

1 Pre-Check

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

- 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.

True. Floating point:

- Provides support for a wide range of values. (Both very small and very large)
- Helps programmers deal with errors in real arithmetic because floating point can represent $+\infty$, $-\infty$, NaN (Not a number)
- Keeps high precision. Recall that precision is a count of the number of bits in a computer word used to represent a value. IEEE 754 allocates a majority of bits for the significand, allowing for the use of a combination of negative powers of two to represent fractions.

- 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.

False. Floating Point can represent infinities as well as NaNs, so the total amount of representable numbers is lower than Two's Complement, where every bit combination maps to a unique integer value.

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

True. The uneven spacing is due to the exponent representation of floating point numbers. There are a fixed number of bits in the significand. In IEEE 32 bit storage there are 23 bits for the significand, which means the LSB is 2^{-22} times the MSB. If the exponent is zero (after allowing for the offset) the difference between two neighboring floats will be 2^{-22} . If the exponent is 8, the difference between two neighboring floats will be 2^{-14} because the mantissa is multiplied by 2^8 . Limited precision makes binary floating-point numbers discontinuous; there are gaps between them.

- 1.4 Floating Point addition is associative.

False. Because of rounding errors, you can find Big and Small numbers such that: $(\text{Small} + \text{Big}) + \text{Big} \neq \text{Small} + (\text{Big} + \text{Big})$

FP approximates results because it only has 23 bits for Significand.

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:

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

For denormalized floats:

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

| 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.

- $0x00000000$ $0x421E4000$
- 0 $0xFF94BEEF$
- 8.25 NaN
- $0x41040000$ $-\infty$
- $0x0000F00$ $0xFF800000$
- $(2^{-12} + 2^{-13} + 2^{-14} + 2^{-15}) * 2^{-126}$ $1/3$
- 39.5625 N/A — Impossible to actually represent, we can only approximate it

We'll go more into depth with converting 8.25 and $0x0000F00$. For the sake of brevity, the rest of the conversions can be done using the same process.

To convert 8.25 into binary, we first split up our 32b hexadecimal number into three parts. The sign is positive, so our sign bit -1^S will be 0. Then, we can solve for our significand. We know that our number will have a non-zero exponent, so we

will have a leading 1 for our mantissa. Splitting 8.25 into its integer and decimal portions, we can determine that 8 will be encoded in binary as 1000. and 0.25 will be .01 (the 1 corresponds to the 2^{-2} place), so by implying the MSB, our significand will be 00001000.. Finally, we can solve for the exponent. As our leading 1 is in the 2^3 place to encode 8, we must use the bias in reverse to find what exponent we encode in binary. 130 added with a bias of -127 results in 3, so our exponent is 0b10000010. Our final binary number concatenated is 0 100 0001 0 000 0100 0000 0000 0000 0000, or 0x41040000.

For 0x00000F00, splitting up the hexadecimal gives us a sign bit and exponent bit of 0, and a significand of 0b 000 0000 0000 1111 0000 0000. We now know that this will be some sort of denormalized positive number. We can find out the true exponent by adding the bias + 1 to get the actual exponent of -126. Then, we can evaluate the mantissa by inspecting the bits that are 1 to the right of the radix point, and finding the corresponding negative power of two. This results in the mantissa evaluated as $2^{-12} + 2^{-13} + 2^{-14} + 2^{-15}$. Combining these get the extremely small number $(-1)^0 * 2^{-126} * (2^{-12} + 2^{-13} + 2^{-14} + 2^{-15})$

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?

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.

$$(1 + 2^{-23}) * 2 = 2 + 2^{-22}$$

- 3.2 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.

$$(1 + 2^{-23}) * 4 = 4 + 2^{-21}$$

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

To find the largest odd number we can represent, we want to find when odd numbers will stop appearing. This will be when the LSB will have a step size of 2, subtracted by 1. After this number, only even numbers can be represented in floating point.

We can think of each binary digit in the significant as corresponding to a different power of 2 to get to a final sum. For example, 0b1011 can be evaluated as $2^3 + 2^1 + 2^0$, where the MSB is the 3rd bit and corresponds to 2^3 and the LSB is the 0th bit at 2^0 .

We want our LSB to correspond to 2^1 , so by counting the number of mantissa bits (23) and including the implicit 1, we get a total exponent of 24. The smallest number with this power would have a mantissa of 00.00, so after taking in account the implicit 1 and subtracting, this gives $2^{24} - 1$

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 is a snippet of C code and on the right is a chunk of RISC-V code that accomplishes the same thing.

```

int x = 5;           // x -> s0, &y -> s1
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)

```

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 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)` --> Sets `t0` equal to `arr[3]`

b) `sw t0, 16(s0)` --> Stores `t0` into `arr[4]`

c) `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)`

d) `lw t0, 0(s0)`
`xori t0, t0, 0xFFFF` --> Sets `t0` to `-1 * arr[0]`
`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 ≠ s1` `s0 ≤ s1` `s0 > s1`

`blt s0, s1, LABEL` `bne s0, s1, LABEL` `bge s1, s0, LABEL` `blt s1, s0, LABEL`

Note that RISC-V does not provide a `bgt` instruction because you can manipulate the `blt` instruction to get an equivalent result. Also note that the above solutions assume that `s0` and `s1` contained signed integers. If they are unsigned, then we would use the unsigned variants of the above commands (namely, `bltu`, `bgeu`).

5 Lost in Translation

5.1 Translate between the C and RISC-V verbatim.

| C | RISC-V |
|--|---|
| <pre>// s0 -> a, s1 -> b // s2 -> c, s3 -> z int a = 4, b = 5, c = 6, z; z = a + b + c + 10;</pre> | <pre>addi s0, x0, 4 addi s1, x0, 5 addi s2, x0, 6 add s3, s0, s1 add s3, s3, s2 addi s3, s3, 10</pre> |

| | |
|--|---|
| <pre>// s0 -> int * p = intArr; // s1 -> a; *p = 0; int a = 2; p[1] = p[a] = a;</pre> | <pre>sw x0, 0(s0) addi s1, x0, 2 sw s1, 4(s0) slli t0, s1, 2 add t0, t0, s0 sw s1, 0(t0)</pre> |
| <pre>// s0 -> a, s1 -> b int a = 5, b = 10; if(a + a == b) { a = 0; } else { b = a - 1; }</pre> | <pre>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:</pre> |
| <pre>// computes s1 = 2^30 // assume int s1, s0; was declared above s1 = 1; for(s0 = 0; s0 != 30; s0++) { s1 *= 2; }</pre> | <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> |
| <pre>// s0 -> n, s1 -> sum // assume n > 0 to start for(int sum = 0; n > 0; n--) { sum += n; }</pre> | <pre>addi s1, x0, 0 loop: beq s0, x0, exit add s1, s1, s0 add s0, s0, -1 jal x0, loop exit:</pre> |