CS61C Discussion 4 – RISC-V Addressing, CALL, and Floating Point

1 RISC-V Addressing

1.1 Overview

- We have several **addressing modes** to access memory (immediate not listed):
  (a) **Base displacement addressing:** Adds an immediate to a register value to create a memory address (used for lw, lb, sw, sb)
  (b) **PC-relative addressing:** Uses the PC and adds the immediate value of the instruction (multiplied by 2) to create an address (used by branch and jump instructions)
  (c) **Register Addressing:** Uses the value in a register as a memory address (jr)

1.2 Exercises

1. What is range of 32-bit instructions that can be reached from the current PC using a branch instruction?

2. What is the range of 32-bit instructions that can be reached from the current PC using a jump instruction?

3. Given the following RISC-V code (and instruction addresses), fill in the blank fields for the following instructions (you'll need your RISC-V green card!).

```
0x002cff00: loop: add t1, t2, t0          |_______|_______|_______|_______|_______|__0x33__|
0x002cff04:       jal ra, foo             |_____________________|_________________|__0x6F__|
0x002cff08:       bne t1, zero, loop      |_______|_______|_______|_______|_______|__0x63__|
...
0x002cff2c: foo:  jr ra                 ra=__________________
```

2 Floating Point

2.1 Overview

The IEEE standard defines a binary representation for floating point values using three fields:

- The *sign* determines the sign of the number (0 for positive, 1 for negative).
- The *exponent* is in **biased notation** with a bias of 127.
- The *significant* is akin to unsigned, but used to store a fraction instead of an integer.

The image below shows the bit breakdown for the single precision (32-bit) representation:

Sign	Exponent	Significand
1 bit	8 bits	23 bits

How a float is interpreted depends on the values in the exponent and significand fields. To keep track of the requirements for each case, a table is provided below:

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

Normal Floats: $(-1)^{Sign} * 2^{(Exponent-Bias)} * 1.\text{significand}_2$
Denormalized Floats: $(-1)^{Sign} * 2^{(Exponent-Bias+1)} * 0.\text{significand}_2$

2.2 Exercises

1. How many zeros can be represented using a float?
2. What is the largest finite positive value that can be stored using a single precision float?
3. What is the smallest positive value that can be stored using a single precision float?
4. What is the smallest positive normalized value that can be stored using a single precision float?
5. Convert the following between decimal and binary:
   0x00000000 8.25 0x00000F00 39.5625 0xFF94BEEF -∞

2.3 Representation

Not every number can be represented perfectly using floating point. For example, $\frac{1}{3}$ can only be approximated and thus must be rounded in any attempt to represent it. For this question, we will only look at positive numbers.

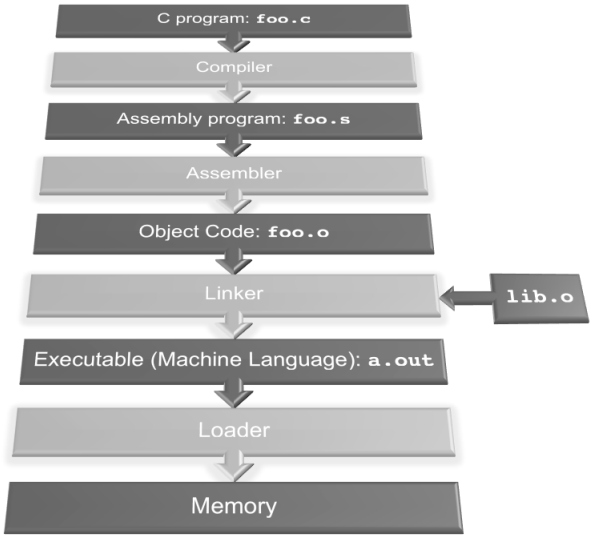
1. What is the next smallest number larger than 2 that can be represented completely? What is the next smallest number larger than 4 that can be represented completely?
2. Define stepsize to be the distance between some value x and the smallest value larger than x that can be completely represented. What is the step size for 2? 4?
3. Now let's see if we can generalize the stepsize for normalized numbers (we can do so for denorms as well, but we won't in this question). If we are given a normalized number that is not the largest representable normalized number with exponent value x and with significand value y, what is the stepsize at that value? Hint: There are 23 significand bits.

4. Now let's apply this technique. What is the largest odd number that we can represent? Part 3 should be very useful in finding this answer.

3 Compile, Assemble, Link, Load, and Go!

3.1 Overview

High level code cannot run on its own; it must pass through multiple stages of preparation before execution begins. We call this process CALL and is shown below.



3.2 Exercises

1. What is the Stored Program concept and what does it enable us to do?

2. How many passes through the code does the Assembler have to make? Why?

3. What are the different parts of the object files output by the Assembler?

4. Which step in CALL resolves relative addressing? Absolute addressing?

5. What does RISC stand for? How is this related to pseudoinstructions?

4 Writing RISC-V Functions

1. Write a function `double` in RISC-V that, when given an integer x, returns $2x$.

2. Write a function `power` in RISC-V that takes in two numbers x and n, and returns x^n. You may assume that $n \geq 0$ and that multiplication will always result in a 32-bit number.

3. Write a function `sumSquare` in RISC-V that, when given an integer `n`, returns the summation below. If `n` is not positive, then the function returns 0.

   $$n^2 + (n-1)^2 + (n-2)^2 + \ldots + 1^2$$

   For this problem, you are given a RISC-V function called `square` that takes in an integer and returns its square. Implement `sumSquare` using `square` as a subroutine.