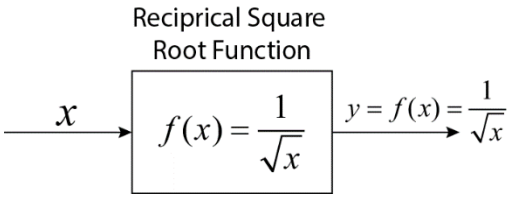
EELE 468
Lab 1
Due February 11, 2020

Creating a Hardware
Reciprocal Square Root Function

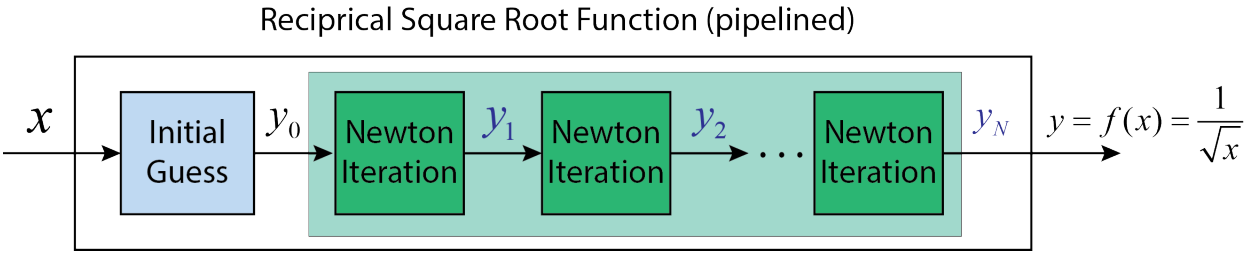
The goal of this lab is to implement the reciprocal square root function in hardware. A top level view of the function can be seen in the following figure.



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.

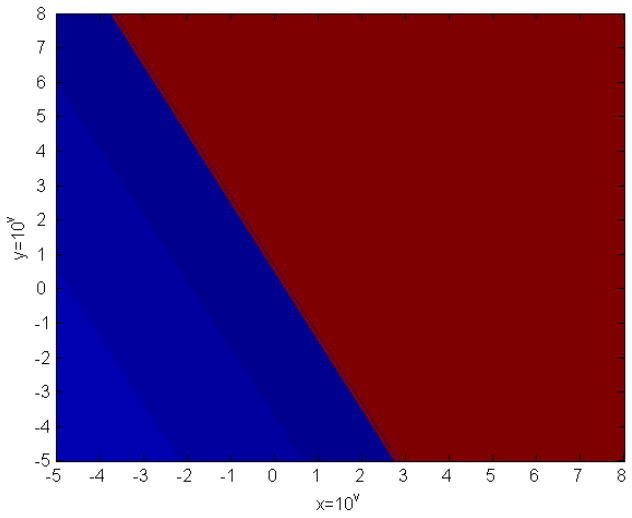


Create a top level VHDL component called rsqrt.vhd (reciprocal square root) that instantiates two sub-components. The first component computes the initial guess y_0 and the second component implements Newton's method that has been unrolled in a pipeline (each iteration is a separate component).

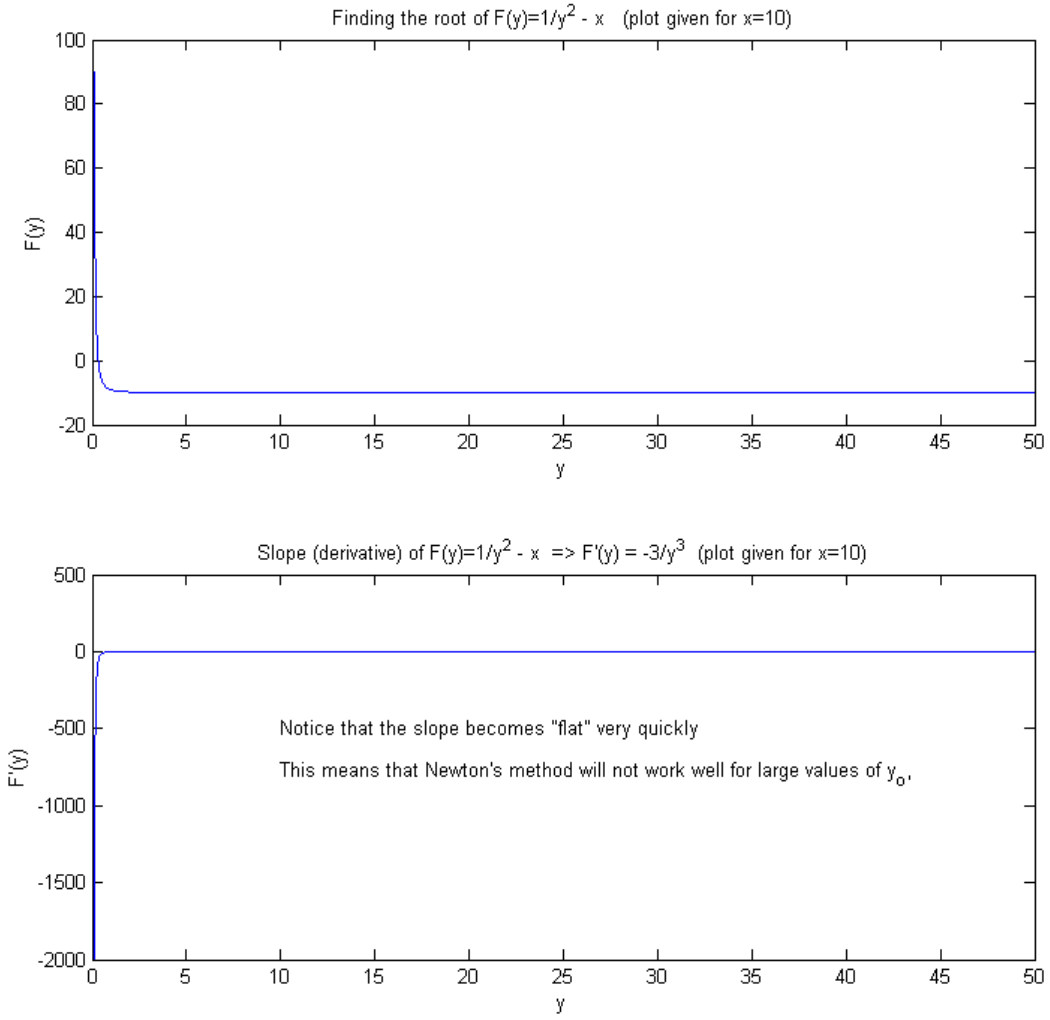
You will need to use the entity shown below. The entity uses generics that will allow your function to be compiled for any arbitrary length fixed-point data type. You will need to use generics in your code since one of your tests will be to use a datatype (vector length) that you don't know the length of at this time.

```
entity rsqrt is

   generic (W_bits       : positive := 32;  -- size of word
            F_bits       : positive := 16;  -- number of fractional bits
            N_iterations : positive := 3);  -- number of Newton's iterations

   port    (x : in  std_logic_vector(W_bits-1 downto 0);
            y : out std_logic_vector(W_bits-1 downto 0));

end entity rsqrt;
```

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 y_0 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 for data types with arbitrary W (size of vector) and F (number of fractional bits). 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 on D2L that will create a lzc VHDL component with arbitrary W. The lzc.vhd component will report back the number of leading zeros in the input std_logic_vector.
2. Compute $\beta = W - F - Z - 1$ and note whether beta is even or odd. Note that W and F are generics.
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 shifted both left and right one bit and then subtracted. Don't multiply or divide by two since this is free in hardware.
4. Compute $x_\alpha = x2^{\alpha}$ by shifting input **x** by alpha-bits.
5. Compute $x_\beta = x2^{-\beta}$ by shifting input **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). Information on how to create the lookup table using a ROM in the FPGA fabric will be supplied.
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).

Create a block diagram for computing the initial estimate y_0. Show the data flow and computations as subblocks. This block needs to be pipelined, which means that data is continuously flowing through the block and a new input value can be accepted every clock cycle. **Submit this block diagram as Homework #1 on D2L (Due Friday 1/24).**

Component 2 – Newton's Method

Once you have y_0, you will then need to create a second component that takes y_n as the input and computes y_{n+1}. The number of iterations you implement will depend on the numerical precision you need.

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

To create a pipelined design, you will need to use the **generate statement** in VHDL and control the number of times this component is created with the **generic N_iterations**. Note: you will probably want to create the vector size in this component with F_bits a

few bits larger than the generic F_bits to get to the precision you need (rounding errors, etc.). You then will send out the final value that has F_bits of precision.

Test and Verification

In order to verify your VHDL code you will need to do the following:

1. Create the VHDL component rsqrt.vhd
2. Create an equivalent bit-true Matlab function rsqrt.m using the fixed-point toolbox.
3. Show that these two approaches give identical results when simulating rsqrt.vhd in ModelSim.
    a. Matlab generates an array of text vectors (input values).
    b. Matlab writes the test vectors to a file
    c. ModelSim reads this file, performs the simulation, and writes results to a file.
    d. Matlab reads the ModelSim generated file and compares with the output of the bit-true function created with the fixed-point toolbox. The results need to be identical.

Matlab's HDL Verification toolbox allows Matlab to driver ModelSim for you. However, this toolbox doesn't support the use of generics so we will use ModelSim and Matlab's fixed-point toolbox instead.

Creating bit-true Matlab code

You will need to create bit-true Matlab code that does exactly what your VHDL code is doing. In order to do this, you will need to use the fixed-point toolbox in Matlab and set the fimath properties for each operation performed in VHDL.

An example fimath setup is shown below that will keep the same W and F widths after every multiplication and addition rather than growing larger, which is the default behavior in the fixed-point toolbox. See the document Setting_fimath_attributes.pdf on D2L for more fimath settings.

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

Develop VHDL and Matlab together

You need to be systematic and test each VHDL code segment with the equivalent bit-true Matlab code. Don't move on until your VHDL and Matlab subblocks or code segments agree with each other.

Note: In Matlab you can print out the hex values of the fixed-point values after each operation (e.g. a.hex). This is convenient so that you don't have to change the radix settings in ModelSim when you run simulations and are comparing results.

ModelSim

You can get a student version of ModelSim at:
https://www.mentor.com/company/higher_ed/modelsim-student-edition
This will allow you to get started and test your computational sublocks.

The steps that you will need to do for the ModelSim simulations are:

1. Be able to perform file I/O. You will need to be able to read in a file that Matlab generates (fixed-point input values, i.e. the test vectors) and write results out to a file. Details will be given on how to do this.
2. Setup Quartus simulation libraries so that IP generated by Quartus will work with ModelSim. This is needed for the ROM. See the document Setting up Quartus II Simulation Libraries.pdf on D2L. (Note: we need to check if the student version can do this, otherwise this will have to be done with the full-blown version of ModelSim on the computers in the Digital Lab).

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

Lab 1

Implementing the

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

via Newton's Method.

Due Date: 2/11/20

Name : _____

Demo: For the given values of W_bits, F_bits, and N_iterations, show that your VHDL code and Matlab code give identical results.

Verified: _____ Date:_____