Homework 1: floating point operators

Over HW1 and HW2, we will implement a couple floating point operators to get you familiar with PyRTL's basic functionalities. PyRTL doesn't have native floating point operators, which can in part be attributed to the complexity of floating point operations; since floating point operations are complex, implementing them strictly combinatorially would make the clock cycles unusably long. We'll cheat here and implement most of our operations in one cycle; the assignments are vastly simpler than real floating point operators.

In fact, floating point representations can get *much* more complex than what we will cover. NVIDIA has its own floating point representation specifications. Far more exotic representations have also been proposed, such as posits. Even the IEEE 754 specs contain specs that we won't bother implementing, such as rounding rules.
(see, https://docs.nvidia.com/cuda/floating-point/index.html, https://en.wikipedia.org/wiki/IEEE_754)

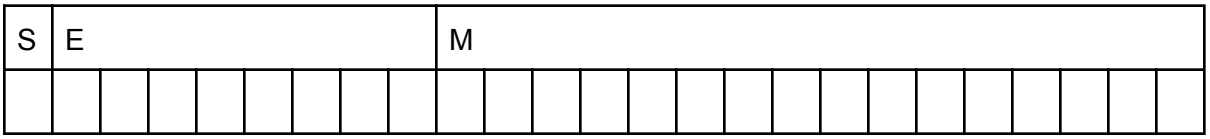
For each of the following 4 assignments, test benches have been provided for each of the required functionalities. You will need to finish part 1 before being able to run the test benches for parts 2-4. Each of the test benches can be ran like:

```
python3 adder_class.py
```

If your code works, it will simply complete. If it doesn't it will fail the assertion at the end of the file, which compares the circuit output with a python logical model (eg, basically the native floating point operations provided by Python) as well as spit out the test case that failed.
One massive benefit of a language like PyRTL is the simplified built-in simulator, which allows us to write native python test benches and run them without a confusing and error-prone process of transferring generated Verilog in between files.

1: to/from floating point format

The first assignment is simply to write a python function to convert python floating point numbers into a list of binary strings representing the floating point number in memory. You have probably learned about how IEEE 754 floating point numbers are formatted. To jog your memory, we'll briefly review them here:



IEEE 754 establishes that floating point numbers be stored in three contiguous binary numbers; the sign bit, s, the exponent bits, e, and the mantissa bits, m. The floating point number represented by these bits is:

$(-1 * s) * 1.m * 2^{bias-e}$

where $bias = 2^{bits\ in\ exponent-1}$
Note that the mantissa bits always start with 1, so the initial 1 is simply left out!
Throughout this assignment, we will use an 8 bit exponent and a 16 bit mantissa. In fact, IEEE 754 also specifies double-precision floats, but again, we won't worry about that.

Your job here is to write the python functions logicFloat_to_float and float_to_Logicfloat. In this assignment, we'll use a standard python format for representing the S/E/M bits. Each will be represented as a binary string, like so:

7.2271 => [sign, exp, mantissa] => ['0', '10000010', '11001110100010001100111001110000']

This doesn't have anything to do with PyRTL yet: however, writing some of the helper functions for testing also helps demonstrate one HUGE benefit of PyRTL: writing test benches is very easy!

For float_to_Logicfloat:
floatIn is the python float we want to convert into three binary strings
expLen is the number of bits in the exponent
manLen is the number of bits in the mantissa

2: shifter

Next, we will write a circuit that just shifts a floating point number to the right. This literally just subtracts from the exponent, so there shouldn't be too many lines of code.

3: multiplier

Floating point multiplication is a bit harder but the algorithm is still fairly simple.
-Add the exponents
-multiply the mantissas together
-if the mantissas' product is bigger than 1, you need to regularize it
-also, truncate the mantissa if required

Some points to watch out for:

-The product of two mantissas is 2m bits long. You'll need to truncate the last m bits.
-Consider the mantissas
1.000 * 1.000 = 1.000000
The regularization here is simply truncating the last 2 bits
However, for:
1.111 * 1.111 = 11.100001

The product is bigger than 1: you need to check for this case and regularize accordingly.
-look at the exponent bits: they are not stored as a positive or negative number. Instead, IEEE 754 dictates that they be stored as a bias minus a positive number. So, when you add exponents, you actually need to do:
$(bias - exp_1) + (bias - exp_2) - bias$
Keep in mind that adding the bias + bias will result in a number bigger than just expLen can store: you have to figure out the wire length here! Note that we won't expect you to use the minimal amount of wires in this project.

4: adder

Floating point addition is the trickiest of the 4 assignments in part 1, although very manageable. The process of adding floating points is:
-compare the exponents, keep the bigger one
-shift the smaller exponent's corresponding mantissa so that the exponent is equal to the bigger one
-add the mantissas
-check if the mantissa is greater than 1. Note that 1.111…+1.111…<100, so we can always just truncate the LSB and shift right one
-check if the mantissa is less than 1. Note that in addition, we might be adding negative numbers. In that case:
1.110101100110001
-1.110101100101010
—----------
0.000000000…
This actually requires knowing where the most significant digit with a '1' is.
Fortunately for us, PyRTL has this exactly functionally, only in reverse: it checks for the least significant 1, from the right to the left.
https://pyrtl.readthedocs.io/en/latest/rtllib.html#pyrtl.rtllib.muxes.prioritized_mux
The code can be found here. SImply copy it into the 'provided libraries' file and make the change to make it check from MSB -> LSB. Hint: it shouldn't be more than a couple characters.
https://github.com/UCSBarchlab/PyRTL/blob/a3d618d67621266f6f4209f68c8fe7321aff5f61/pyrtl/rtllib/muxes.py#L17

Some points to watch out for:

-Don't try to add and subtract things manually; PyRTL handles negation using two's complement, which should work just fine for mantissas.
-You'll need to check the wire length very carefully here! We've provided an example of the integers being added and the correct wire length.

Example of FP addition

rand_flt_a = 27
rand_flt_b = -5

				Represented mantissa						
27		1	.	1	0	1	1	0	0	0
-5		1	.	0	1	0	0	0	0	0
Add leading 1, shift lesser exp										
27			1	1	0	1	1	0	0	0
-5			0	0	1	0	1	0	0	0
Negate (bitflip, add 1)										
27			1	1	0	1	1	0	0	0
-5			1	1	0	1	1	0	0	0
Sign extend (0 for pos, 1 for neg)										
27	0	0	1	1	0	1	1	0	0	0
-5	1	1	1	1	0	1	1	0	0	0
Add, discard overflow (see note NOTE053023_1)										
<1>	0	0	1	0	1	1	0	0	0	0
check “extra” final bit for signage: Extra bit == 0, so the final result is gotten by chopping off the extra bit										
	0	0	1	0	1	1	0	0	0	0
check final bit for cut: since bit 1 is a 0, cut MSB and first 1 (always will be 1, see NOTE_053123)										
		0	1	0	1	1	0	0	0	0
Sanity check: 1.011_2 = 1.375 * pow(2,4) = 22										

-try this with a couple other values so you understand why we have to add two bits for the sign extend step.