CS 61C — Floating Point, RISC-V

Fall 2024 — Discussion 3

1 Pre-Check

1.1 The idea of floating point is to use the ability to move the radix (decimal) point wherever to represent a large range of real numbers as exact as possible.

1.2 Floating Point and Two's Complement can represent the same total amount of numbers (any reals, integer, etc.) given the same number of bits.

1.3 The distance between floating point numbers increases as the absolute value of the numbers increase.

1.4 Floating Point addition is associative.

1.5 Let `a0` point to the start of an array `x`. `lw s0, 4(a0)` will always load `x[1]` into `s0`.

1.6 Assuming no compiler or operating system protections, it is possible to have the code jump to data stored at `0(a0)` (offset 0 from the value in register `a0`) and execute instructions from there.

1.7 `jalr` is a shorthand expression for a `jal` that jumps to the specified label and does not store a return address anywhere.

2 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 which comes from -($2^{8-1} - 1$) 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:
$\textbf{Value} = (-1)^{\textbf{Sign}} * 2^{\textbf{Exp+Bias}} * 1.\textbf{significand}_2$
For denormalized floats:
$\textbf{Value} = (-1)^{\textbf{Sign}} * 2^{\textbf{Exp+Bias}+1} * 0.\textbf{significand}_2$

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.

2.1 Convert the following single-precision floating point numbers from hexadecimal to decimal or from decimal to hexadecimal. You may leave your answer as an expression.

- 0x00000000
- 8.25
- 0x00000F00
- 39.5625

- 0xFF94BEEF
- -∞
- 1/3

3 More Floating Point Representation

As we saw above, not every number can be represented perfectly using floating point. For this question, we will only look at positive numbers.

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

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

3.3 What is the largest odd number that we can represent? Hint: At what power can we only represent even numbers?

4 Instructions

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, RISC-V code accomplishes the same task as the C code, on the right, with its streamlined instructions.

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

For your reference, here are some of the basic instructions for arithmetic/bitwise operations and memory access (Note: rs1 is argument register 1, rs2 is argument register 2, and rd 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 rs1 by rs2 and stores in rd
srl	Logical right shifts rs1 by rs2 and stores in rd
sra	Arithmetic right shifts rs1 by rs2 and stores in rd
slt/u	If rs1 < rs1, stores 1 in rd, 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 rs1 == rs2, moves to label
bne	If rs1 != rs2, moves to label

[inst]	[destination register] [label]
jal	Stores the next instruction's address into rd 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 rs2 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 that the size (maximum number of bits) of an immediate in any given instruction depends on what type of instruction it is (more on this soon!).

4.1 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 element at index 0 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)       -->

b) sw   t0, 16(s0)       -->

c) slli t1, t0, 2
   add  t2, s0, t1
   lw   t3, 0(t2)        -->
   addi t3, t3, 1
   sw   t3, 0(t2)

d) lw   t0, 0(s0)
   xori t0, t0, 0xFFF    -->
   addi t0, t0, 1
```

4.2 Assume that `s0` and `s1` contain signed integers. Without any pseudoinstructions, how can we branch on the following conditions to jump to some LABEL?

`s0` $<$ `s1`   `s0` $\neq$ `s1`   `s0` $\leq$ `s1`   `s0` $>$ `s1`

5 Memory Access

Using the given instructions and the sample memory array, what will happen when the RISC-V code is executed? For load instructions (`lw, lb, lh`), write out what each register will store. For store instructions (`sw, sh, sb`), update the memory array accordingly. Recall that RISC-V is little-endian and byte addressable.

5.1

```
li t0 0x00FF0000
lw t1 0(t0)
addi t0 t0 4
lh t2 2(t0)
lw s0 0(t1)
lb s1 3(t2)
```

0xFFFFFFFF	
	...
0x00FF0004	0x000C561C
0x00FF0000	36
	...
0x00000036	0xFDFDFDFD
0x00000024	0xDEADB33F
	...
0x0000000C	0xC5161C00
	...
0x00000000	

What value does each register hold after the code is executed?

5.2

```
li t0 0xABADCAFE
li t1 0xF9120504
li t2 0xBEEFCACE
sw t0 0(t1)
addi t1 t1 4
addi t0 t0 4
sh t1 2(t0)
sb t2 1(t2)
sb t2 3(t1)
sb t2 3(t0)
```

0xFFFFFFFF	
0xF9120504	
0xABADCAFE	
0x00000004	
0x00000000	0x00000000

Update the memory array with its new values after the code is executed. Some memory addresses may not have been labeled for you yet.

6 Lost in Translation

6.1 Translate between the C and RISC-V verbatim.

<table>
<tr><th>C</th><th>RISC-V</th></tr>
<tr><td><pre>
// s0 -> a, s1 -> b
// s2 -> c, s3 -> z
int a = 4, b = 5, c = 6, z;
z = a + b + c + 10;
</pre></td><td></td></tr>
<tr><td><pre>
// s0 -> int * p = intArr;
// s1 -> a;
*p = 0;
int a = 2;
p[1] = p[a] = a;
</pre></td><td></td></tr>
<tr><td><pre>
// s0 -> a, s1 -> b
int a = 5, b = 10;
if(a + a == b) {
    a = 0;
} else {
    b = a - 1;
}
</pre></td><td></td></tr>
<tr><td></td><td><pre>
    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:
</pre></td></tr>
<tr><td><pre>
// s0 -> n, s1 -> sum
// assume n > 0 to start
for(int sum = 0; n > 0; n--) {
  sum += n;
}
</pre></td><td></td></tr>
</table>