Homework 2: Newton-Raphson division

Using the modules you have written previously, we will now write a Newton-Raphson divider. IEEE specifies a different method for division, but let's say we wanted to test out some new division algorithms on a new accelerator. PyRTL might help us make a divider module and quickly do some tests with it.

Newton Raphson division is based on Newton's method for finding roots. We can use newton's method for finding $\frac{1}{b}$, and then we can multiply this by a to get $\frac{a}{b}$. This means that we want a function for which the root is $\frac{1}{b}$, *but* we also need that function to have the property where $\frac{f(x)}{f'(x)}$ is a simple expression and can be calculated using only additions and multiplications. Thankfully, we don't need to figure this out ourselves: we will simply use the function provided to us by the wikipedia page on this method (see link below). Once we have $\frac{f(x)}{f'(x)}$, we simply calculate it and add it to x_0 a bunch of times.

One more detail needs to be taken care of: the initial guess. We assign a linear approximation for the guess, initially. This will help us get an accurate initial guess with the few simple operators we have. From the wikipedia article, we can see that all we need to do if write the error as a function of our desired value, $\frac{1}{b}$, and then set the derivative to 0. We're still left with a question in that we have *two* values to minimize. We won't go into detail about how to choose the values here. Finally, if b is allowed to be any real number, there is no good initial guess for the root we want. To fix this, we will regularize a and b to be between 0.5 and 1, which will limit the range of b and give us a good estimate for the initial value for $\frac{1}{b}$. Thankfully, division makes the regularizing easy: for any regularizing constant e;

$\frac{a}{b} = \frac{ae}{be}$

All you need to do is to calculate the regularizing constant for b and then multiply it to a as well.
https://en.wikipedia.org/wiki/Division_algorithm#Newton%E2%80%93Raphson_division

While it's important you understand what the algorithm does, the point isn't to teach you about the mathematics of the method. You can simply copy the pseudo code from the article if you wish.

1: python model of Newton-Raphson division

In the file nr_logical.py, you should first complete a python model of this algorithm. For each intermediate value along the way, you should print out its value: this will help immensely when you are debugging the actual divider! For example, in calculating

```
rec_guess + rec_guess*(1-(div*rec_guess))
```

You may want to print out a couple intermediate values, like

```
1-(div*rec_guess)
```

We don't provide a test bench here, nor should you need one. Just test on a couple values to make sure that the answers look about right. The important point is that you have a python model to compare intermediate values against.

You are encouraged to simply copy the steps given in the wikipedia article. And translate them into Python.

2: Newton-Raphson divider

We will finish by using the hardware we wrote for homework 1 to write a divider. We have provided a testbench that works in the same way the testbenches for part 1 did: please read this test bench and understand why it works. The native testing functionality for python is a huge benefit to PyRTL.

One notable difference here is that the divider is not placed into a class. We aren't expecting to turn this into a module, so the wires and registers can simply be declared in main.

This module will require you to write a simple state machine. We provide the states WAITING, MULTIPLYING, and DONE. The behavior should be as follows:

states
-when WAITING, ready_ab is 1 and valid_c is 0. When the input is valid (valid_ab==1), get the initial guess and set the state to MULTIPLYING. When valid_ab is 0, simply wait in this state and do not change anything. -when MULTIPLYING, iteratively add the incremental improvement to the guess. You should be able to exit if the problem is done early; otherwise, exit when max_iter is reached. You will need a counter for this state. When either exit condition is fulfilled, set the state to DONE. valid_c and ready_c should both be set to 0 -when DONE, assign float_c to the correct output value. As soon as ready_c is 1, we will assume that the consumer has read in the ready_c value and no longer needs it. Reset all registers you have used and set the state back to WAITING.

You'll notice a few limitations that will make writing this class a bit harder. We'll enumerate them here so you know what to look out for.

Some points to watch out for

-keep in mind that PyRTL requires you to declare a new context every time you want to write a new conditional. Eg;

```
with conditional_assignment:
```

```
    with a:
        r1.next |= i  # set when a is true
        with b:
            r2.next |= j  # set when a and b are true
    with c:
        r1.next |= k  # set when a is false and c is true
        r2.next |= k
```
(taken from the conditional.py file in PyRTL lib)

Note that the code under “with c” is only run when the prior condition, “with a” is false. If you have assignments that you want to carry out when some condition “d” is true, independent of whether or not “a” is true, you will need a new “with conditional_assignment” block. However, You cannot nest “with conditional_assignment” blocks. All this is to say that if you used any “with conditional_assignment” in your adder/multiplier/shifter, you need to get rid of them and replace them with selector functions, or you can write the divider using only selectors, but the state machine makes this much more difficult.

-You might have wondered previously why we added something to each wire name in the prior modules. This is because no new wire, regardless of which file/class it is in, can share a name with any other wire in any other file/class. This presents a very obvious problem when we have more than one instance of a module! This means that if you have a wire

```
self.expA = pyrtl.WireVector(self.expLen, 'expA'+nameTag)
```

in a module, and you instantiate the class with

```
addTestObj1 = addTestClass("addobj1",expLen,manLen)
```

You can debug this by inspecting the wire

```
sim.inspect("expAaddobj1")
```

-At this point, you’ve probably already learned that “|=” conditionally assigns a driver to a wire. However, you might have also used the PyRTL feature where you do not declare a write, and simply write

“newWireName = operand1 PyRTLOp operand2)”

PyRTL does a cool thing where it can infer that newWireName is a new wire, as well as automatically assign new wire length. THIS DOES NOT WORK IN CONDITIONALS! If you write a code block like:

```
with conditional_assignment:
    with a:
        wireX |= wire1 * 2
        newWireName = wire1 + wire2
    with c:
        wireX |= wire1 * 3
        newWireName = wire1 - wire2
```

wireX will be assigned according to the conditionals, meaning when a is true, wireX will be wire1*2. newWireName, however, will simply follow the last assignment!