CS61C Spring 2025 — Floating Point, RISC-V — Discussion 3

1 Floating Point

1.1 Convert the following single-precision floating point numbers from hexadecimal to decimal or from decimal to hexadecimal using the IEEE 754 Floating Point Standard. You may leave your answer as an expression.

a) `8.25`

Answer: `0x41040000`

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`.

b) `39.5625`

Answer: `0x421E4000`

Writing `39.5625` in binary results in the bits `100111.1001`. In normalized form, we get 1.001111001×2^5. We can find our floating point exponent with $2^5 = 2^{\text{exp + bias}} = 2^{\text{exp} - 127} \Rightarrow \text{exp} = 132 =$ `0b10000100`. From the normalized form, our signficand is `0b00111100100` and our sign bit is zero. Our final binary number is `0b0_10000100_00111100100000000000000` which is `0x421E4000`.

c) `0x00000F00`

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

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 $(-1)^0 * 2^{-126} * (2^{-12} + 2^{-13} + 2^{-14} + 2^{-15})$

d) `0x00000000`

Answer: `0x00000000`

To represent zero in floating point, the significand will be zero but we need to set the correct exponent. Recall that any non-zero exponent will indicate a *normalized* float which means the significand has a leading 1. Zero's binary representation does not have a leading 1, so we need an exponent of zero to indicate a *denormalized* float which implies a leading zero in front of the significand. Fortunately, this nicely works out to represent zero as `0x00000000`.

e) `0xFF94BEEF`

Answer: NaN

Certain exponent fields are reserved for representing special values. Floating point representations with exponents of 255 and a zero significand are reserved for ± ∞, and exponents of 255 with a nonzero significand are reserved for repres-ntations of NaN. Deconstructing the fields of this number gives an exponent of `0b11111111 = 255` and a nonzero significand which indicates that this represents NaN. Note that there are many possible ways to represent NaN.

f) ∞

Answer: `0x7F800000`

Certain exponent fields are reserved for representing special values. Floating point representations with exponents of 255 and a zero significand are reserved for ± ∞, and exponents of 255 with a nonzero significand are reserved for represntations of NaN. Because we need to represent positive infinity, we use a 0 for the sign bit, 255 for the exponent field, and a zero significand giving `0b01111111100000000000000000000000` or `0x7F800000` for positive ∞. Note that -∞ would be `0xFF800000`.

g) `1/3`

Answer: N/A - impossible to represent in single-precision floating point, we can only approximate it

2 More Floating Point

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

$\boxed{2.1}$ What is the next smallest number larger than 2 that can be represented completely?

Answer: $2 + 2^{-22}$

For this question, we start at the number 2 and increment the number by the smallest amount possible. This is the same as incrementing the significand of 2's floating point representation by 1 at the rightmost location, which adds 2^{-22}.

Normalized: $2 = 10.000... = 1.000...00 \times 2^1$

Increment: $2 = (1.000...00 + 2^{-23}) \times 2^1 = (1 + 2^{-23}) \times 2 = 2 + 2^{-22}$

$\boxed{2.2}$ What is the next smallest number larger than 4 that can be represented completely?

Answer: $4 + 2^{-21}$

Following the previous question, we increment the number 4 by the smallest amount possible. This is the same as incrementing the significand by 1 at the rightmost location.

Normalized: $4 = 100.000... = 1.000...00 \times 2^2$

Increment: $(1.000...00 + 2^{-23}) \times 2^2 = (1 + 2^{-23}) \times 4 = 4 + 2^{-21}$

$\boxed{2.3}$ What is the largest odd number that we can represent? Hint: at what power can we only represent even numbers?

Answer: $2^{24} - 1$

To find the largest odd number we can represent, we want to find when odd numbers will stop appearing. Because we are always multiplying the significand by a power of 2^x, we will only be able to represent even numbers when the exponent grows large enough. In particular, odd numbers will stop appearing when the significand's LSB has a step size (distance between each successive number) of 2, so the largest odd number will be the first number with 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 account of the implicit 1 and subtracting, this gives $2^{24} - 1$

3 RISC-V Instructions

$\boxed{3.1}$ 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 following 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)`

Answer: Sets `t0` equal to `arr[3]`

b) `sw t0, 16(s0)`

Answer: Stores `t0` into `arr[4]`

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

Answer: Increments `arr[t0]` by 1

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

Answer: Sets `t0` to `-1 * arr[0]`

4 RISC-V 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. For any unknown instructions, use the CS 61C reference card!

4.1

```
1 li t0 0x00FF0000
2 lw t1 0(t0)
3 addi t0 t0 4
4 lh t2 2(t0)
5 lw s0 0(t1)
6 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?

`t0`: `0x00FF0004`. Line 3 adds 4 to the initial address.

`t1`: `36`. Line 2 loads the 4-byte word from address `0x00FF0000`.

`t2`: `0xC`. Line 4 loads two bytes starting at the address `0x00FF0004 + 2 = 0x00FF0006`. This returns `0x000C`

`s0`: `0xDEADB33F`. Line 5 loads the word starting at address `36 = 0x24` which is `0xDEADB33F`.

`s1`: `0xFFFFFFC5`. Line 6 loads the MSB starting of the 4-byte word at address `0xC`. The value is `0xC5` which is sign-extended to `0xFFFFFFC5`.

4.2 Update the memory array with its new values after the code is executed. Assume each byte in the memory array is initialized to zero.

```
li t0 0xABADCAF8
li t1 0xF9120504
li t2 0xBEEFDAB0
sw t0 0(t1)
addi t0 t0 4
sh t1 2(t0)
sh t2 0(t0)
lw t3 0(t1)
sb t1 1(t3)
sb t2 3(t3)
```

0xFFFFFFFF	0x00000000
	...
0xF9120504	0xABADCAF8
	...
0xBEEFDAB0	0x00000000
	...
0xABADCAFC	0x0504DAB0
0xABADCAF8	0xB0000400
	...
0x00000000	0x00000000

5 Lost in Translation

5.1 Translate the code verbatim between C and RISC-V.

These code snippets are a great reference for writing RISC-V code!

C	**RISC-V**

`// s0 -> a`
`// s1 -> b`
`// s2 -> c`
`// s3 -> z`
`int a = 4, b = 5, c = 6;`
`int 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`

`// int *p = intArr;`
`// s0 -> p;`
`// 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;`
`int b = 10;`
`if (a + a == b) {`
`  a = 0;`
`} else {`
`  b = a - 1;`
`}`
	`start:`
	`  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`

C	**RISC-V**
	`exit:`
	`  ...`
`// Compute s1 = 2^30`	`start:`
`int s0 = 0;`	`  addi s0, x0, 0`
`int s1 = 1;`	`  addi s1, x0, 1`
`for (; s0 != 30; s0 += 1) {`	`loop:`
`  s1 *= 2;`	`  beq s0, t0, exit`
`}`	`  sll s1, s1, 1`
	`  addi s0, s0, 1`
	`  jal x0, loop`
	`exit:`
	`  ...`
`// s0 -> n`	`start:`
`// s1 -> sum`	`  addi s1, x0, 0`
`for (int sum = 0; n > 0; n--) {`	`loop:`
`  sum += n;`	`  beq s0, x0, exit`
`}`	`  add s1, s1, s0`
	`  addi s0, s0, -1`
	`  jal x0, loop`
	`exit:`
	`  ...`