

May 2024

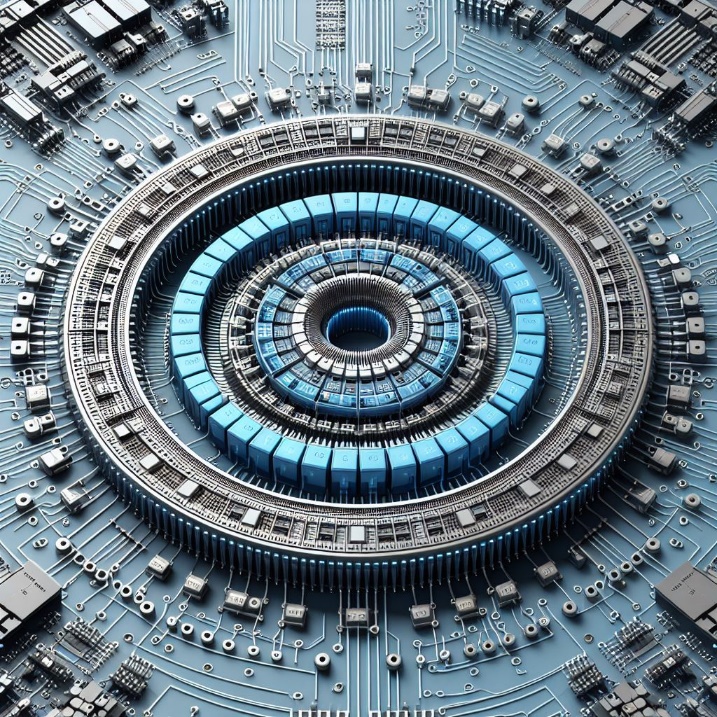
Amirreza Nadi Chaghadari | 810101538

Digital systems

Dr. Navabi

Flip-Flops, Registers, Shifters

**Computer Assignment 4**



**Table of Contents**

[Datapath 1](#_Toc168936897)

[Methodology 1](#_Toc168936898)

[Schematic 2](#_Toc168936899)

[SystemVerilog Description 4](#_Toc168936900)

[Control Unit (Controller) 5](#_Toc168936901)

[Methodology 5](#_Toc168936902)

[State Diagram 6](#_Toc168936903)

[Controller Schematic 7](#_Toc168936904)

[SystemVerilog Description 9](#_Toc168936905)

[Divider Verilog Description 11](#_Toc168936906)

[Pre-synthesis Simulations 12](#_Toc168936907)

[Partial Waveforms 13](#_Toc168936908)

[Top Module Waveforms 14](#_Toc168936909)

[Synthesis and Post-Synthesis Simulations 15](#_Toc168936910)

[Synthesis 15](#_Toc168936911)

[Netlist Simulations: Timing Challenges 17](#_Toc168936912)

[Netlist Simulations: Testbench and Waveforms 18](#_Toc168936913)

[Netlist Simulations: Confirming Delays and Timings (Clock Level) 19](#_Toc168936914)

[Pre/Post Synthesis Simulations 20](#_Toc168936915)

[Testbench 20](#_Toc168936916)

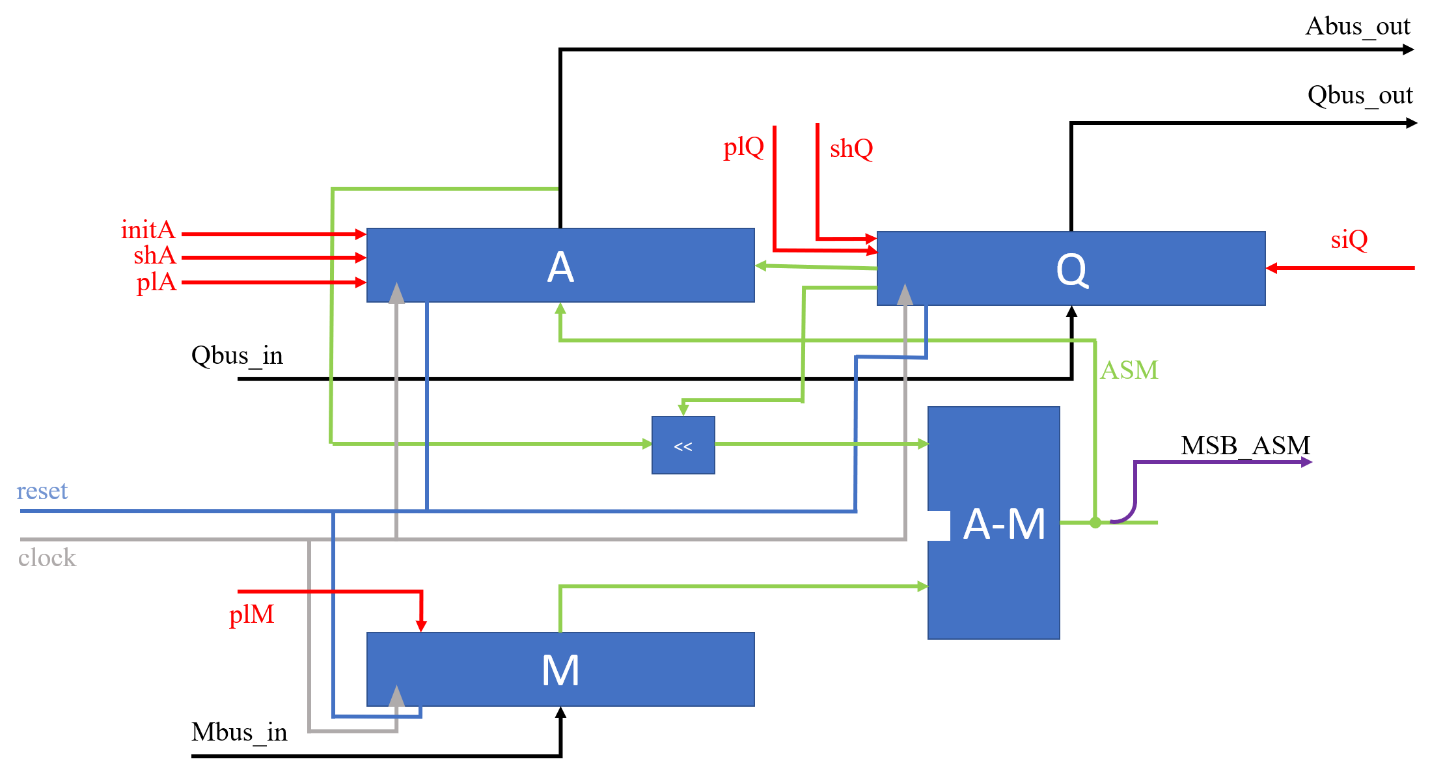
Datapath

Methodology

The algorithm suggests that we need at least three registers, i.e. A, Q, and M. at some point during each iteration of the algorithm, A and Q are shifted. So, they need to have shifting ability and therefore, they must have a shift enable input, issued by the controller unit. Also, serial out of Q must be wired to serial in of A. At the beginning of the process, A needs to be initialized to 0 and Q and M need to be loaded from the BUS. So, A needs a control signal for initialization and Q and M need parallel load enable inputs. also, A might need to save A-M. Therefore, it must have a control signal to decide when to save A-M. As mentioned, one of the algorithm’s steps requires subtracting M from A. Therefore, a subtractor is also required. The most significant bit of the subtractor’s result is needed for some decision makings, so it must feedback into the control unit. Also, in order to have a faster divider, it is best to have a barrel shifter between A and M, and the subtractor. There is also a counter involved in the algorithm. But, it is best to include this in the controller. Also, every single control signal needs to be issued by the controller as well. It is best not to contain any controlling logic inside the datapath, so that it gets more maintainable. The schematic of the datapath is shown in the following page.

Schematic

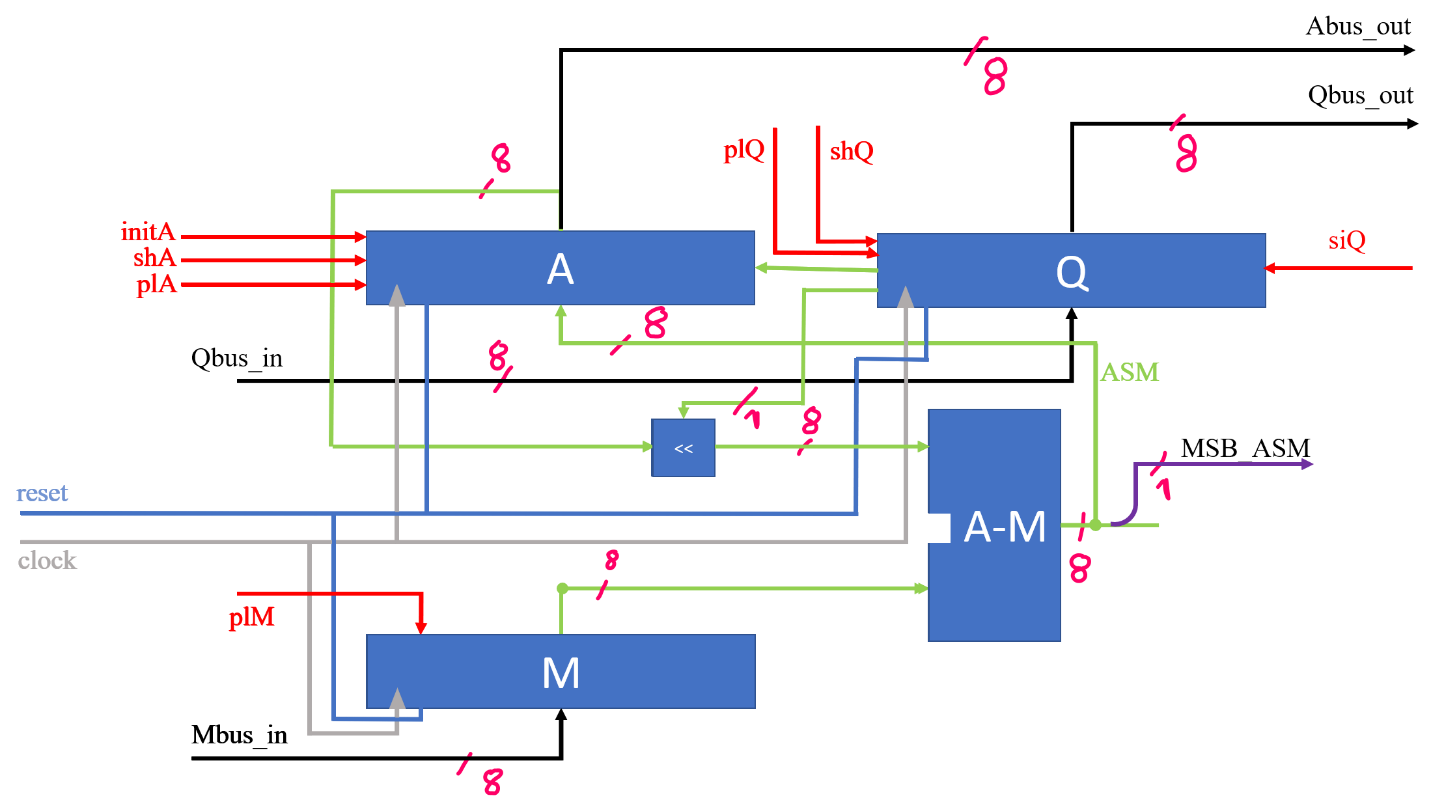
Here’s how the datapath looks like schematically:



Green wires carry signals between components inside the datapath. Red wires are for controlling signals, issued by the controller. Light blue and gray wires are for reset and clock signals and therefore, not to be played with[[1]](#footnote-1)! Violet wires are outputs of the datapath to the controller and finally, black wires are external buses bringing data into and out from the datapath.

There is one thing about the barrel shifter that need to be discussed. it takes 1 bit as the least significant bit and shifts the rest. This shifter makes the divider one clock faster, since it does not need separated states -and therefore, separated clock cycles- for shifting and checking. The image on the next page shows the same circuit, but with wire arrays’ sizes specified.

Datapath: wire arrays’ sizes specified

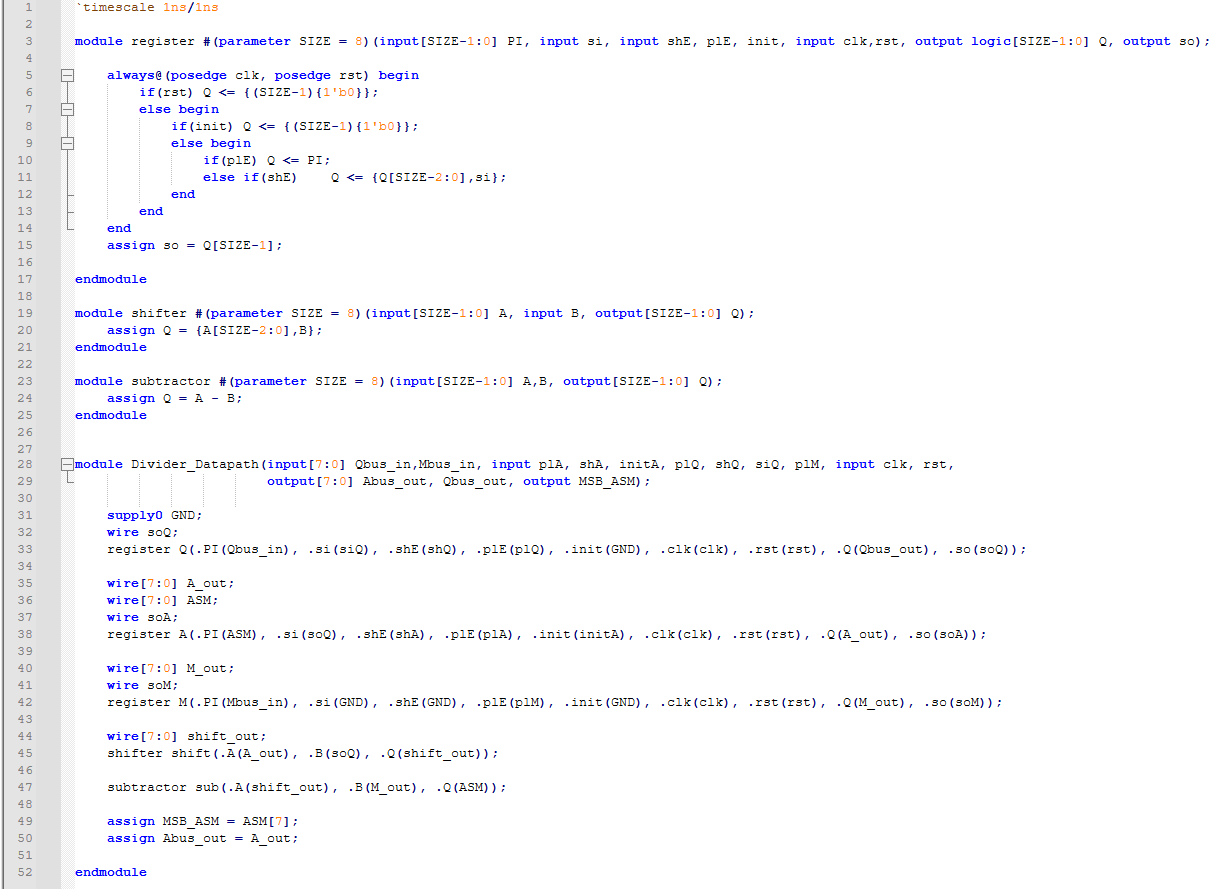


Control signals are always single wires, since I have decided to use one-hot method. Therefore, their wire count is not shown in this picture.

SystemVerilog Description

For easier description, I used the same register type for all three registers, but wired some of their control inputs to GND to make sure they stay inactive. For example, M does not need shifting. Therefore, its shift enable is set to inactive.

SystemVerilog description of the datapath looks like this:

As is shown above, I separately described every single component. All that remained after that were naming wires, instantiating components, and wiring them together.

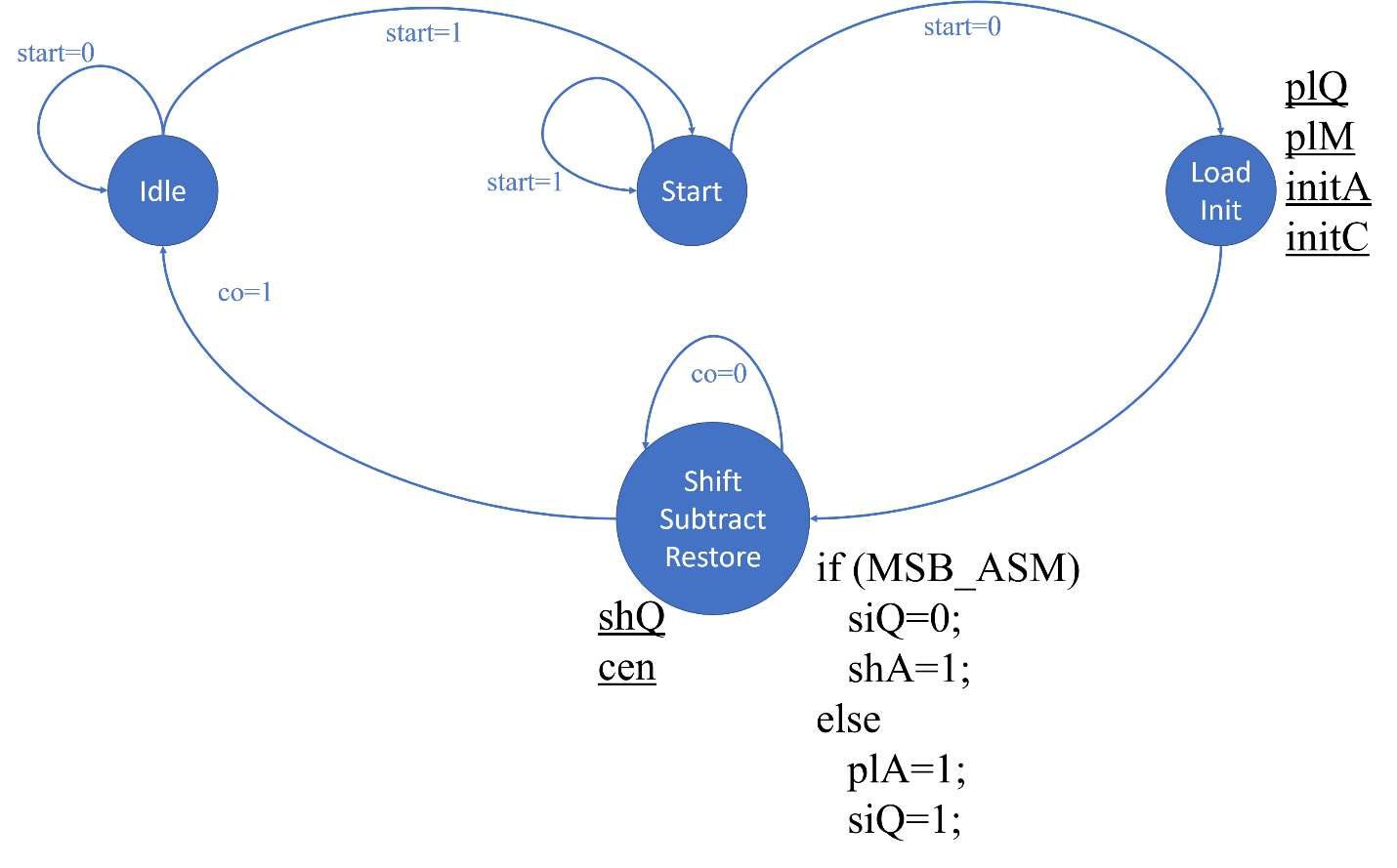
Control Unit (Controller)

Methodology

First things first, we need a default state. I named it **Idle** state. In this state, **ready** signal is issued to show that the divider is ready to begin. This signal could also mean that the divider has finished a given task. The wrapper could take this signal and separate it into two separate ones, should it be needed.[[2]](#footnote-2) The next thing we need is a state where initializations and loadings take place in. I named it **Load Init**. The algorithm requires a set of actions being done a specific number of times. So, the control unit needs a counter as well, and it needs to be initialized in the mentioned state. If we interpreted the counter as a complete state machine, the entire pack of controlling unit would be considered an orthogonal state machine. The next state/states are for following the algorithm’s steps. The steps can be done in a single state, thanks to the barrel shifter we used in the datapath. I call this state   
**Shift Subtract Restore**. The mentioned counter decides how long we stay in this state. The clock pulse that sees 1 on counter’s carry-out, is the last clock we stay in the mentioned in mentioned state. After this state, we go right back to **Idle** state, issuing **ready** signal to announce that the divider has done its job and is ready to receive next set of inputs.

The state diagram of this controller is shown in the following page.

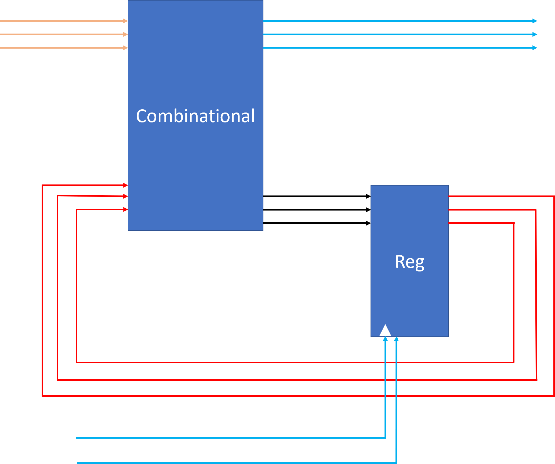
State Diagram



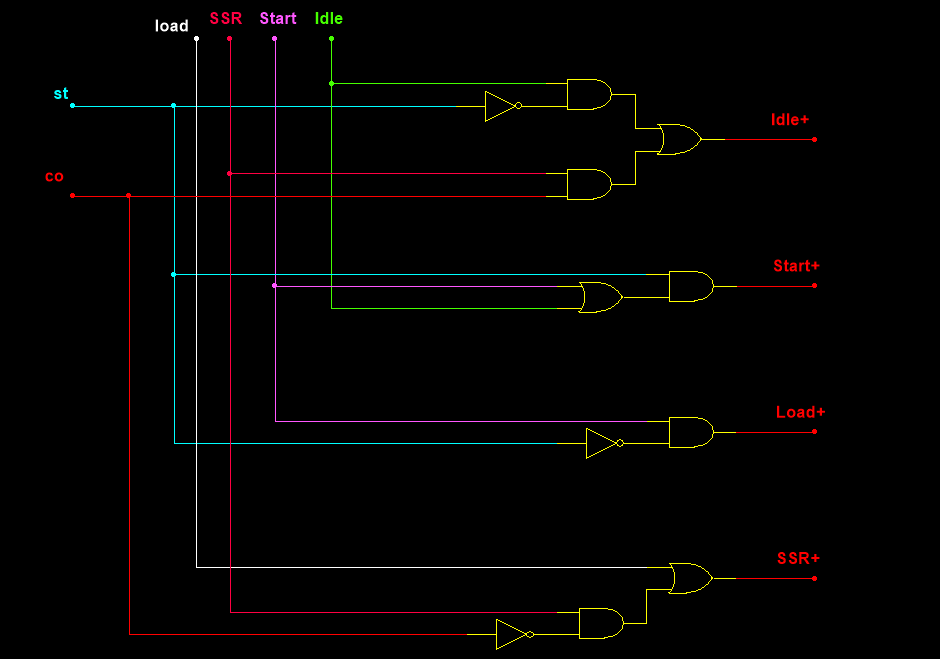
Blue arrows and texts indicate how the machine flows through its states. Black underlined signals are the ones issued in the state they’re written next to and the black code next to **Shift Subtract Restore** state is what makes this machine a hybrid one, since it issues some signals based on the current state and the inputs.

Controller Schematic

The Huffman model says that every sequential circuit can be broken into two parts: storing unit (registers) and combinational part. Here’s the abstract schematic of the Huffman style:

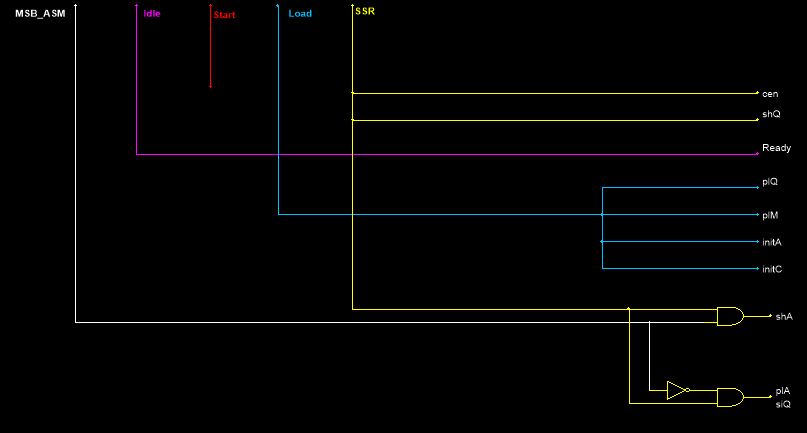


Schematic for state outputs look like this:



Obviously, I have used one-hot method because it is easier and uses less logic, leading to use of less gates. The image on the next page shows other outputs as well.

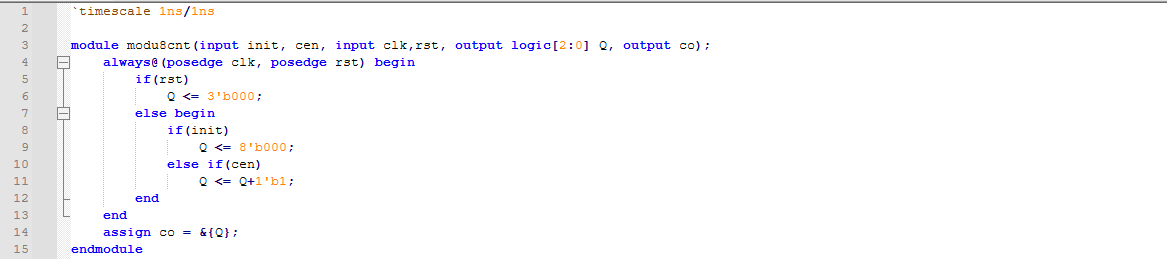
Other outputs of the controller are issued like this:



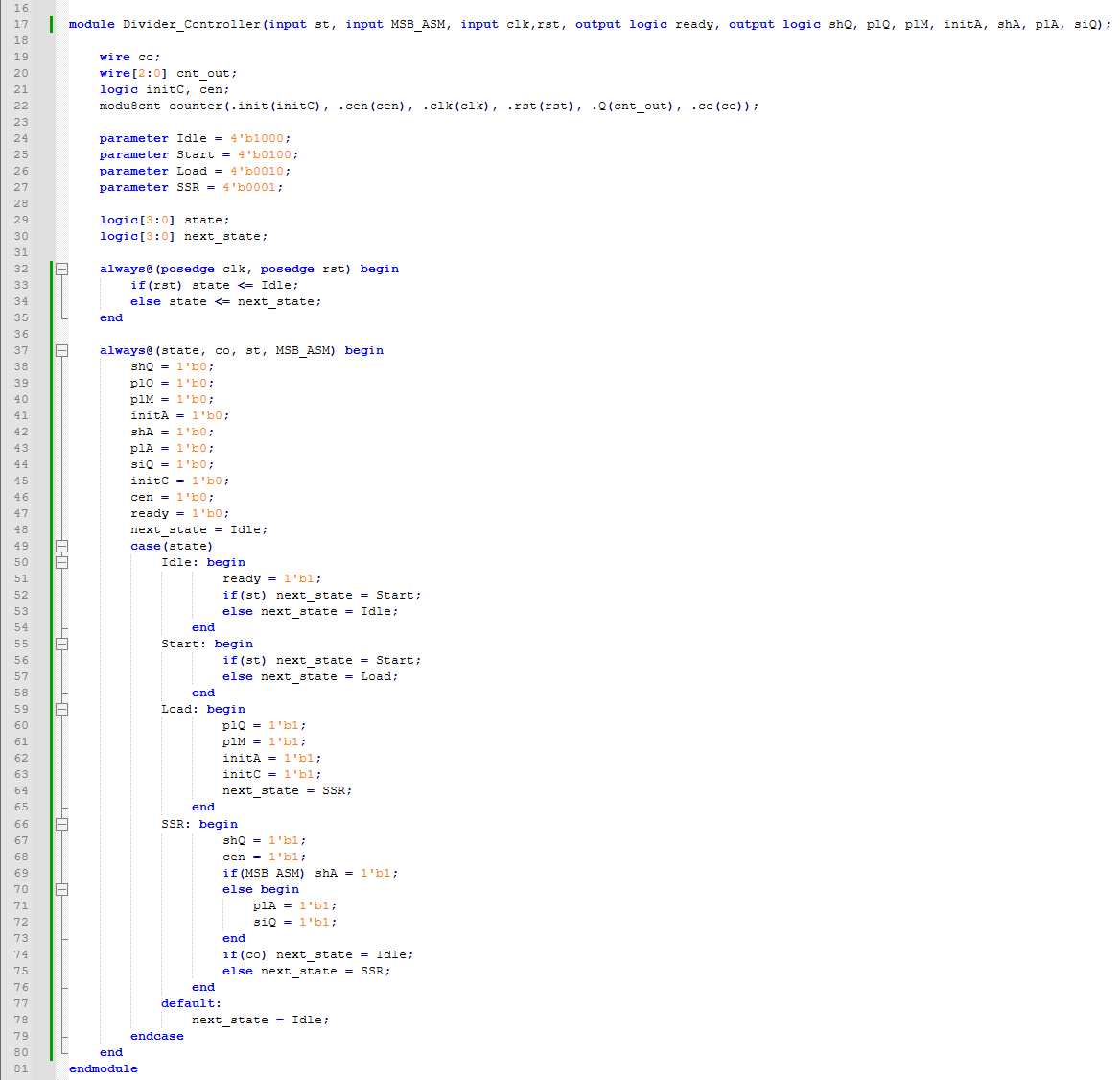
Now we are all set for the SystemVerilog description.

SystemVerilog Description

I broke the controller into two parts to follow Huffman style. Also, the controller needs a counter as well. This is how I described the counter:



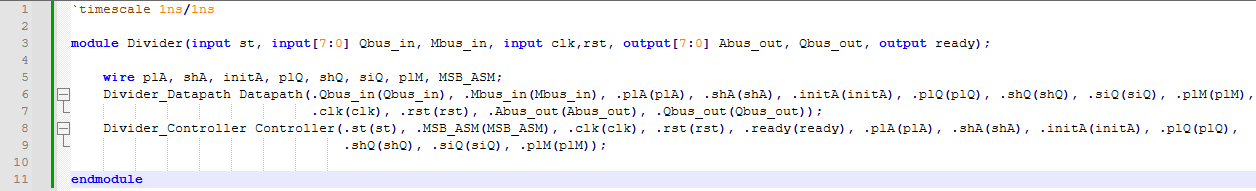
There is really nothing special to it. Just a counter with needed control inputs. as for the state machine itself, my description is shown in the following page.



Lines 19 to 22 instantiate a counter. Lines 24 to 35 handle state changes, which concerns the storage unit. The rest is the combinational part of the Huffman description style.

Divider Verilog Description

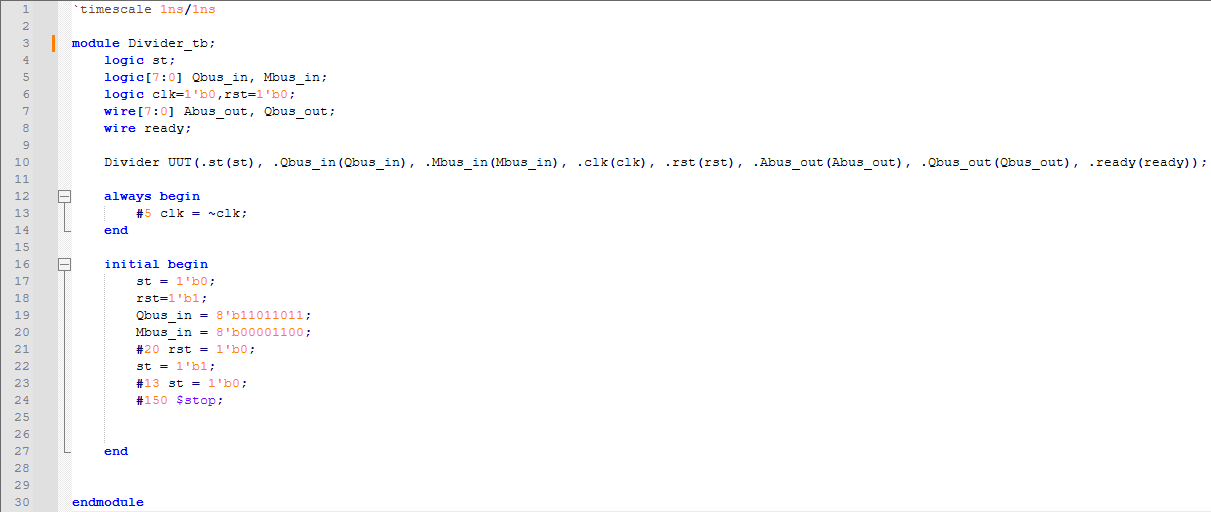
Thanks to all the hard work previously done, describing the divider from here is very simple. All there are to do are instantiating datapath and controller and wiring them together. My description looks like this:



With this, the divider is complete. All there is to do now, is test everything. Instead of testing them all independently, I decided to test the whole thing together, but show internal waveforms as well. The images are shown in the following page.

Pre-synthesis Simulations

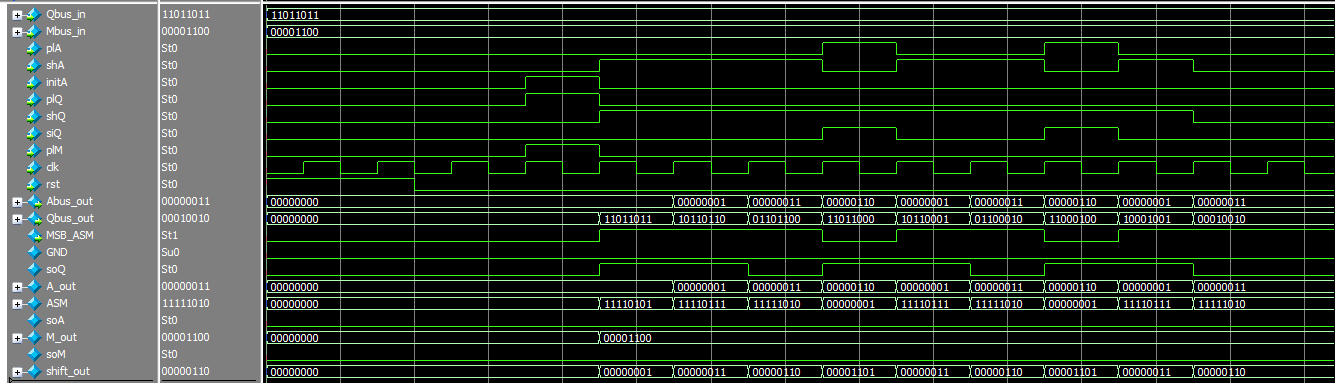
The testbench looks like this:



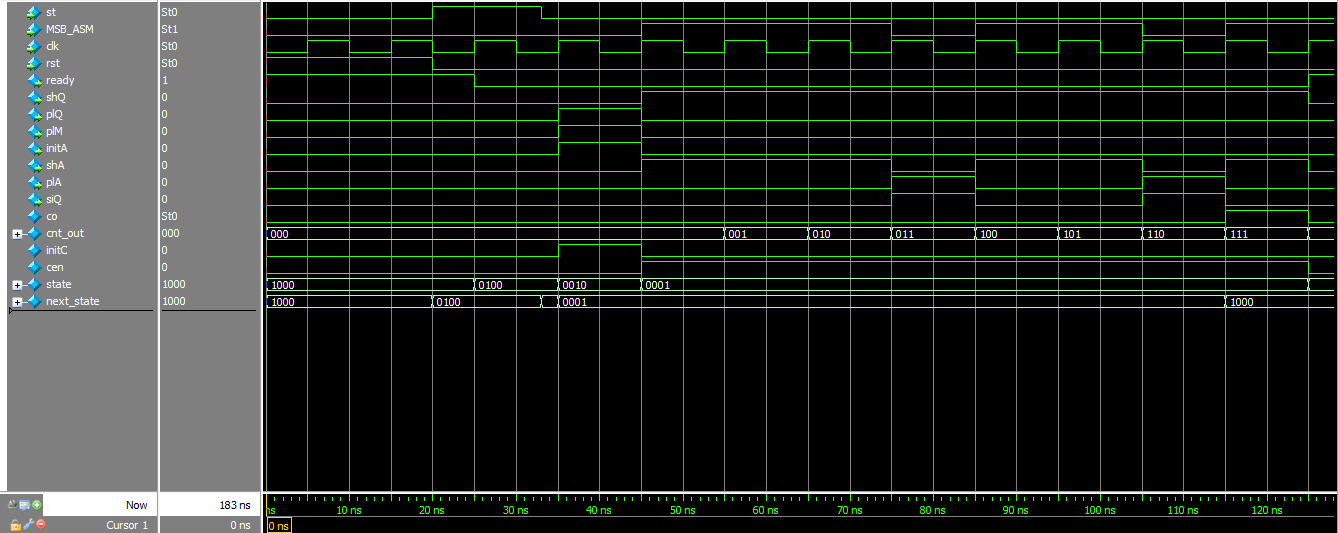
The waveforms are shown in the next page.

Partial Waveforms

Datapath’s waveform looks like this:

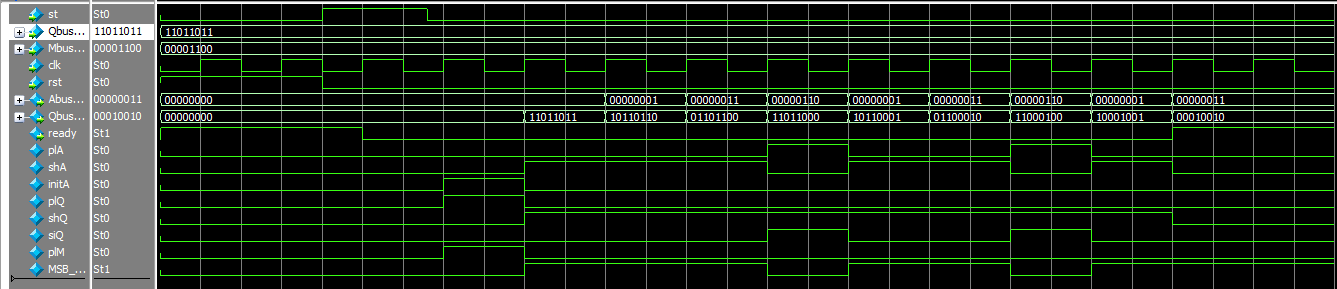


Controller’s waveform looks like this:

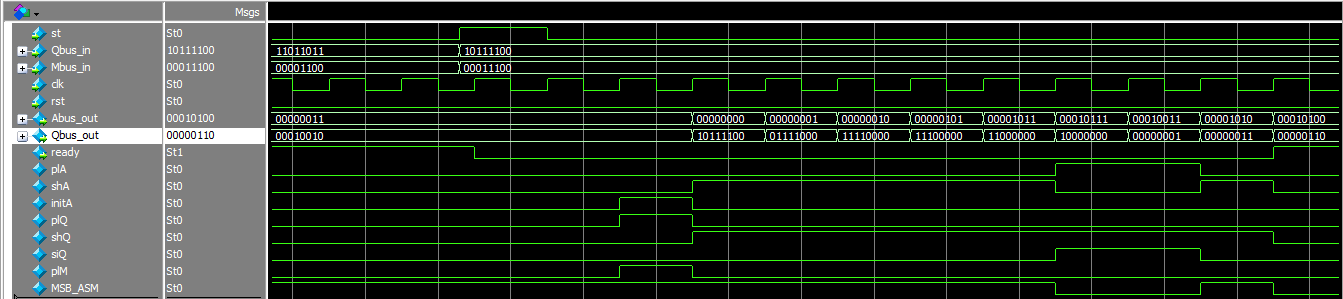


The top module’s waveforms are shown in the next page.

Top Module Waveforms



Inputs are , which translate to . The quotient is 18 and the remainder is 3. Here’s another waveform for another set of inputs:



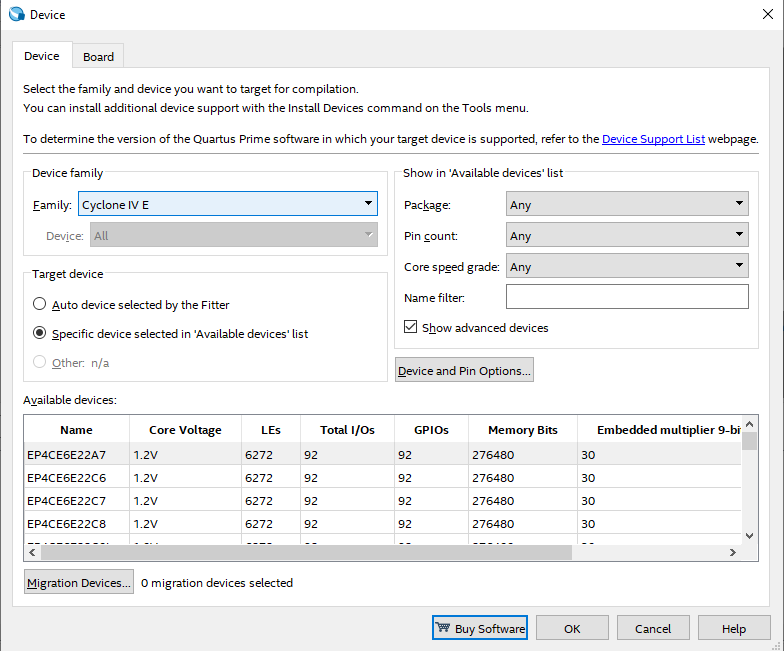
The division is , which translate to . The quotient is 6 and the remainder is 20, which confirm the waveforms.

Now that we have done the pre-synthesis simulations, we can go on and synthesize the circuit in Quartus.

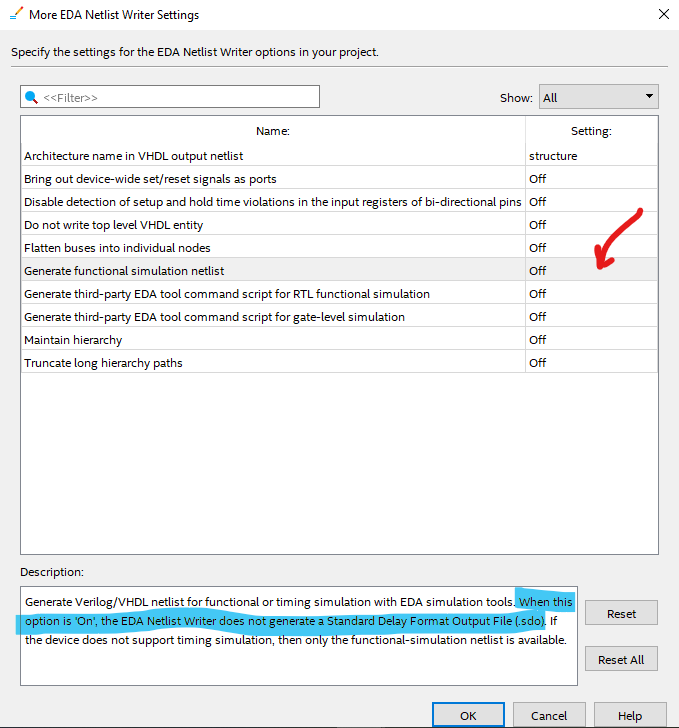
Synthesis and Post-Synthesis Simulations

Synthesis

I used Quartus Prime to synthesize my circuit as wanted in the project description. Since no specific device was mentioned in the description, I chose the following:



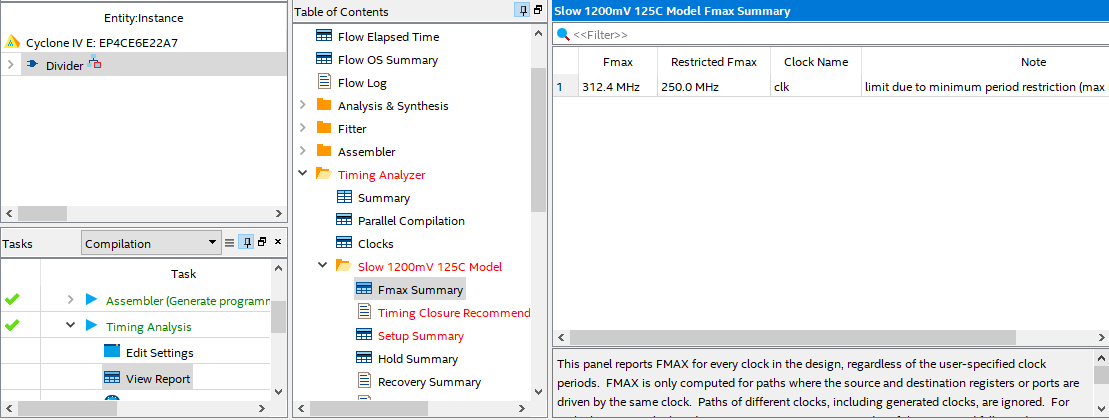
This will synthesize the description due to pinout of the device named “EP4CE6E22A7”, using Cyclone IV E engine. To make sure the result includes the timing file, I toggled the drop-down box shown in the next page.



The setting is accessible in the settings, EDA Tool Setting, Simulation, More EDA Netlist Writer Settings. Once the synthesis was over, I took two of the output files for simulations, which are “Divider.svo” and “Divider\_v.sdf”. Simulation result of the generated netlist is shown in the next page.[[3]](#footnote-3)

Netlist Simulations: Timing Challenges

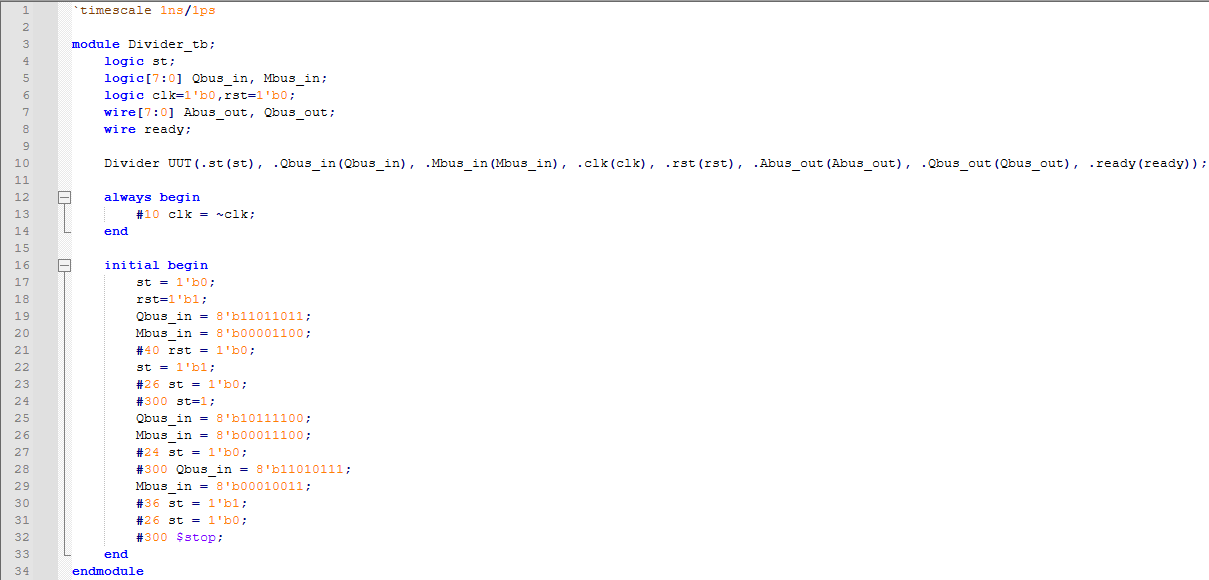
I could not use the testbench I used for the original description, due to its fast clock and propagations. According to this image:



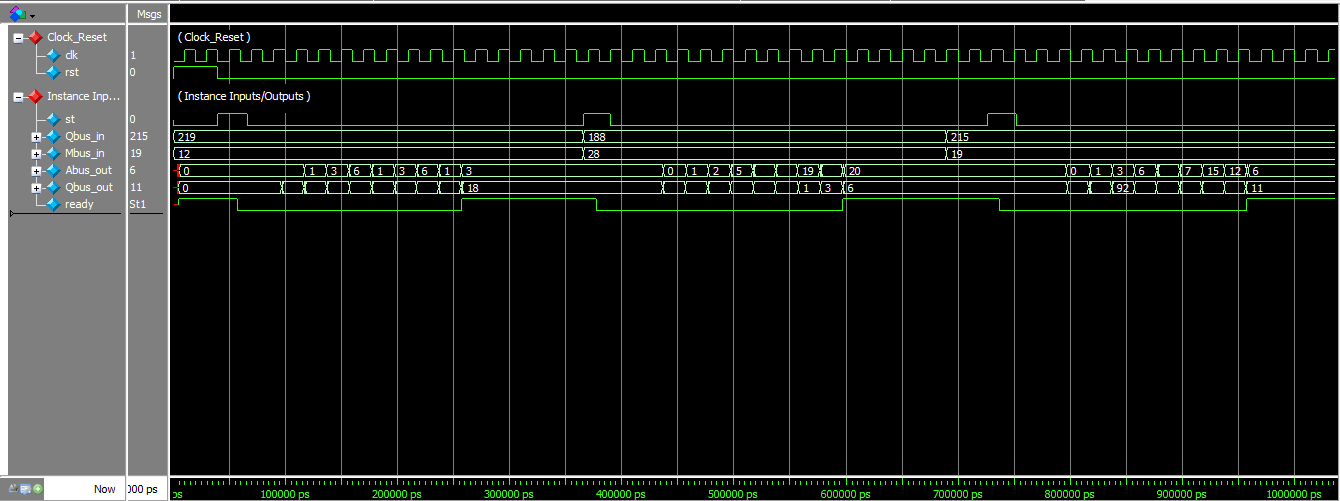
The limit on the clock is , and is restricted. I changed my clock’s period from to , leading to the frequency of , which is significantly less than the maximum allowed frequency. I also changed other propagations’ timings to match the clock and make sure at least one clock pulse sees each of them and while I was at it, I added one more set of inputs just to make sure the divider works fine. The output files were taken to ModelSim and simulated and its resulting waveforms are shown in the next page.

Netlist Simulations: Testbench and Waveforms

I used the testbench bellow to simulate the testbench.

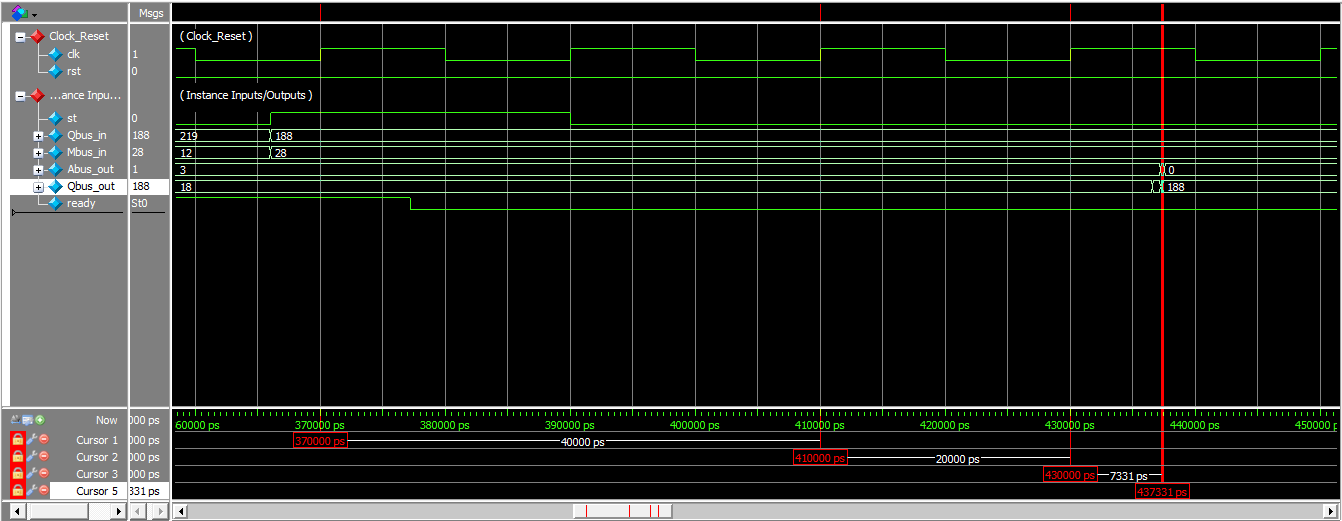


There is one thing here I need to discuss, and that is the timescale: it’s set to 1ns/1ps. I needed higher resolution than 1ns for further measurements using cursors. So I used higher resolution. There is also one more thing about this testbench, and it is how I simulated it. I added two libraries “cycloneive\_ver” and “altera\_ver” to the simulation and “Divider\_v.sdo” to “UUT” instantiation to make it work. But, it was a long process. So I found the right command for it and used it for further simulations. The resulting waveforms are shown in the image bellow:

This definitely shows nothing about the delays and timings, but briefly confirms the circuit’s functionality. The following page contains images confirming the circuit’s delays, using zoomed views and cursors.

Netlist Simulations: Confirming Delays and Timings (Clock Level)

The image bellow shows the delay of the circuit very well.



The first cursor is the clock pulse that sees 1 on “st” input. This pushes the controller into **Start** state. The second cursor is the pulse that sees a 0 on “st”, pushing controller into **Loading Init** state. Therefore, the next clock pushes the inputs to the output. But as you can see, there is a delay of between the third cursor indicating the pulse, and the fourth cursor indicating when the inputs are loaded into registers. So, the circuit does have delay, and it’s based on properties of the chosen device I mentioned before.

Next thing I did was simulate the netlist, alongside the original description. The images of its results are shown in the next page, alongside the testbench I used, which is very similar to the current one.

Pre/Post Synthesis Simulations

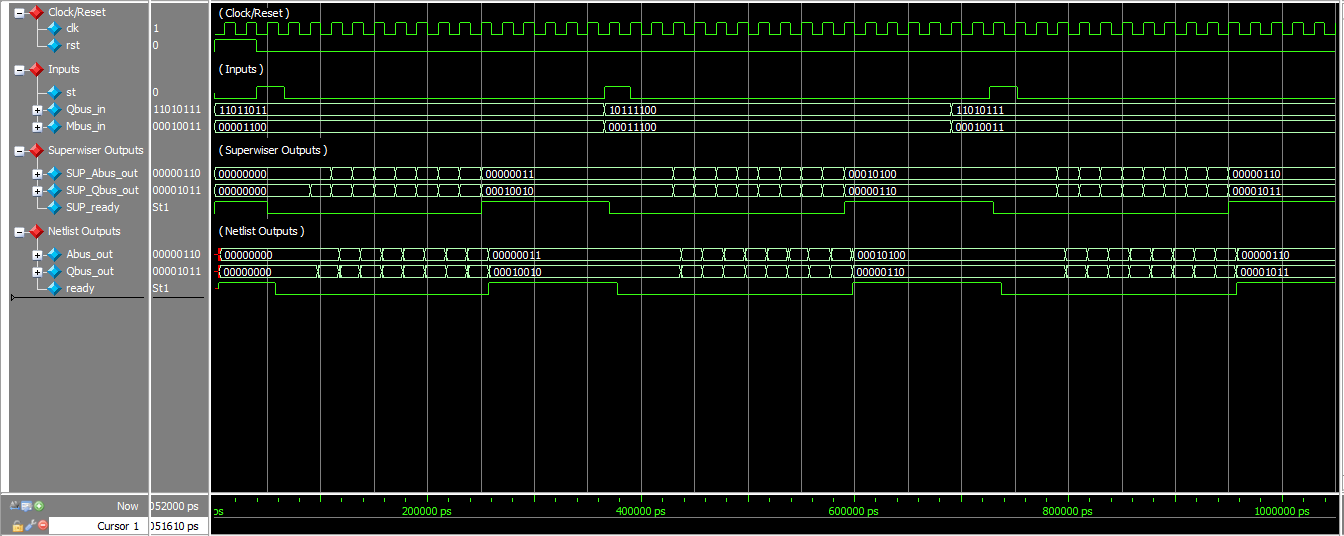
Testbench

I used the testbench bellow to test the two circuits together.

I also used the command bellow to simulate the testbench, as mentioned before:



And the image bellow shows the results:



Thankfully, the netlist’s outputs completely follow the original outputs [supervisor’s outputs], only with some delay. this means that the circuit is expected to work fine, if implemented on the mentioned PLD.

THE END

1. Appling the smallest logic on these causes the circuit’s timing to become impossible to analyse. [↑](#footnote-ref-1)
2. I did not design a wrapper for this divider because wrappers need to be customized for the system they are used in and this assignment is all about a single component, which could be use in various systems. [↑](#footnote-ref-2)
3. It is only the generated netlist. Later on, I had shown the simulation result of the netlist, alongside the original description for comparison and delay analysis. [↑](#footnote-ref-3)