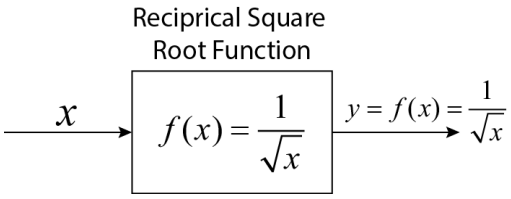
EELE 466
Lab 4
Due February 27, 2018

Creating a Hardware Reciprocal Square Root Function

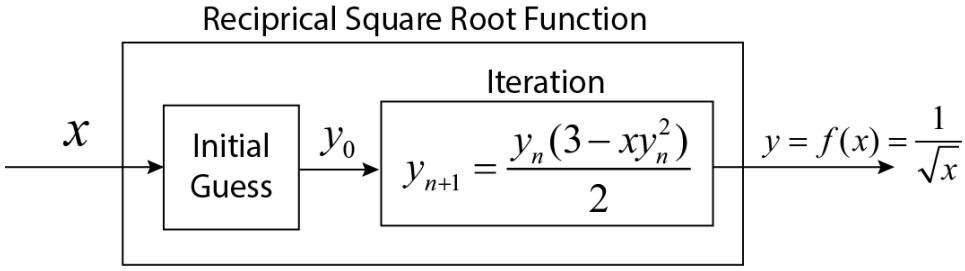
The goal of this lab is to implement the reciprocal square root function in hardware. A top level view of the function can be seen in the following figure.



This function can be solved using Newton's method where we define $F(y)=\frac{1}{y^2}-x$ and then solve for the positive root $F(y)=0$. This positive root can be solved in a iterative manner using $y_{n+1}=y_n-\frac{F(y_n)}{\frac{dF(y_n)}{dy}}$ where $\frac{dF(y)}{dy}=\frac{-2}{y^3}$. This gives us the iterative update formula:

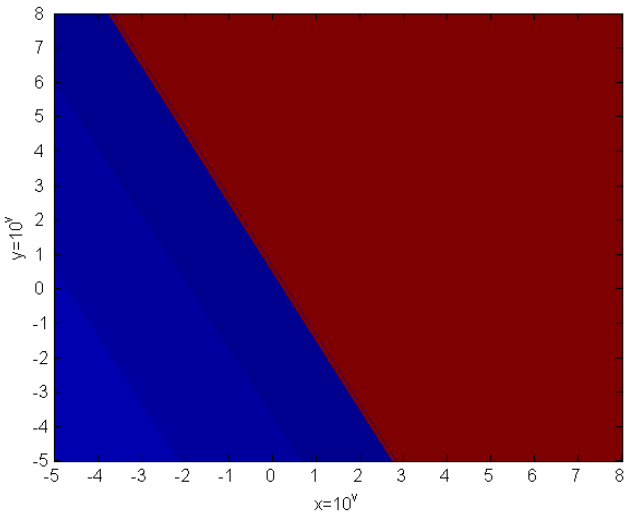
$$y_{n+1}=\frac{y_n(3-xy_n^2)}{2}$$

We start the iterations with y_0, which is our initial guess. This means we should break up the reciprocal square root function computation into two parts as seen in the next figure.

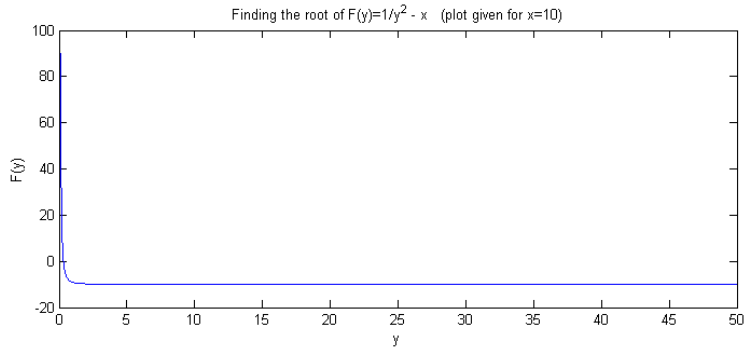


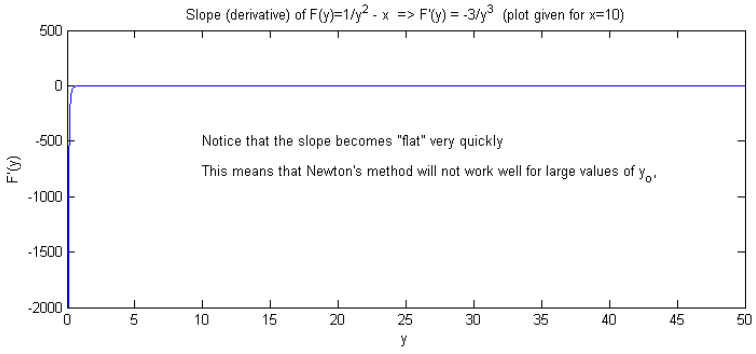
Create a top level VHDL component called rsr.vhd (reciprocal square root) that instantiates two sub-components. The first component computes the initial guess y_0 and the second component implements the iterations necessary for Newton's method.

It turns out that we can't use arbitrary values for our initial guess y_0 . In the figure below, the blue shaded region is where the function converges. The red shaded region is where the function doesn't converge. This shows that the method won't work for large values of the initial estimate y0.



The problem with convergence for large values of y0 is that the slope becomes flat quickly and due to finite computer precision, the slope becomes zero, which causes Newton's Method to fail (we don't know which direction in which to take a step).





Component 1 - Computing y0

Since we are implementing this in hardware, we need a hardware friendly way of computing the initial estimate y0. Use the following hardware friendly steps to create a VHDL component that finds the initial estimate y0 (See the lecture notes on how this was derived).

1. Determine **Z**, the number of leading zeros in x that has bit-width W. A Matlab function lzc.m is provided that will create a lzc VHDL component. The lzc.vhd component will report back the number of leading zeros in the input std_logic_vector.
   Note: Use W=36 and F=18 for this lab exercise.
2. Compute $\beta = W - F - Z - 1$ and note whether beta is even or odd.
3. Compute $\alpha = -2\beta + \frac{1}{2}\beta$ (beta even) or $\alpha = -2\beta + \frac{1}{2}\beta + \frac{1}{2}$ (beta odd). Note that beta is shift both left and right one bit and then subtracted.
4. Compute $x_\alpha = x2^{\alpha}$ by shifting **x** by alpha-bits.
5. Compute $x_\beta = x2^{-\beta}$ by shifting **x** by beta-bits.
6. Get $(x_\beta)^{-\frac{3}{2}}$ via a lookup table (use the fractional bits of x_β as the address)
7. Compute $y_0 = x_\alpha (x_\beta)^{-\frac{3}{2}}$ (beta even) or $y_0 = x_\alpha (x_\beta)^{-\frac{3}{2}} 2^{-\frac{1}{2}}$ (beta odd).

Component 2 – Newton's Method

Once you have y_0, you will then need to create a second component that contains a state machine that iterates and finds y_n. The number of iterations will depend on the numerical precision you need. Choose the number of iterations that is consistent with the choice F=18.

$$y_{n+1} = \frac{y_n(3 - xy_n^2)}{2}$$

Test and Verification

You will need to use Matlab's HDL Verification toolbox to verify your design. The HDL Verification toolbox will drive ModelSim for you. It allows you to create test vectors in Matlab and then compare the output you get back from ModelSim to check if the outputs are correct given the inputs that were put into the unit under test. See the separate document Matlab_VHDL_CoSimulation.pdf for the process you will need to

follow to verify your design. You will need to look at the following documents to perform the test and verification (which are on the D2L site under Lab 4):

- Matlab_VHDL_CoSimulation.pdf (How to use the Matlab Cosimulation Wizard)
- Setting up Quartus II Simulation Libraries.pdf (So that Quartus will work with ModelSim)
- Setting_fimath_attributes.pdf (So you can compute in Matlab what your hardware is doing).

The example fimath properties shown below will keep the same W and F widths after every multiplication for addition rather than growing larger, which is the default behavior.

```
Fm = fimath('RoundingMethod' ,'Floor',...
    'OverflowAction'         ,'Wrap',...
    'ProductMode'            ,'SpecifyPrecision',...
    'ProductWordLength'      ,W,...
    'ProductFractionLength'  ,F,...
    'SumMode'                ,'SpecifyPrecision',...
    'SumWordLength'          ,W,...
    'SumFractionLength'      ,F);
```

Instructor Verification Sheet
Have this sheet signed off
and upload your VHDL and Matlab code to D2L
to get credit for the lab.

Lab 4

Implementing the

reciprocal square root $y = \frac{1}{\sqrt{x}}$

via Newton's Method.

Due Date: 2/27/18

Name : _____

Demo: Show Matlab verifying your design over a uniform distribution of input values.
You will need to answer these questions:

1. How precise can you get your results?
2. How many iterations are you performing?
3. What is your Max Error?
4. What is your Min Error?
5. What is our Average Error?

Verified: _____ Date:_____