CS 61C Fall 2019 — Floating Point, RISC-V Intro

Discussion 3: September 16, 2019

1 Floating Point

The IEEE 754 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**. For instance, the bias is 127 ($2^{8-1}-1$) for single-precision floating point numbers.
- The *significand* or *mantissa* is akin to unsigned integers, but used to store a fraction instead of an integer.

The below table shows the bit breakdown for the single precision (32-bit) representation. The leftmost bit is the MSB and the rightmost bit is the LSB.

1	8	23
Sign	Exponent	Mantissa/Significand/Fraction

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

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

Note that in the above table, our exponent has values from 0 to 255. When translating between binary and decimal floating point values, we must remember that there is a bias for the exponent.

1.1 How many zeroes can be represented using a float?

2

1.2 What is the largest finite positive value that can be stored using a single precision float?

`0x7F7FFFFF` $= (1 + (1 - 2^{-23})) * 2^{127}$

The mantissa for the largest value will be 23 1's. This corresponds to a value of

$$.11\ldots1 = 2^{-1} + 2^{-2} + \cdots + 2^{-23} = 2^{-23}(2^{22} + 2^{21} + \cdots + 1)$$

Here, we apply the formula that $\sum_{i=0}^{n-1} 2^i = 2^n - 1$, so we have that the mantissa is

$$2^{-23}(2^{22} + 2^{21} + \cdots + 1) = 2^{-23}(2^{23} - 1) = 1 - 2^{-23}$$

We have $1 + (1 - 2^{-23})$ since we this is a normalized number and thus has a 1 to the left of the decimal point.

1.3 What is the smallest positive value that can be stored using a single precision float?

`0x00000001` $= 2^{-23} * 2^{-126}$

1.4 What is the smallest positive normalized value that can be stored using a single precision float?

`0x00800000` $= 2^{-126}$

1.5 Cover the following single-precision floating point numbers from binary to decimal or from decimal to binary. You may leave your answer as an expression.

- 0x00000000

  0

- 8.25

  0x41040000

- 0x00000F00

  $(2^{-12} + 2^{-13} + 2^{-14} + 2^{-15}) * 2^{-126}$

- 39.5625

  0x421E4000

- 0xFF94BEEF

  NaN

- -∞

  0xFF800000

2 More Floating Point 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.

2.1 What is the next smallest number larger than 2 that can be represented completely?

For this question, you increment the number by the smallest amount possible. This is the same as incrementing the significand by 1 at the rightmost location.
$(1 + 2^{-23}) * 2 = 2 + 2^{-22}$

2.2 What is the next smallest number larger than 4 that can be represented completely?

For this question, you increment the number by the smallest amount possible. This is the same as incrementing the significand by 1 at the rightmost location.
$(1 + 2^{-23}) * 4 = 4 + 2^{-21}$

2.3 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?

This would be the amount added in part 1. This gives 2^{-22} and 2^{-21}.

2.4 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.

Here we need to generalize the solution we got in 1 and 2. However, this is the same approach just increment the signifcand by the 1.
$curr\_number = 2^{x-127} + 2^{x-127} * y$
$next\_number = 2^{x-127} + 2^{x-127} * y + 2^{x-127} * 2^{-23}$
$stepsize = next\_number - curr\_number = 2^{x-150}$

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

To find the largest odd number we can represent, we want to find when odd numbers will stop appearing. This will be with step size of 2.

As a result, plugging into Part 4: $2 = 2^{x-150} \rightarrow x = 151$
This means the number before $2^{151-127}$ was a distance of 1 (it is the first value whose stepsize is 2) and no number after will be odd. Thus, the odd number is simply subtracting the previous step size of 1.This gives,
$2^{24} - 1$

3 RISC-V: A Rundown

RISC-V is an assembly language, which is comprised of simple instructions that each do a single task such as addition or storing a chunk of data to memory.

For example, on the left is a line of C code and on the right is a chunk of RISC-V code that accomplishes the same thing.

```
                                    // x -> s0, &y -> s1
int x = 5, y[2];                    addi s0, x0, 5
y[0] = x;                           sw   s0, 0(s1)
y[1] = x * x;                       mul  t0, s0, s0
                                    sw   t0, 4(s1)
```

3.1 Can you figure out what each line in the RISC-V code is doing?

`addi s0, x0, 5` evaluates to `x = 5`. `sw s0, 0(s1)` evaluates to `y[0] = x`. `mul t0, s0, s0` calculates `x * x`. `sw t0, 4(s1)` evaluates to `y[1] = x * x`.

4 Registers

In RISC-V, we have two methods of storing data: main memory and registers. Registers are much faster than using main memory, but are very limited in space (32 bits each). Note that you should ALWAYS use the named registers (e.g. `s0` rather than `x8`).

Register(s)	Alt.	Description
x0	zero	The zero register, always zero
x1	ra	The return address register, stores where functions should return
x2	sp	The stack pointer, where the stack ends
x5-x7, x28-x31	t0-t6	The temporary registers
x8-x9, x18-x27	s0-s11	The saved registers
x10-x17	a0-a7	The argument registers, a0-a1 are also return value

4.1 Can you convert each instruction's registers to the other form?

```
add s0, zero, a1     -->          add x8, x0, x11
or  x18, x1, x30     -->          or  s2, ra, t5
```

Note that you should ALWAYS use the named registers (e.g. `s0` rather than `x8`).

5 Basic Instructions

For your reference, here are some of the basic instructions for arithmetic operations and dealing with memory (Note: ARG1 is argument register 1, ARG2 is argument register 2, and DR is destination register):

[inst]	[destination register] [argument register 1] [argument register 2]
add	Adds the two argument registers and stores in destination register
xor	Exclusive or's the two argument registers and stores in destination register
mul	Multiplies the two argument registers and stores in destination register
sll	Logical left shifts ARG1 by ARG2 and stores in DR
srl	Logical right shifts ARG1 by ARG2 and stores in DR
sra	Arithmetic right shifts ARG1 by ARG2 and stores in DR
slt/u	If ARG1 < ARG2, stores 1 in DR, otherwise stores 0, u does unsigned comparison

[inst]	[register] [offset]([register containing base address])
sw	Stores the contents of the register to the address+offset in memory
lw	Takes the contents of address+offset in memory and stores in the register

[inst]	[argument register 1] [argument register 2] [label]
beq	If ARG1 == ARG2, moves to label
bne	If ARG1 != ARG2, moves to label

[inst]	[destination register] [label]
jal	Stores the next instruction's address into DR and moves to label

You may also see that there is an "i" at the end of certain instructions, such as addi, slli, etc. This means that ARG2 becomes an "immediate" or an integer instead of using a register. There are also immediates in some other instructions such as **sw** and **lw**. NOTE: The size of an immediate in any given instruction depends on what type of instruction it is (more on this soon!).

5.1 Assume we have an array in memory that contains `int* arr = {1,2,3,4,5,6,0}`. Let register `s0` hold the address of the zeroth element in arr. You may assume integers are four-bytes and our values are word-aligned. What do the snippets of RISC-V code do? Assume that all the instructions are run one after the other in the same context.

```
a) lw   t0, 12(s0)        -->       Sets t0 equal to arr[3]

b) slli t1, t0, 2
   add  t2, s0, t1
   lw   t3, 0(t2)         -->       Increments arr[t0] by 1
   addi t3, t3, 1
   sw   t3, 0(t2)

c) lw   t0, 0(s0)
   xori t0, t0, 0xFFF     -->       Sets t0 to -1 * arr[0]
   addi t0, t0, 1
```

6 C to RISC-V

6.1 Translate between the C and RISC-V verbatim

C	RISC-V

`// s0 -> a, s1 -> b`
`// s2 -> c, s3 -> z`
`int a = 4, b = 5, c = 6, z;`
`z = a + b + c + 10;`
	`addi s0, x0, 4`
	`addi s1, x0, 5`
	`addi s2, x0, 6`
	`add  s3, s0, s1`
	`add  s3, s3, s2`
	`addi s3, s3, 10`

`// s0 -> int * p = intArr;`
`// s1 -> a;`
`*p = 0;`
`int a = 2;`
`p[1] = p[a] = a;`
	`sw   x0, 0(s0)`
	`addi s1, x0, 2`
	`sw   s1, 4(s0)`
	`slli t0, s1, 2`
	`add  t0, t0, s0`
	`sw   s1, 0(t0)`

`// s0 -> a, s1 -> b`
`int a = 5, b = 10;`
`if(a + a == b) {`
`    a = 0;`
`} else {`
`    b = a - 1;`
`}`
	`    addi s0, x0, 5`
	`    addi s1, x0, 10`
	`    add  t0, s0, s0`
	`    bne  t0, s1, else`
	`    xor  s0, x0, x0`
	`    jal  x0,  exit`
	`else:`
	`    addi s1, s0, -1`
	`exit:`

`// computes s1 = 2^30`
`s1 = 1;`
`for(s0=0;s0<30;s++) {`
`    s1 *= 2;`
`}`
	`    addi s0, x0, 0`
	`    addi s1, x0, 1`
	`    addi t0, x0, 30`
	`loop:`
	`    beq  s0, t0, exit`
	`    add  s1, s1, s1`
	`    addi s0, s0, 1`
	`    jal  x0, loop`
	`exit:`

```
// s0 -> n, s1 -> sum
// assume n > 0 to start
for(int sum = 0; n > 0; n--) {
  sum += n;
}
```

```
    addi s1, x0, 0
loop:
    beq  s0, x0, exit
    add  s1, s1, s0
    add  s0, s0, -1
    jal  x0, loop
exit:
```