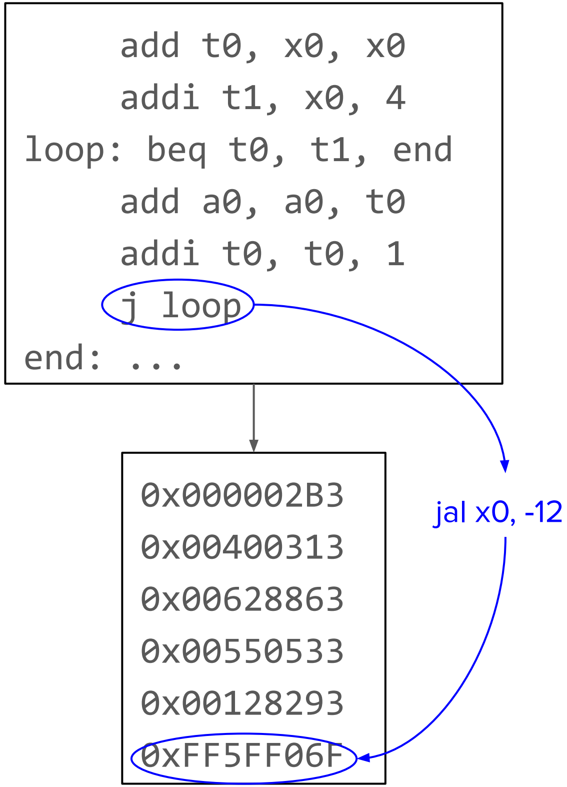
Assembler

Purpose: Convert RISC-V into Object (Binary) Code

Input: Assembly Language Code (i.e. foo.s)

Output: Object Code + Information Tables (i.e. foo.o)

- Reads + Uses Directives (i.e. .text, .data) to translate modes into numerical equivalents.
- May required multiple passes to build symbol + relocation tables, then generate the code.
    - Symbol: List of “items” like labels and .data stuff in this file that is usable by other files
    - Relocation: list of “items” whose address this file needs like absolute labels, static stuff
- Convert Pseudo-Instructions into counterparts



```
      add t0, x0, x0
      addi t1, x0, 4
loop: beq t0, t1, end
      add a0, a0, t0
      addi t0, t0, 1
      j loop
end: ...
```

```
0x000002B3
0x00400313
0x00628863
0x00550533
0x00128293
0xFF5FF06F
```

Assembler

The assembler outputs several files, including…

- **Header**: Details size + location of other sections
- **Data**: Binary representation of data in source file
- **Code**: Machine code for instructions of source file
- **Relocation Table**: List of unresolved lines of code a linker handles
- **Symbol Table**: List of labels + data from one source file that can be referenced by other files
- **Debugging**: Information for debugging programs (i.e. cgdb, Valgrind)

Assembler

2-Pass Mechanism

Consider the following instructions:

```
    jmp later
    ...
later: ...
```

What if we tried assembling in 1 pass?

- Forward Reference (Label used later)
- Can’t reliably calculate jump distance
- Labels could left unresolved!

Solution - 2 Passes!

Pass 1

- Assign address to each statement
- Store Symbolic Labels’ addresses (Symbol Table!)
- Process assembler directives

Pass 2

- Generate Machine Code
- Use Symbol Table to determine jump distances
- Generate efficient instructions

Assembling: Two Passes

```
     add t0, x0, x0             Addr = 0
     addi t1, x0, 4             Addr = 4
loop: beq t0, t1, end           Addr = 8
     add a0, a0, t0             Addr = 12
     jal ra square              Addr = 16
     jal ra printf              Addr = 20
     addi t0, t0, 1             Addr = 24
     jal x0, loop               Addr = 28
end: ...                        Addr = 32
                                ...
square: mul a0, a0, a0          Addr = 240
     jr ra                      Addr = 244
```

This jump doesn't need to be resolved: it's to a register!

Pass #1

loop	Addr = 8
end	Addr = 32
square	Addr = 240

Goes in **symbol table**

Resolved:

- jal x0, loop

Unresolved:

- beq t0, t1, end
- jal ra square
- jal ra printf

Pass #2

Resolved:

- jal x0, loop
- beq t0, t1, end
- jal ra square

Unresolved:

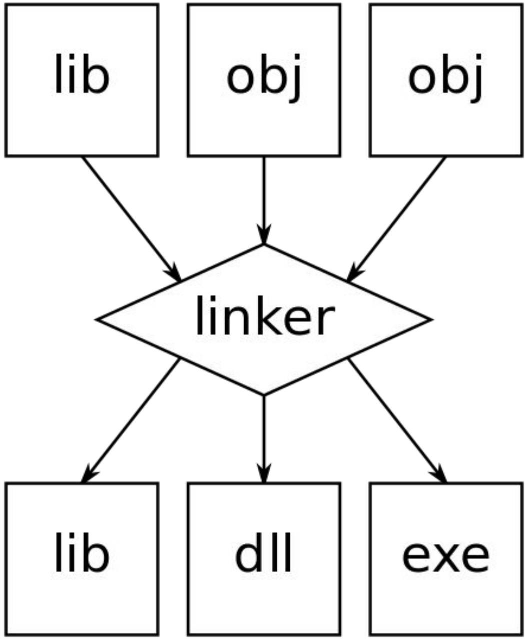
- jal ra printf

Goes in **relocation table**

Linker

Purpose: “Links” together object files into single executable file
Input: Object Code + Information Tables (i.e. foo.o)
Output: Executable File (i.e. a.out)

- Links text, then data segments together
- Enables separate compilation of files! Changing one file doesn’t require re-compiling the program.
- Performs 2 Major Tasks
  - Symbol Resolution: Map each reference to one definition. Go through relocation table and fill in absolute addresses
  - Relocation: Relocate Code + Data into output executable



Loader

Purpose: Load + Run the program
Input: Executable File (i.e. a.out)
Output: Program is Run

- Loads the file from disk into memory
    - Copy instructions + data into address space
    - Copy arguments to stack
    - Initialize Machine Registers
- Passes control to loaded program code after loading is complete.
- Nowadays, OS serves as the loader

Terminal Commands:

`gcc -g -o hello hello.c` ← gcc: **compiling**, **assembling**, **linking**

`./hello students` ← OS: **loading**

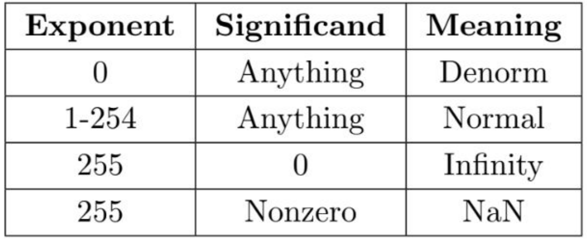
Floating Point

Floating Point

Motivation: Represent Decimal, Super Large, Super Small, and Special Numbers

Solution: Binary Representation + Delegate Bits to 3 Different Fields

- Sign (1 Bit): 0 - Positive, 1 - Negative
- Exponent (8 Bits): Remember to add the bias term!
- Significand / Mantissa (23 Bits): Interpret as Unsigned, stores fraction



Exponent	**Significand**	**Meaning**
0	Anything	Denorm
1-254	Anything	Normal
255	0	Infinity
255	Nonzero	NaN

$$\textbf{Value} = (-1)^{Sign} * 2^{Exp-Bias} * 1.\textbf{significand}_2$$

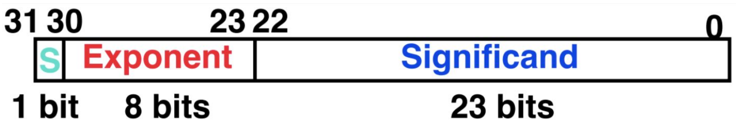
For denormalized floats:

$$\textbf{Value} = (-1)^{Sign} * 2^{Exp-Bias+1} * 0.\textbf{significand}_2$$

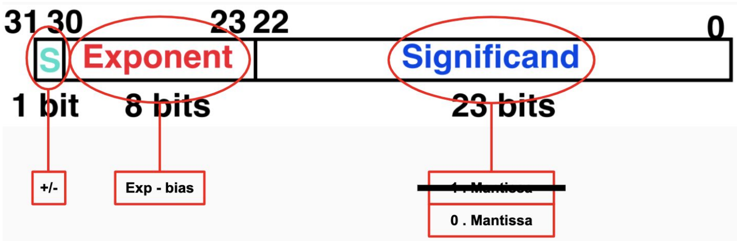
1 bit	8 bits	23 bits
Sign	Exponent	Significand/Mantissa

Floating Point Visualized

Regular



Denormalized



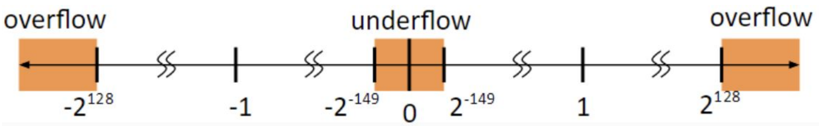
Smallest number possible

- No Denormalization: 2^{-126}
- With denormalization: $(0+2^{-23})*2^{-126} = 2^{-149}$

Special Numbers

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

Over + Underflow


overflow
underflow
overflow
-2^{128}
-1
-2^{-149}
0
2^{-149}
1
2^{128}