CS61C Precheck: Floating Point, RISC-V
Spring 2025 Discussion 3

1 Discussion 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.

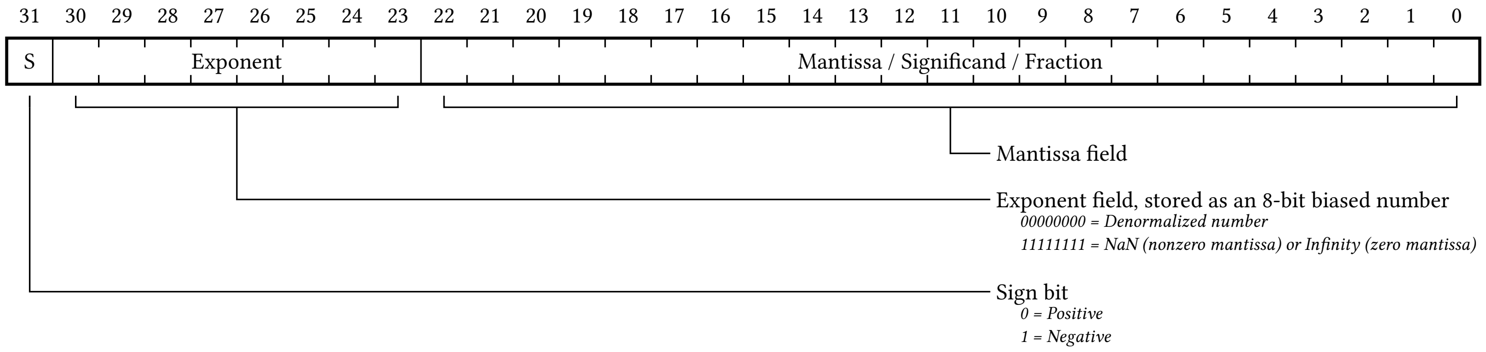
1.7 `jalr` is a shorthand expression for a `jal` that jumps to the specified label and does not store a return address anywhere.

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 and refers to the bits to the right of the leading "`1`" when normalized. For example, the significand of `1.010011` is `010011`.

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


31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
S
Exponent
Mantissa / Significand / Fraction
Mantissa field
Exponent field, stored as an 8-bit biased number
00000000 = Denormalized number
11111111 = NaN (nonzero mantissa) or Infinity (zero mantissa)
Sign bit
0 = Positive
1 = Negative

For normalized floats:

$$\mathbf{Value} = (-1)^{\text{Sign}} \times 2^{\text{Exp+Bias}} \times 1.\text{Significand}_2)$$

For denormalized floats:

$$\mathbf{Value} = (-1)^{\text{Sign}} \times 2^{\text{Exp+Bias+1}} \times 0.\text{Significand}_2)$$

Exponent (Pre-bias)	**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.

3 RISC-V Instructions

RISC-V is an assembly language composed of simple instructions that each perform a single task such as addition of two numbers or storing data to memory. Below is a comparison between RISC-V code and its equivalent C code:

```
// x in s0, &y in s1
addi s0, x0, 5          int x = 5;
sw s0, 0(s1)            y[0] = x;
mul t0, s0, s0
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, which can also be found on the 61C Reference Card.

The below are abbreviations that will be used in the table:

- `rs1`: Argument register 1
- `rs2`: Argument register 2
- `rd`: Destination register
- `imm`: Immediate value (integer literal constant)
- `R[register]`: Value contained in `register`
- `inst`: One of the instructions in the table

Register-to-register operations (R-type): `inst rd rs1 rs2`	
`add`	Adds `R[rs1]` and `R[rs2]` and stores the result in `rd`
`xor`	Exclusive ORs `R[rs1]` and `R[rs2]` and stores the result in `rd`
`mul`	Multiplies `R[rs1]` by `R[rs2]` and stores the result in `rd`
`sll`	Logical left shifts `R[rs1]` by `R[rs2]` and stores the result in `rd`
`srl`	Logical right shifts `R[rs1]` by `R[rs2]` and stores the result in `rd`
`sra`	Arithmetic right shifts `R[rs1]` by `R[rs2]` and stores the result in `rd`
`slt(u)`	If `R[rs1] < R[rs2]`, puts 1 in `rd`, else puts 0 (`u` compares unsigned)

Memory operations	
`sw rs2 rs1(imm)`	Stores `R[rs2]` *to* the address `R[rs1] + imm` in memory
`lw rd rs1(imm)`	Loads address `R[rs1] + imm` *from* memory into `rs2`

Branch operations (B-type): `inst rs1 rs2 label`	
`bne`	If `rs1 != rs2`, jump to `label`
`beq`	If `rs1 == rs2`, jump to `label`

Jump operations (J-type): `inst rd label`	
`jal`	Stores the next instruction's address into `rd` and jumps to `label`

A RISC-V "immediate" is any numeric constant. For example, `addi t0, t0, 20`, `sw a4, -8(sp)`, and `lw a1, 0x44(t2)` have immediates `20`, `-8`, and `0x44` respectively. 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!).

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 immediates in instructions which use an offset such as `sw` and `lw`. When coding in RISC-V, use the 61C reference card for the details of each instruction (the reference card is your friend)!