ECE4700J Computer Architecture

Summer 2024

Lab #2 A Pipelined Integer Square Root Module

Due: 11:59pm (Beijing time) June. 4th, 2023

Logistics:

- This lab is an individual exercise.
- All the design code should be in SystemVerilog.
- Please use the discussion board on Piazza for Q&A.
- All codes MUST be submitted to the assignment of Canvas.
- Internet usage is allowed and encouraged.

Contents

1 Overview

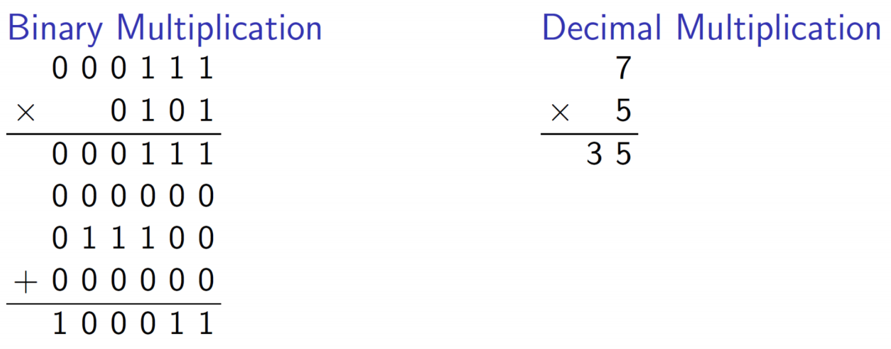
In this lab, you will study how to design some complex hardware module design.

- Study how user-defined types are used in SystemVerilog.
- Study how pipelining is achieved by SystemVerilog.
- Be able to implement more complex FSMs.

2 Designing the Pipelined Multiplier

2.1 Introduction

We have provided you with a skeleton of pipelined multiplier (incomplete), found in `pipe_mult.sv` and `mult_stage.sv`. The multiplier does multiplication in stages, somewhat like you would have learned to carry out multiplication in elementary school. It multiplies the first 8 bits of the multiplier with the whole multiplicand in one clock cycle, then the next 8 bits of the multiplier against a shifted multiplicand, and so on to get 8 partial products. Summing those gives us the desired multiplication. This means that each multiplication will take 8 clock cycles. Each partial product is created by a separate multiplier stage, which can be found in the `mult_stage.sv` file. For detailed sample steps of a pipelined multiplier, you can refer to the following steps.

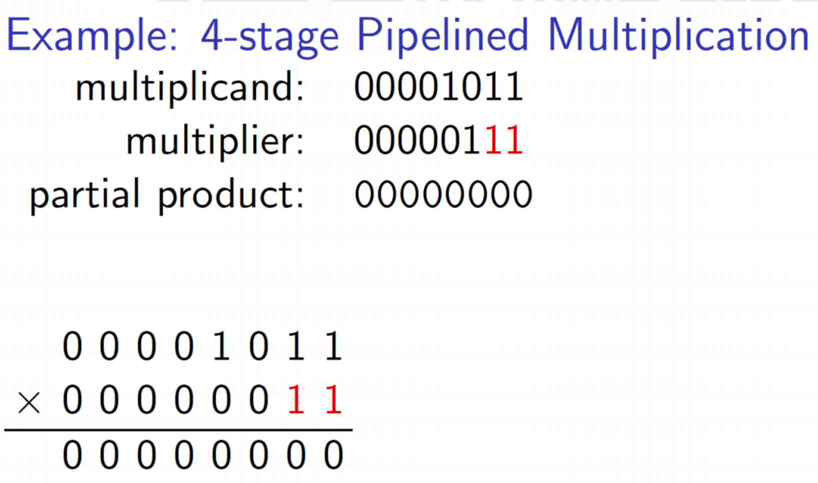


Binary Multiplication

$$\begin{array}{r} 0\,0\,0\,1\,1\,1 \\ \times \quad 0\,1\,0\,1 \\ \hline 0\,0\,0\,1\,1\,1 \\ 0\,0\,0\,0\,0\,0 \\ 0\,1\,1\,1\,0\,0 \\ +\;0\,0\,0\,0\,0\,0 \\ \hline 1\,0\,0\,0\,1\,1 \end{array}$$

Decimal Multiplication

$$\begin{array}{r} 7 \\ \times \quad 5 \\ \hline 3\,5 \end{array}$$



Example: 4-stage Pipelined Multiplication

multiplicand: 00001011

multiplier: 00000011

partial product: 00000000

$$\begin{array}{r} 0\,0\,0\,0\,1\,0\,1\,1 \\ \times\;0\,0\,0\,0\,0\,0\,1\,1 \\ \hline 0\,0\,0\,0\,0\,0\,0\,0 \end{array}$$

multiplicand:	00001011	<< 2
multiplier:	00000111	>> 2
partial product:	00100001	

$$\begin{array}{r} 0\ 0\ 0\ 0\ 1\ 0\ 1\ 1 \\ \times\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1 \\ \hline 0\ 0\ 1\ 0\ 0\ 0\ 0\ 1 \end{array}$$

multiplicand:	00101100
multiplier:	00000001
partial product:	00100001

$$\begin{array}{r} 0\ 0\ 1\ 0\ 1\ 1\ 0\ 0 \\ \times\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1 \\ \hline 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \end{array}$$

multiplicand:	00101100	<< 2
multiplier:	00000001	>> 2
partial product:	01001101	

$$\begin{array}{r} 0\ 0\ 1\ 0\ 1\ 1\ 0\ 0 \\ \times\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1 \\ \hline 0\ 0\ 1\ 0\ 1\ 1\ 0\ 0 \end{array}$$

2.2 Assignment

Your first assignment will be to finish the 8-stage pipelined multiplier design and pass the **post-synthesis timing simulation**. We have provided you a testbench file in `mult_test.sv` with just the skeleton of the testbenches, carefully read it and write your own testcases for your design.
Your second assignment is to modify the pipelined multiplier we've provided to work as

both a 4-stage multiplier and as a 2-stage multiplier. You can either do this by copying the codes and having separate 2, 4 and 8 stage multipliers or by figuring out a combination of preprocessor macros or parameters that set pipeline depth. Once you have the other two multipliers, you will need to pass the **post-synthesis timing simulation** for 4 and 8 stage multipliers, and pass the **post-synthesis functional simulation** for 2 stage multiplier. (Hint: The default clock period 500ns in our testbench may not be enough for your 2-stage or 4-stage pipelined multiplier. You may need to keep increasing your clock cycle length to pass the tests.)

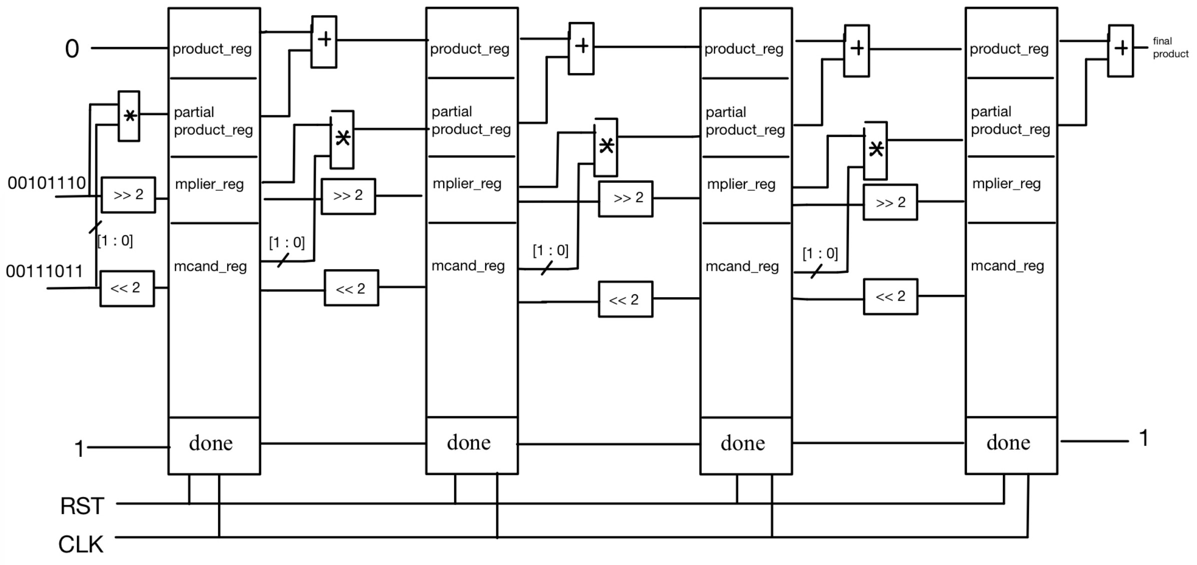


Figure 1: an 8 bit 4 stage pipeline multiplier

For your submission, please hand in three different modules together in one `pipe_mult.sv` file. The three modules should named as follows: `mult` for the 8-stage multiplier, `mult_4` for the 4-stage multiplier, and `mult_2` for the 2-stage multiplier. If your implementation uses parameters, please copy your module for three times, and change the default value of the parameters for each corresponding stages.

3 Designing the Integer Square Root Module

3.1 Introduction

Now, you will need to create a module that uses the multiplier that we supplied, the 8 stage multiplier. You will be writing a module to compute the integer square root of a 64-bit number. It will generate a 32-bit number that is the largest integer that is not larger than the square root of the number provided. For example, the integer square root of 24 is 4.
The module declaration is as follows:

```
module ISR(
    input                   reset,
    input          [63:0]   value,
    input                   clock,
    output logic   [31:0]   result,
    output logic            done
);
```

It should operate as follows:

- If reset is asserted during a rising clock edge (synchronous reset), the value signal is to be stored.
- If reset is asserted part way through a computation, the result of that computation is discarded and a new value is latched into the module.
- When the module has finished computing the answer, the output is placed on the result line and done line is raised on the same cycle.
- It must not take more than 600 clock cycles to compute a result (from the last clock that reset is asserted to the first clock that done is asserted.)

We do not suggest that you pipeline this module. You will likely need to perform something like a binary search to find the result a simple algorithm is as follows:

Algorithm 1 Integer Square Root

```
1: procedure ISR(value)
2:     for $i \leftarrow 31$ to $0$ do
3:         proposed_solution[$i$]$\leftarrow 1$
4:         if proposed_solution$^2 >$ value then
5:             proposed_solution[$i$]$\leftarrow 0$
6:         end if
7:     end for
8: end procedure
```

Note that loops do not have a direct hardware equivalent. What hardware design technique lets us implement a procedure like this?

3.2 Assignment

Once you have the module written and tested, synthesize it and do **post-synthesis timing simulation**. This will probably take several minutes at least.

4 CORDIC Implementation (Optional)

4.1 Introduction

Coordinate rotation digital computer (CORDIC) algorithm is widely used to calculate hyperbolic functions, multiplications, square root, and so on. It makes digital implementation on hyperbolic functions possible and makes multiplications work without multipliers. In this section, you will have a rough view of CORDIC algorithm and implement it using SystemVerilog.

Suppose we rotate a vector (x_{in}, y_{in}) by θ like

$$\begin{cases} x_{out} = x_{in}\cos(\theta) - y_{in}\sin\theta \\ y_{out} = x_{in}\sin\theta + y_{in}\cos\theta \end{cases}$$

We can use $\tan\theta$ to simplify these equations

$$\begin{cases} x_{out} = \cos\theta(x_{in} - y_{in}\tan\theta) \\ y_{out} = \cos\theta(x_{in}\tan\theta + y_{in}) \end{cases}$$

In the above equations, we need to perform four multiplications plus some additions and subtractions. As the previous sections shows, the calculation of multiplication needs lots of hardware resources. To avoid complicated calculations, the CORDIC algorithm introduces two basic ideas.

- Rotating the input vector by θ_d is equal to rotating the vector by some smaller angles, θ_i, $i = 0, 1, ..., n$, provided $\theta_d = \sum_{i=0}^{n} \theta_i$. Note that these smaller angles can also be negative angles too.
- We can choose the small elementary angles in a way that $\tan\theta_i = 2^{-i}$ for $0, 1, ..., n$. In this way, multiplications by $\tan\theta_i$ can be implemented with bitwise shifting instead of general multipliers.

Given these two basic ideas, we can always find some θ_i that fit both $\theta_d = \sum_{i=0}^{n} \theta_i$ and $\tan\theta_i = 2^{-i}$ for any given θ_d. The accuracy of this algorithm will be influenced by the choice of n. If we increase n, we can increase the accuracy of the final result.

Now, we can imply the multiplication involving $\tan\theta$ using bit shifts, but we still need to perform two other multiplications by $\cos\theta$. Fortunately, $\cos\theta$ works as a scaling factor in this algorithm, and we can calculate all the $\cos\theta_i$ together after we have perform all the elementary rotations of the CORDIC algorithm, which is introduced before. For example, if we want to rotate the input vector by $57.535°$, which is $45° + 26.565° - 14.03°$. The first rotation by $45°$ will give us:

$$\begin{bmatrix} x_0 \\ y_0 \end{bmatrix} = \cos(45°) \begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_{in} \\ y_{in} \end{bmatrix} \tag{1}$$

For the second rotation, we will get:

$$\begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = \cos(45°)\cos(26.565°) \begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & -2^{-1} \\ 2^{-1} & 1 \end{bmatrix} \begin{bmatrix} x_{in} \\ y_{in} \end{bmatrix} \tag{2}$$

For the third rotation, we will have:

$$\begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \cos(45°)\cos(26.565°)\cos(-14.03°) \begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & -2^{-1} \\ 2^{-1} & 1 \end{bmatrix} \begin{bmatrix} 1 & 2^{-2} \\ -2^{-2} & 1 \end{bmatrix} \begin{bmatrix} x_{in} \\ y_{in} \end{bmatrix} \tag{3}$$

As we proceed with the algorithm, the angle of rotation rapidly becomes smaller and smaller, and hence $\cos\theta$ tends toward unity. For example, with $i = 6$, θ_i becomes $0.895°$ and $\cos 0.895° = 0.99987$. So if the algorithm is designed to have more than 6 iterations, we can get the scaling factor as:

$$K \approx \cos(45°)\cos(26.565°) \times \ldots \times \cos(0.895°) \approx 0.6072 \tag{4}$$

In this way, we can ignore all the $\cos\theta$ terms in the algorithm, and apply the scaling factor of 0.6072 to the rotated vector at the end of the algorithm, or directly apply the scaling factor directly to the input vector before the algorithm.

Therefore, omitting the scaling factor term, we obtain the following equations for the CORDIC algorithm:

$$\begin{cases} x[i+1] = x[i] - \sigma_i 2^{-i} y[i] \\ y[i+1] = y[i] + \sigma_i 2^{-i} x[i] \end{cases}$$

where $\sigma_i \in \{+1, -1\}$ determines the sign of the elementary angle.

Now we know how to rotate the vector for a set of given elementary rotation, but how can we rotate the vector for any given angle. In other words, we need to choose the value of θ_i for each iteration of the algorithm. To reach this goal, we need to compare the overall achieved rotation with the desired angle. If the desired rotation is larger than the achieved rotation, we need to rotate counter-clockwise in the next iteration. We can then obtain the final version of equations for the CORDIC algorithm.

$$\begin{cases} x[i+1] = x[i] - \sigma_i 2^{-i} y[i] \\ y[i+1] = y[i] + \sigma_i 2^{-i} x[i] \\ z[i+1] = z[i] - \sigma_i \tan^{-1}(2^{-i}) \end{cases}$$

4.2 Assignment

In this part of assignment, you need to finish the design of a CORDIC module based on the starter file `cordic.sv` we provide. Hints are provided in the starter file through comments. You only need to edit the "TODO" part in the design file. After you fully implement the design, you are required to test you design using `cordic_test.sv` based on **the behavioral simulation**. Only a few test cases are provided by us, so you may need to design your own test cases to ensure your design works well.

In this optional assignment, **you will be given an up to 10% bonus to the total score of Lab 2** if you are able to correctly implement the optional assignment. This will be counted towards your final grade.

5 Deliverables

Please compress all the following files into a single `.zip` file. The file name should be in the form of <FirstName>_<Lastname>_<Student_Number>.
(e.g. San_Zhang_5203xxxxxxxx.zip)

- Section 2.2: `mult_stage.sv` file.
- Section 2.2: `pipe_mult.sv` file.
- Section 3.2: `ISR.sv` file.
- Section 3.2: `test_ISR.sv` file.
- Section 4.2: `cordic.sv` file. (Optional)

6 Grading policy

Factors	Percentage
Section 2.2 design	50%
Section 3.2 design and test	50%
Section 4.2 design	bonus

References

1. UM-SJTU JI ECE4700J SU 2022 Lab3
2. Umich EECS470 WN 2021 Project2
3. "An Introduction to the CORDIC Algorithm," *All About Circuits*, May. 31, 2017. [Online]. Available: `https://www.allaboutcircuits.com/technical-articles/an-introduction-to-the-cordic-algorithm/`.

Acknowledgement

- Haoyang Zhang (ECE4700J SU 2022 TA)
- Jon Beaumont (University of Michigan)