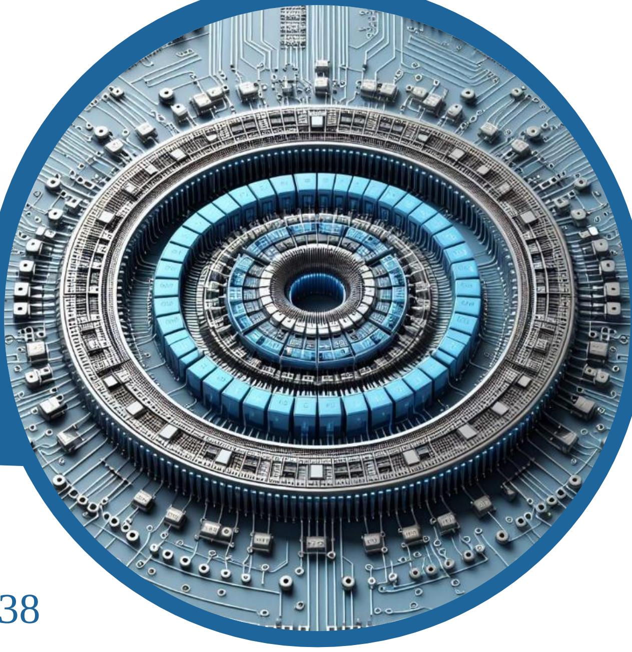
Computer Assignment 4

Flip-Flops, Registers, Shifters

May 2024



Amirreza Nadi Chaghadari | 810101538

Digital systems

Dr. Navabi

Table of Contents

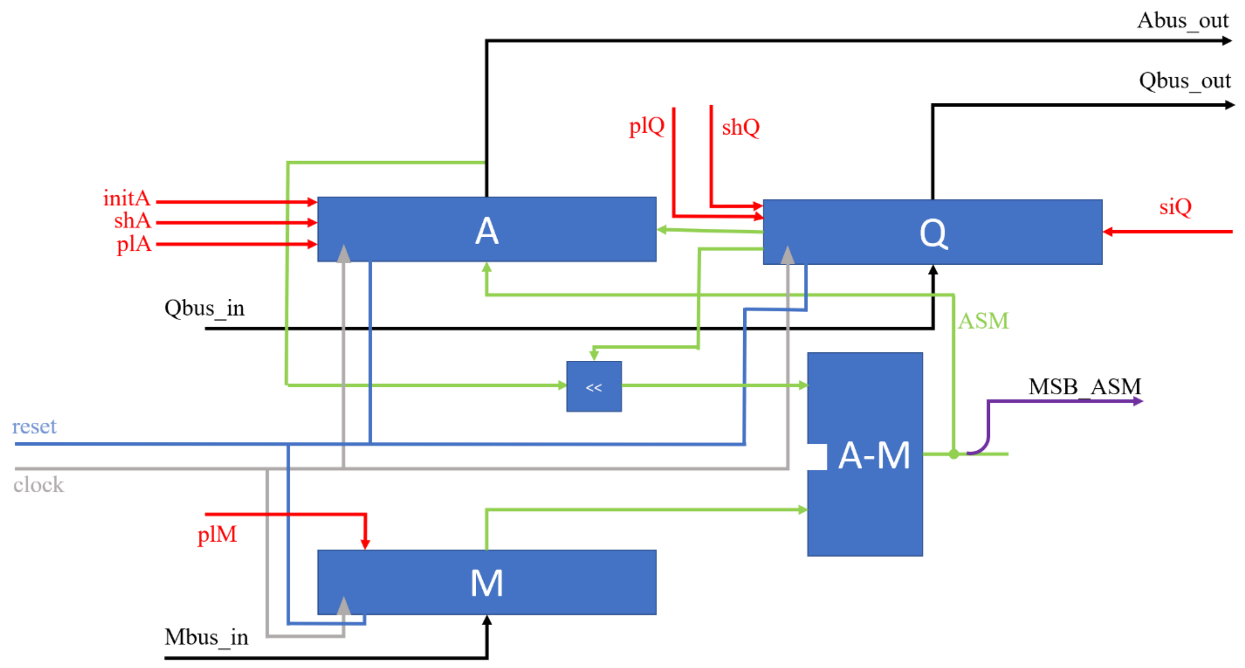
Datapath

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:

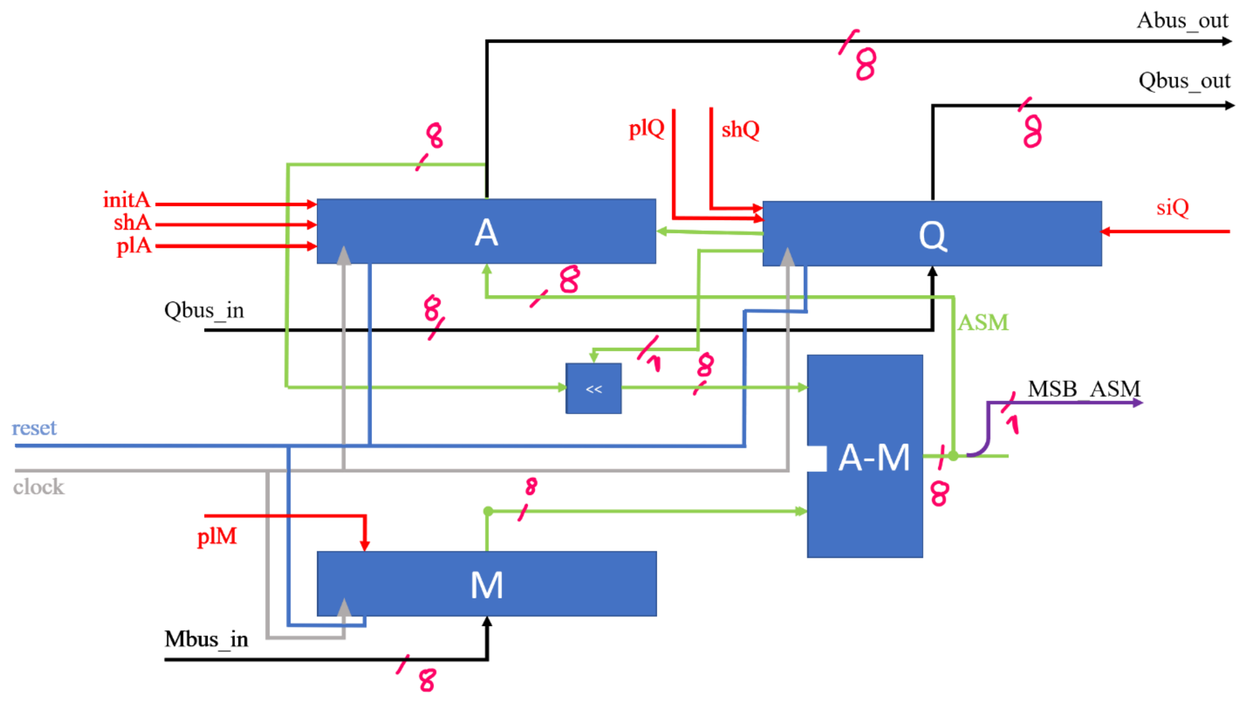


Green wires carry signals between components inside the datapath. Red wires are for controlling signals, issued by the controller. Light blue and gray wires are for reset and clock signals and therefore, not to be played with[1]! Violet wires are outputs of the datapath to the controller and finally, 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.

[1] Appling the smallest logic on these causes the circuit's timing to become impossible to analyse.

Datapath: wire arrays' sizes specified

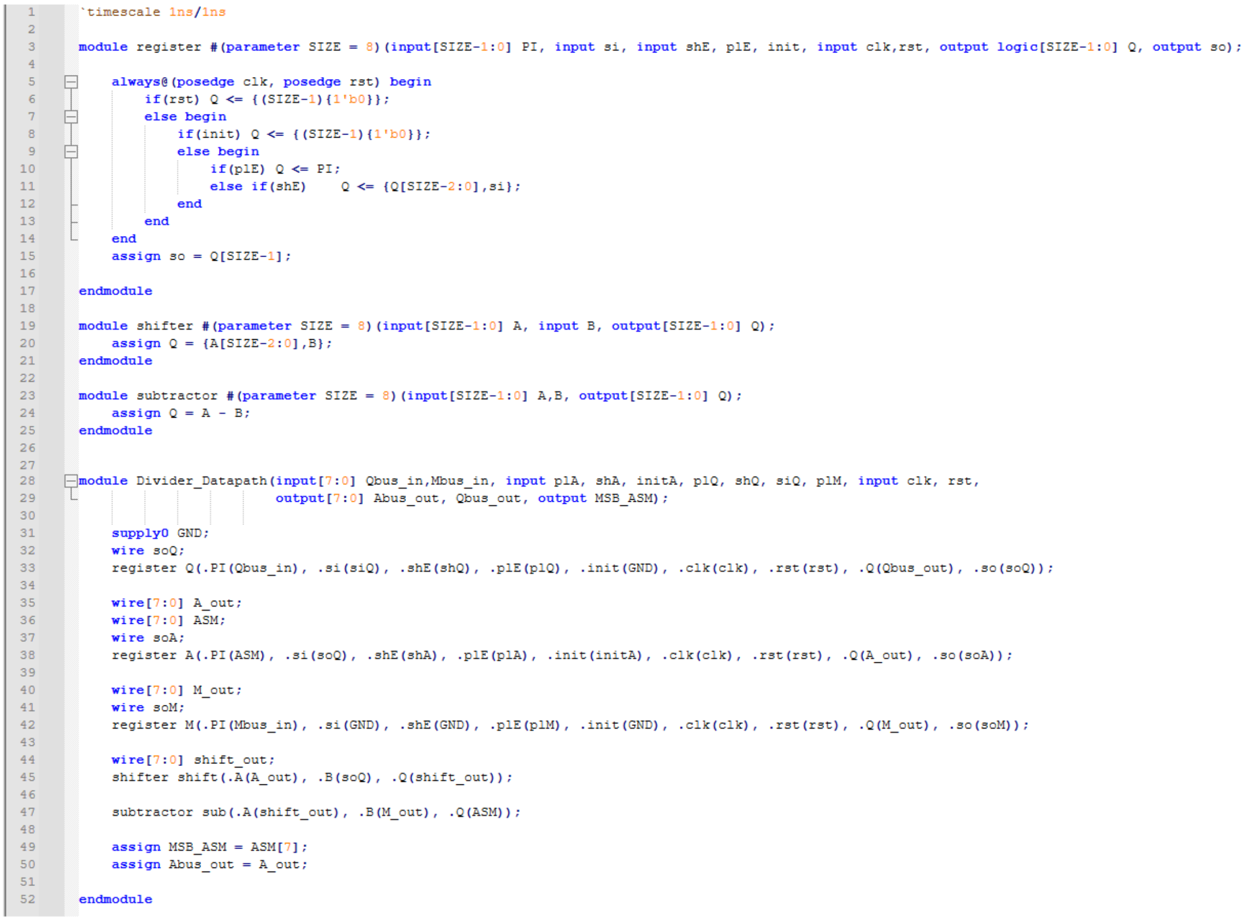


Control signals are always single wires, since I have decided to use one-hot method. Therefore, their wire count is not shown in this picture.

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:



```
`timescale 1ns/1ns

module register #(parameter SIZE = 8)(input[SIZE-1:0] PI, input si, input shE, plE, init, input clk,rst, output logic[SIZE-1:0] Q, output so);

    always@(posedge clk, posedge rst) begin
        if(rst) Q <= {(SIZE-1){1'b0}};
        else begin
            if(init) Q <= {(SIZE-1){1'b0}};
            else begin
                if(plE) Q <= PI;
                else if(shE)    Q <= {Q[SIZE-2:0],si};
            end
        end
    end
    assign so = Q[SIZE-1];

endmodule

module shifter #(parameter SIZE = 8)(input[SIZE-1:0] A, input B, output[SIZE-1:0] Q);
    assign Q = {A[SIZE-2:0],B};
endmodule

module subtractor #(parameter SIZE = 8)(input[SIZE-1:0] A,B, output[SIZE-1:0] Q);
    assign Q = A - B;
endmodule


module Divider_Datapath(input[7:0] Qbus_in,Mbus_in, input plA, shA, initA, plQ, shQ, siQ, plM, input clk, rst,
                        output[7:0] Abus_out, Qbus_out, output MSB_ASM);

    supply0 GND;
    wire soQ;
    register Q(.PI(Qbus_in), .si(siQ), .shE(shQ), .plE(plQ), .init(GND), .clk(clk), .rst(rst), .Q(Qbus_out), .so(soQ));

    wire[7:0] A_out;
    wire[7:0] ASM;
    wire soA;
    register A(.PI(ASM), .si(soQ), .shE(shA), .plE(plA), .init(initA), .clk(clk), .rst(rst), .Q(A_out), .so(soA));

    wire[7:0] M_out;
    wire soM;
    register M(.PI(Mbus_in), .si(GND), .shE(GND), .plE(plM), .init(GND), .clk(clk), .rst(rst), .Q(M_out), .so(soM));

    wire[7:0] shift_out;
    shifter shift(.A(A_out), .B(soQ), .Q(shift_out));

    subtractor sub(.A(shift_out), .B(M_out), .Q(ASM));

    assign MSB_ASM = ASM[7];
    assign Abus_out = A_out;

endmodule
```

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)

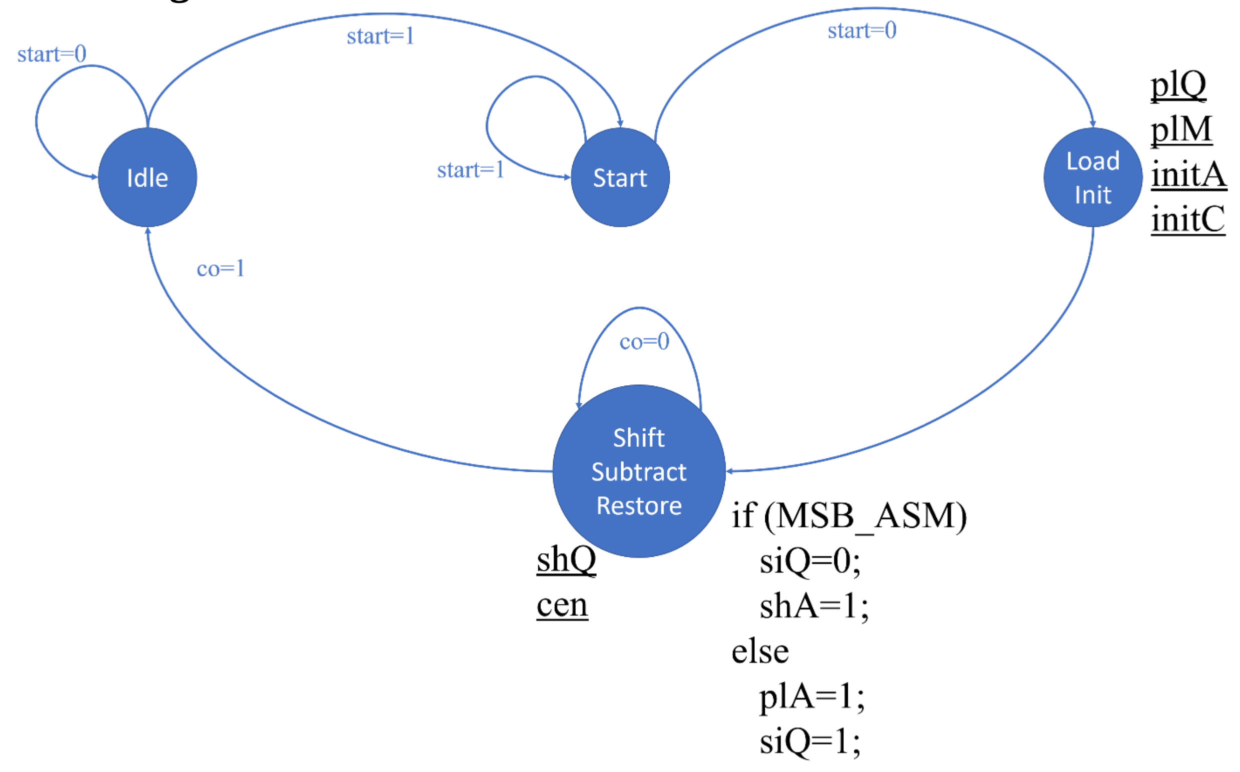
Methodology

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] 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.

[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.

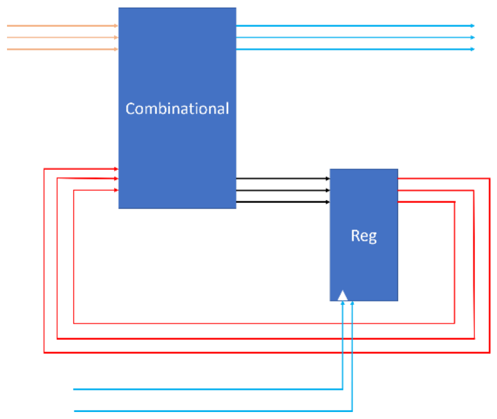
State Diagram



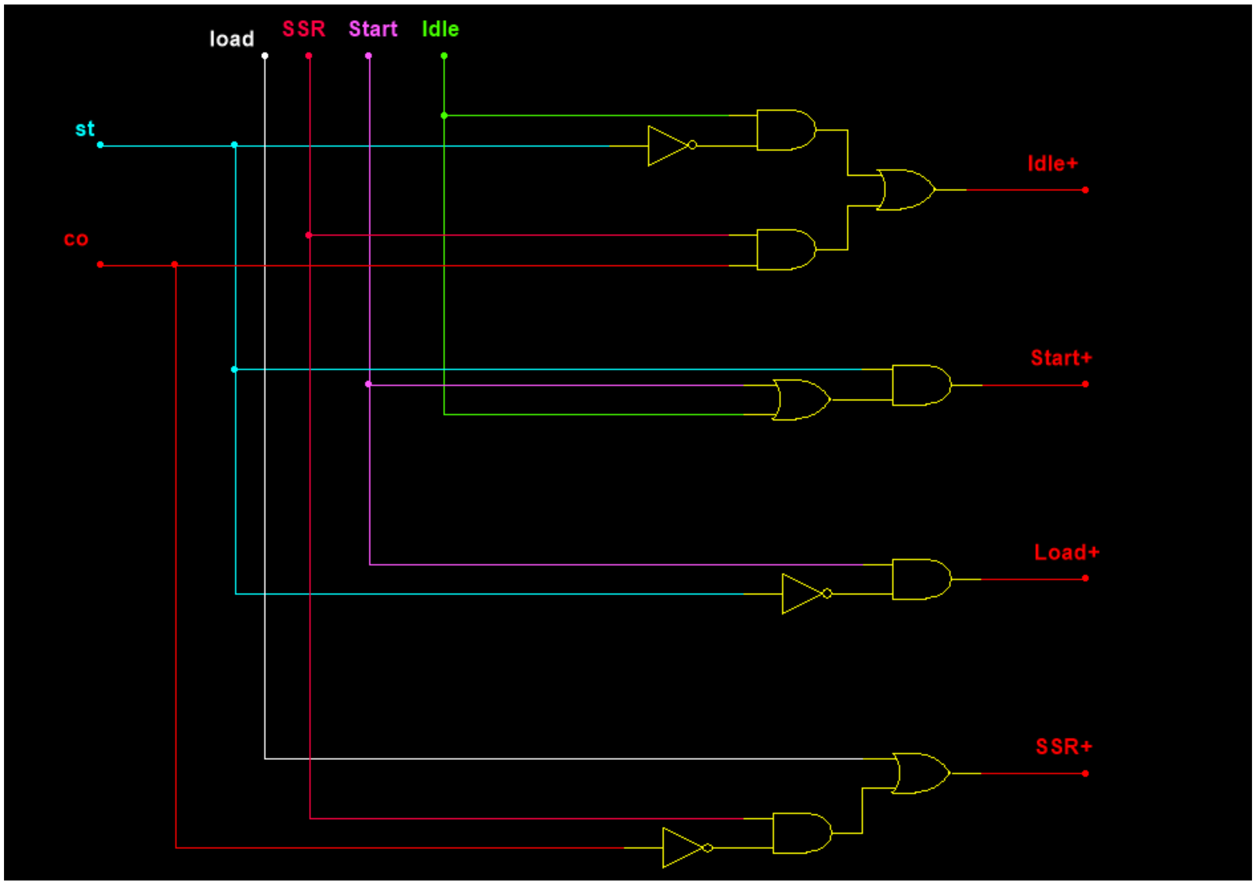
Blue arrows and texts indicate how the machine flows through its states. Black underlined signals are the ones issued in the state they're written next to and the black code next to **Shift Subtract Restore** state is what makes this machine a hybrid one, since it issues some signals based on the current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

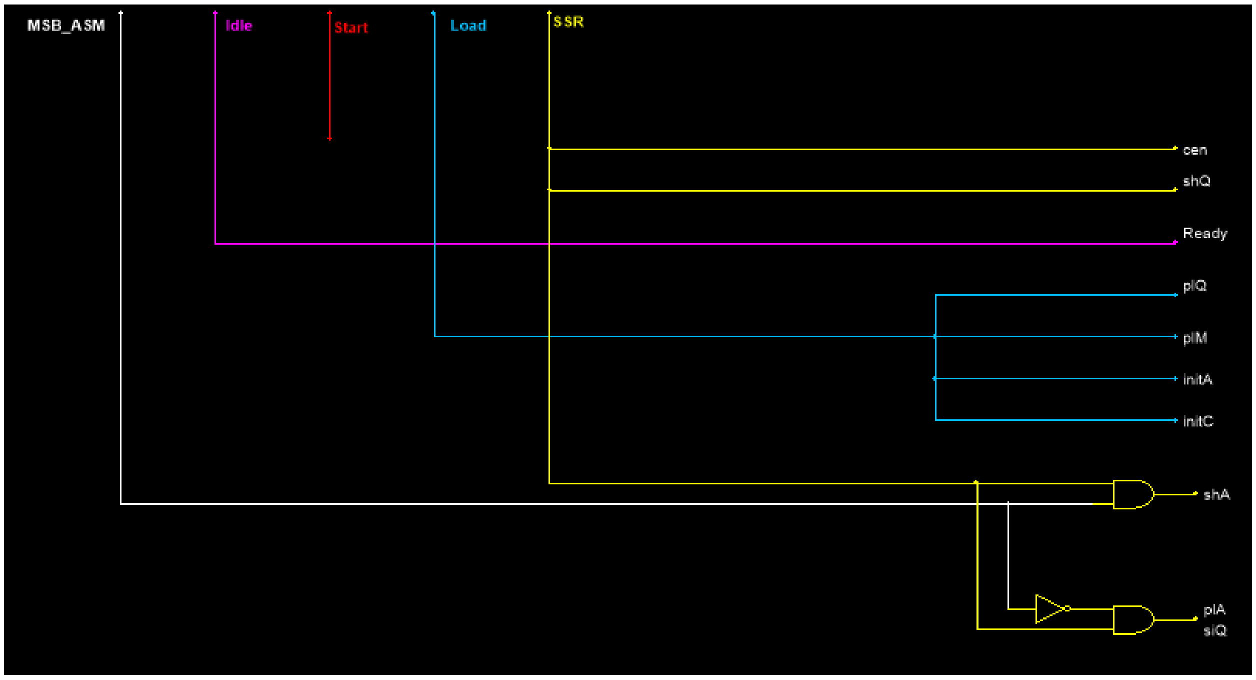


Schematic for state outputs look like this:



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.

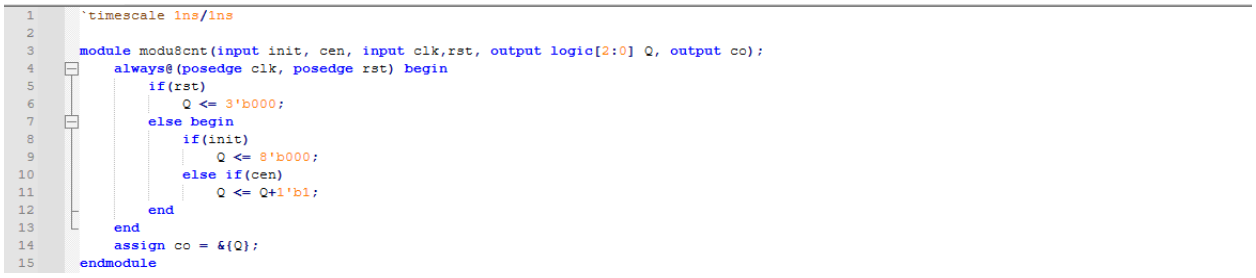
Other outputs of the controller are issued like this:



Now we are all set for the SystemVerilog description.

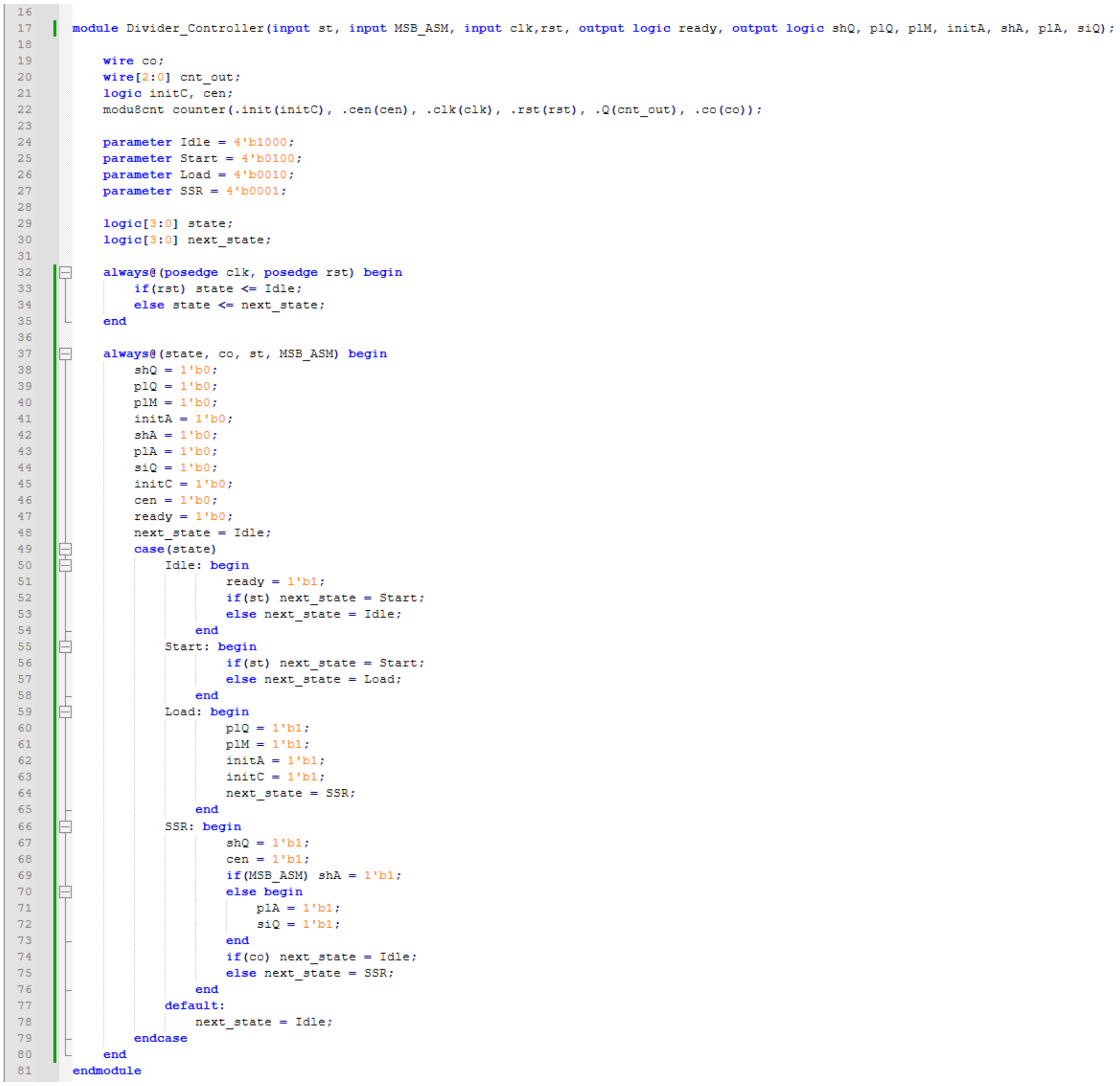
SystemVerilog Description

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:



```
`timescale 1ns/1ns

module modu8cnt(input init, cen, input clk,rst, output logic[2:0] Q, output co);
    always@(posedge clk, posedge rst) begin
        if(rst)
            Q <= 3'b000;
        else begin
            if(init)
                Q <= 8'b000;
            else if(cen)
                Q <= Q+1'b1;
        end
    end
    assign co = &{Q};
endmodule
```

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.

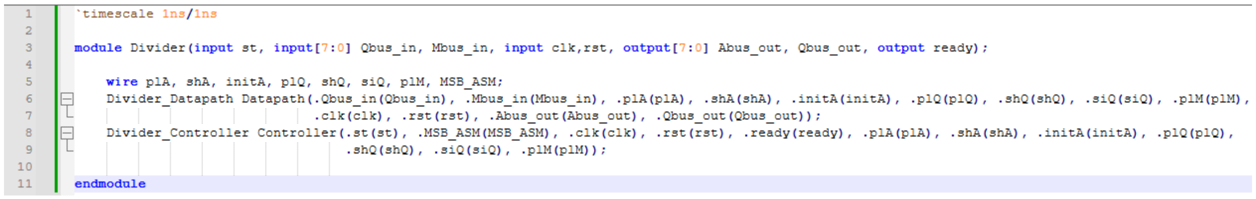


```
16
17 module Divider_Controller(input st, input MSB_ASM, input clk,rst, output logic ready, output logic shQ, plQ, plM, initA, shA, plA, siQ);
18
19     wire co;
20     wire[2:0] cnt_out;
21     logic initC, cen;
22     modu8cnt counter(.init(initC), .cen(cen), .clk(clk), .rst(rst), .Q(cnt_out), .co(co));
23
24     parameter Idle = 4'b1000;
25     parameter Start = 4'b0100;
26     parameter Load = 4'b0010;
27     parameter SSR = 4'b0001;
28
29     logic[3:0] state;
30     logic[3:0] next_state;
31
32     always@(posedge clk, posedge rst) begin
33         if(rst) state <= Idle;
34         else state <= next_state;
35     end
36
37     always@(state, co, st, MSB_ASM) begin
38         shQ = 1'b0;
39         plQ = 1'b0;
40         plM = 1'b0;
41         initA = 1'b0;
42         shA = 1'b0;
43         plA = 1'b0;
44         siQ = 1'b0;
45         initC = 1'b0;
46         cen = 1'b0;
47         ready = 1'b0;
48         next_state = Idle;
49         case(state)
50             Idle: begin
51                     ready = 1'b1;
52                     if(st) next_state = Start;
53                     else next_state = Idle;
54                 end
55             Start: begin
56                     if(st) next_state = Start;
57                     else next_state = Load;
58                 end
59             Load: begin
60                     plQ = 1'b1;
61                     plM = 1'b1;
62                     initA = 1'b1;
63                     initC = 1'b1;
64                     next_state = SSR;
65                 end
66             SSR: begin
67                     shQ = 1'b1;
68                     cen = 1'b1;
69                     if(MSB_ASM) shA = 1'b1;
70                     else begin
71                         plA = 1'b1;
72                         siQ = 1'b1;
73                     end
74                     if(co) next_state = Idle;
75                     else next_state = SSR;
76                 end
77             default:
78                 next_state = Idle;
79         endcase
80     end
81 endmodule
```

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:

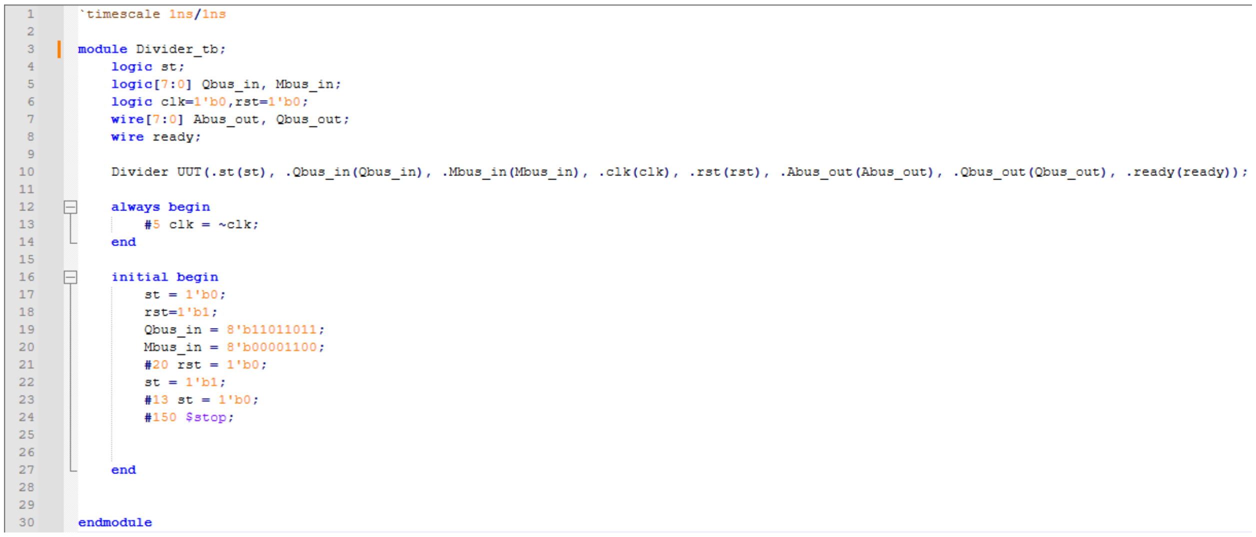


```verilog
`timescale 1ns/1ns

module Divider(input st, input[7:0] Qbus_in, Mbus_in, input clk,rst, output[7:0] Abus_out, Qbus_out, output ready);

    wire plA, shA, initA, plQ, shQ, siQ, plM, MSB_ASM;
    Divider_Datapath Datapath(.Qbus_in(Qbus_in), .Mbus_in(Mbus_in), .plA(plA), .shA(shA), .initA(initA), .plQ(plQ), .shQ(shQ), .siQ(siQ), .plM(plM),
                              .clk(clk), .rst(rst), .Abus_out(Abus_out), .Qbus_out(Qbus_out));
    Divider_Controller Controller(.st(st), .MSB_ASM(MSB_ASM), .clk(clk), .rst(rst), .ready(ready), .plA(plA), .shA(shA), .initA(initA), .plQ(plQ),
                                  .shQ(shQ), .siQ(siQ), .plM(plM));

endmodule
```

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:

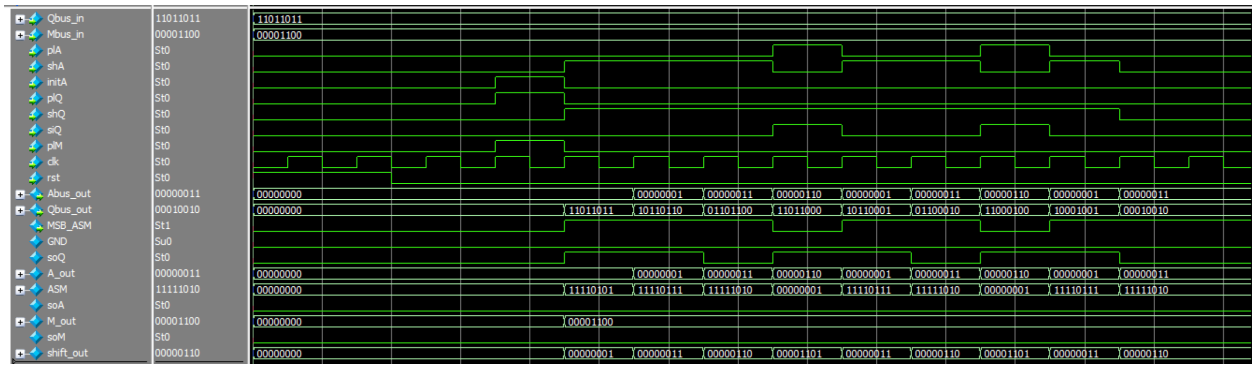


```
`timescale 1ns/1ns

module Divider_tb;
    logic st;
    logic[7:0] Qbus_in, Mbus_in;
    logic clk=1'b0,rst=1'b0;
    wire[7:0] Abus_out, Qbus_out;
    wire ready;

    Divider UUT(.st(st), .Qbus_in(Qbus_in), .Mbus_in(Mbus_in), .clk(clk), .rst(rst), .Abus_out(Abus_out), .Qbus_out(Qbus_out), .ready(ready));

    always begin
        #5 clk = ~clk;
    end

    initial begin
        st = 1'b0;
        rst=1'b1;
        Qbus_in = 8'b11011011;
        Mbus_in = 8'b00001100;
        #20 rst = 1'b0;
        st = 1'b1;
        #13 st = 1'b0;
        #150 $stop;


    end


endmodule
```

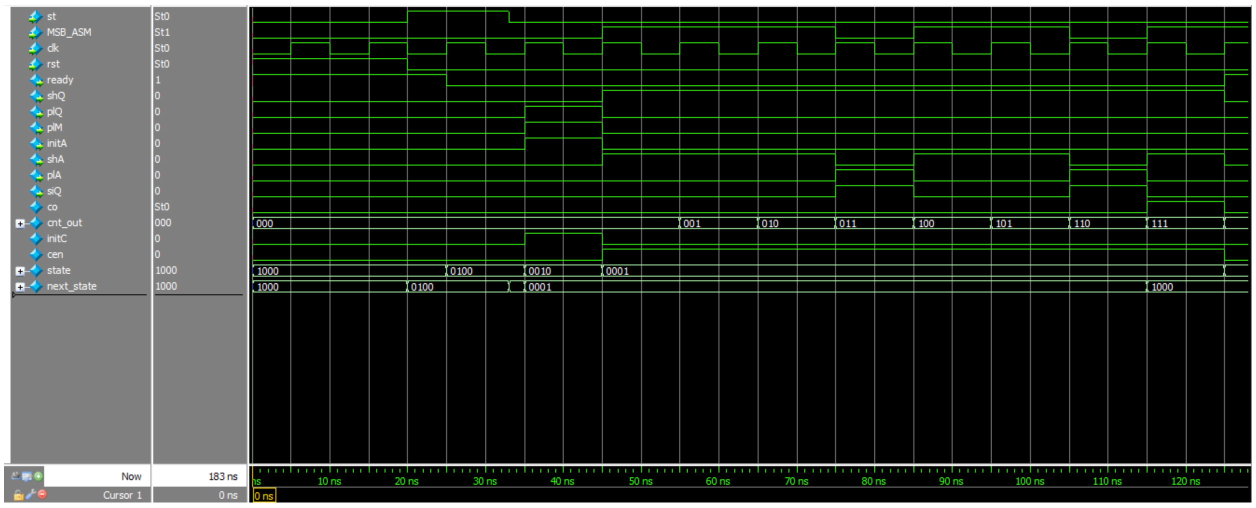
The waveforms are shown in the next page.

Partial Waveforms

Datapath’s waveform looks like this:

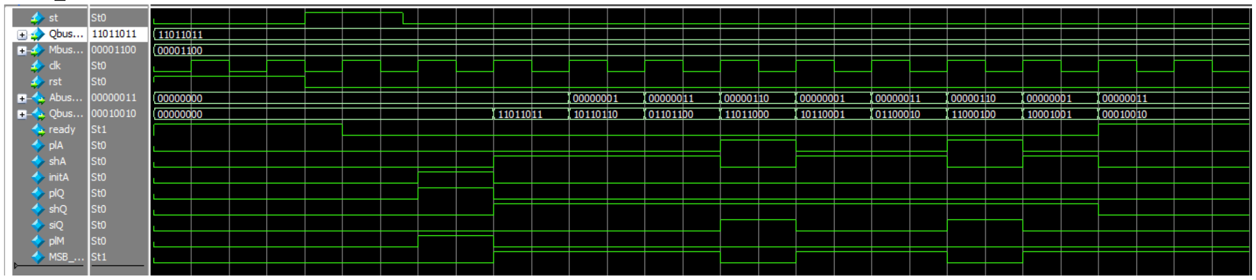


Controller’s waveform looks like this:

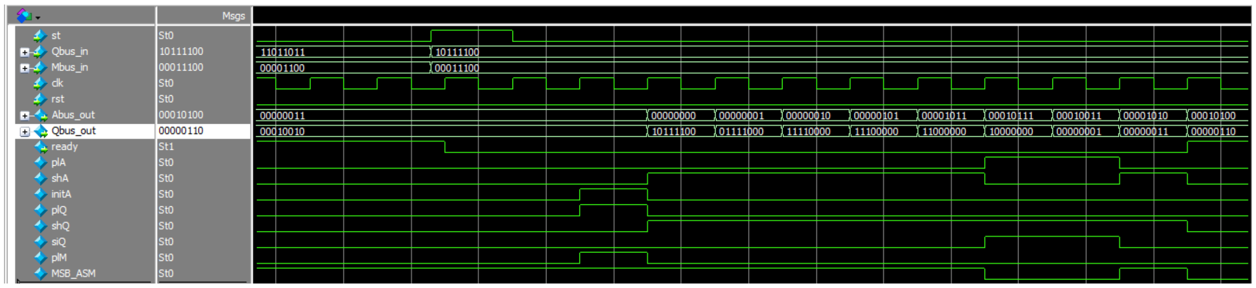


The top module’s waveforms are shown in the next page.

Top Module Waveforms



Inputs are 11011011 ÷ 00001100, which translate to 219 ÷ 12. The quotient is 18 and the remainder is 3. Here's another waveform for another set of inputs:



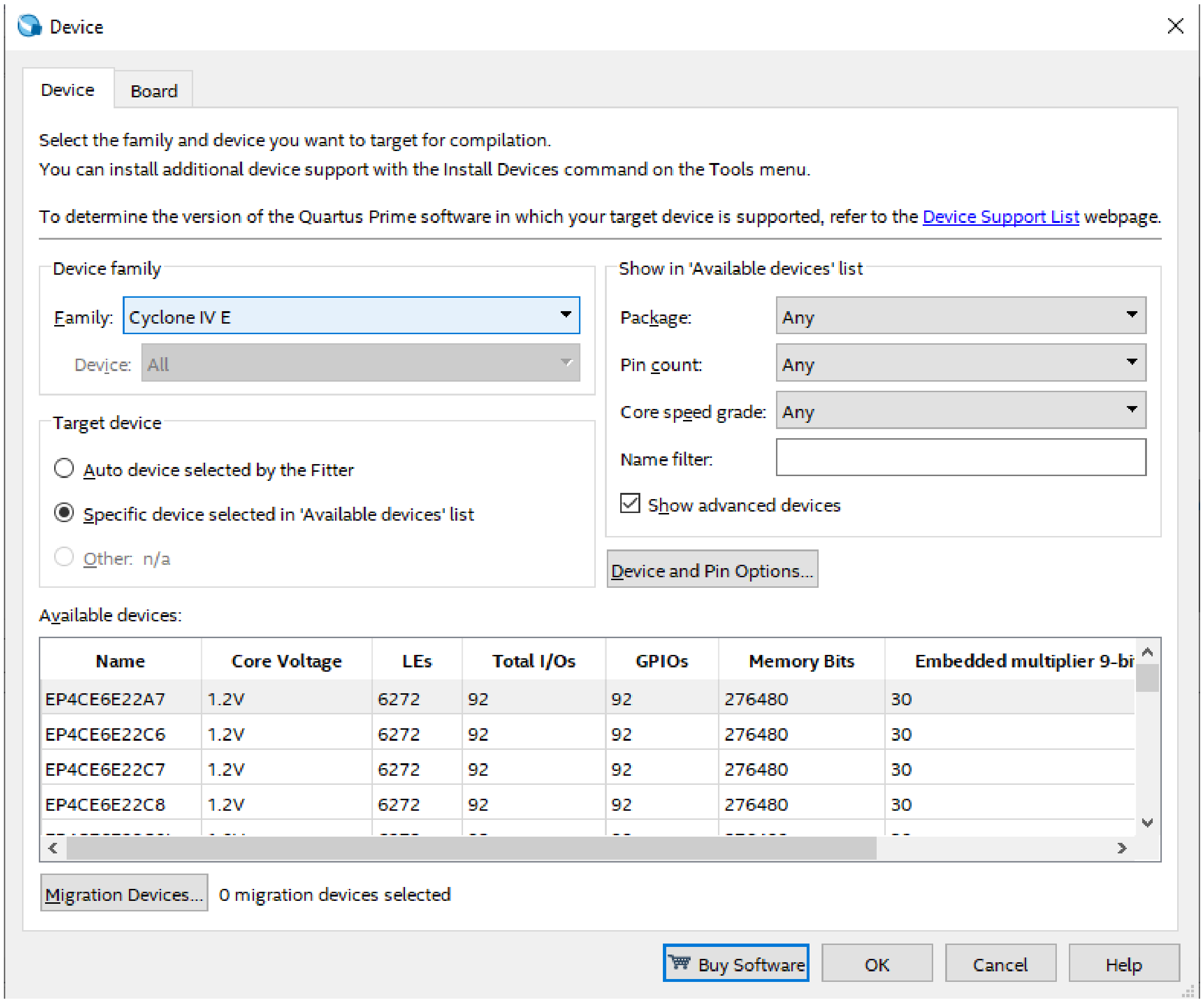
The division is 10111100 ÷ 00011100, which translate to 188 ÷ 28. The quotient is 6 and the remainder is 20, which confirm the waveforms.

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:



Select the family and device you want to target for compilation.
You can install additional device support with the Install Devices command on the Tools menu.

To determine the version of the Quartus Prime software in which your target device is supported, refer to the Device Support List webpage.

Device family
Family: Cyclone IV E
Device: All

Target device
( ) Auto device selected by the Fitter
(●) Specific device selected in 'Available devices' list
( ) Other: n/a

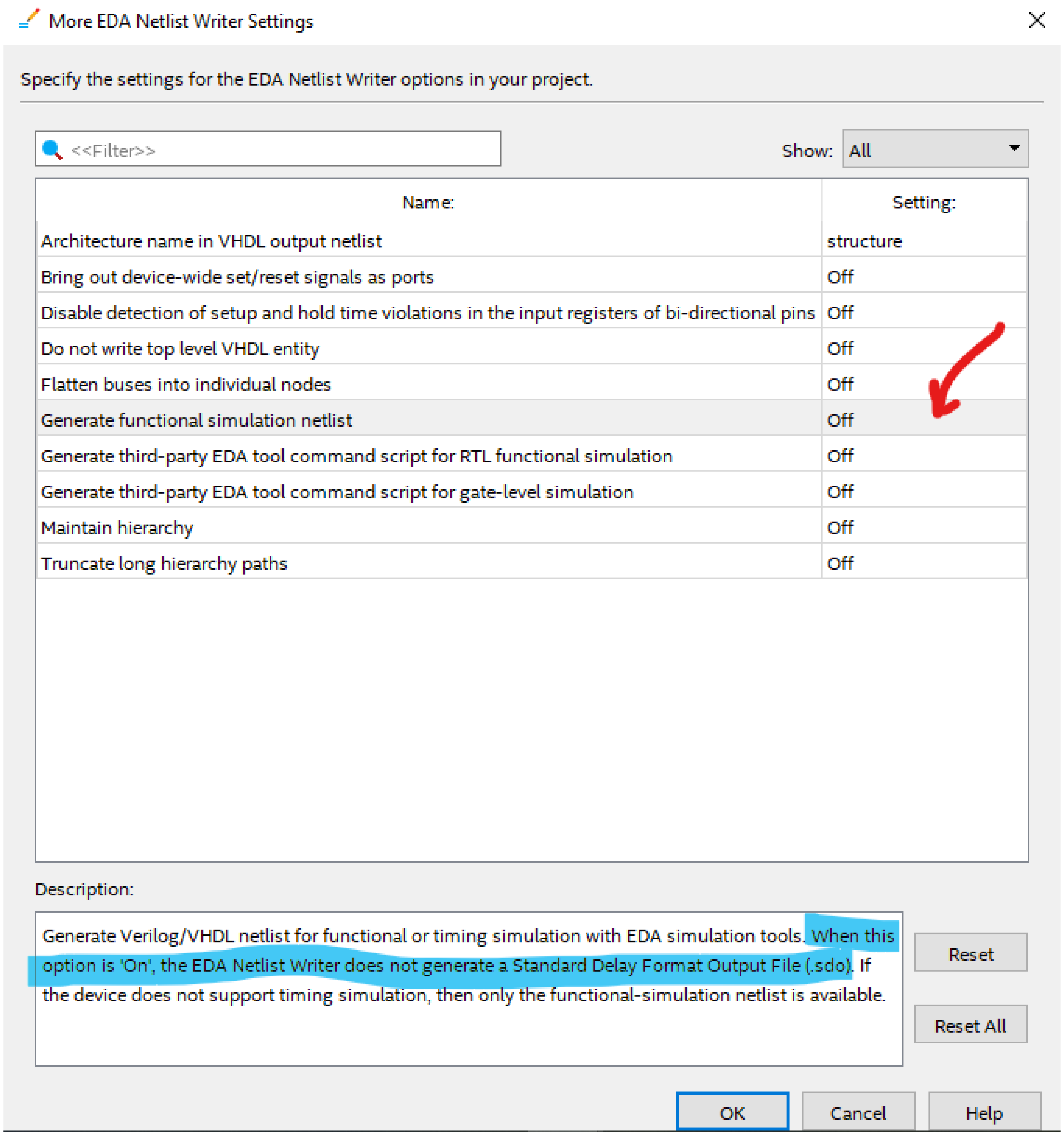
Show in 'Available devices' list
Package: Any
Pin count: Any
Core speed grade: Any
Name filter:
☑ Show advanced devices

Available devices:

Name	Core Voltage	LEs	Total I/Os	GPIOs	Memory Bits	Embedded multiplier 9-bi
EP4CE6E22A7	1.2V	6272	92	92	276480	30
EP4CE6E22C6	1.2V	6272	92	92	276480	30
EP4CE6E22C7	1.2V	6272	92	92	276480	30
EP4CE6E22C8	1.2V	6272	92	92	276480	30

0 migration devices selected

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.



More EDA Netlist Writer Settings

Specify the settings for the EDA Netlist Writer options in your project.

Name:	Setting:
Architecture name in VHDL output netlist	structure
Bring out device-wide set/reset signals as ports	Off
Disable detection of setup and hold time violations in the input registers of bi-directional pins	Off
Do not write top level VHDL entity	Off
Flatten buses into individual nodes	Off
Generate functional simulation netlist	Off
Generate third-party EDA tool command script for RTL functional simulation	Off
Generate third-party EDA tool command script for gate-level simulation	Off
Maintain hierarchy	Off
Truncate long hierarchy paths	Off

Description:

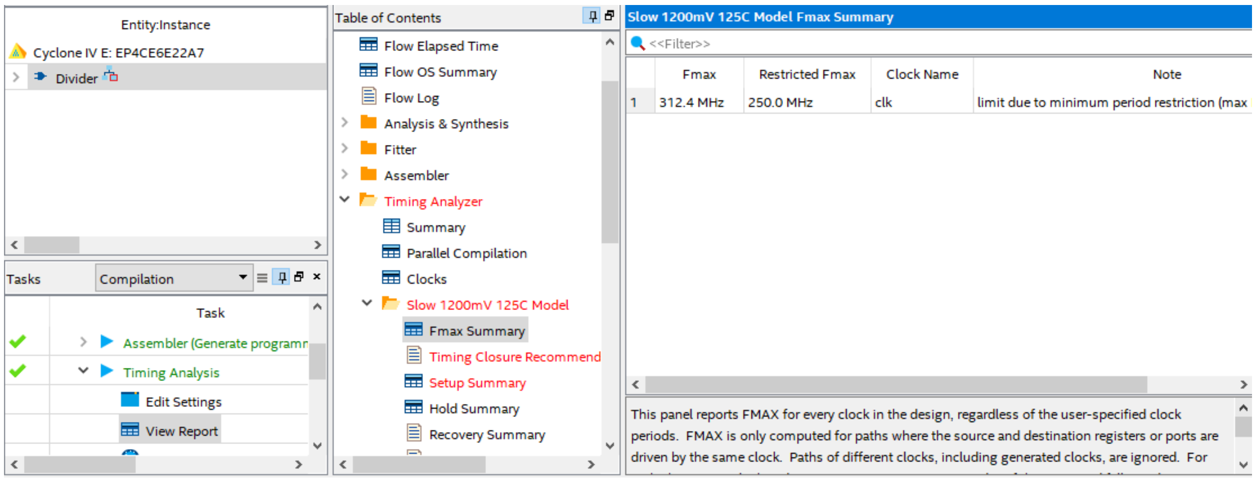
Generate Verilog/VHDL netlist for functional or timing simulation with EDA simulation tools. When this option is 'On', the EDA Netlist Writer does not generate a Standard Delay Format Output File (.sdo). If the device does not support timing simulation, then only the functional-simulation netlist is available.

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]

[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.

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:



Entity:Instance

Cyclone IV E: EP4CE6E22A7

Divider

Tasks | Compilation

Task

Assembler (Generate programn

Timing Analysis

Edit Settings

View Report

Table of Contents

Flow Elapsed Time

Flow OS Summary

Flow Log

Analysis & Synthesis

Fitter

Assembler

Timing Analyzer

Summary

Parallel Compilation

Clocks

Slow 1200mV 125C Model

Fmax Summary

Timing Closure Recommend

Setup Summary

Hold Summary

Recovery Summary

Slow 1200mV 125C Model Fmax Summary

<<Filter>>

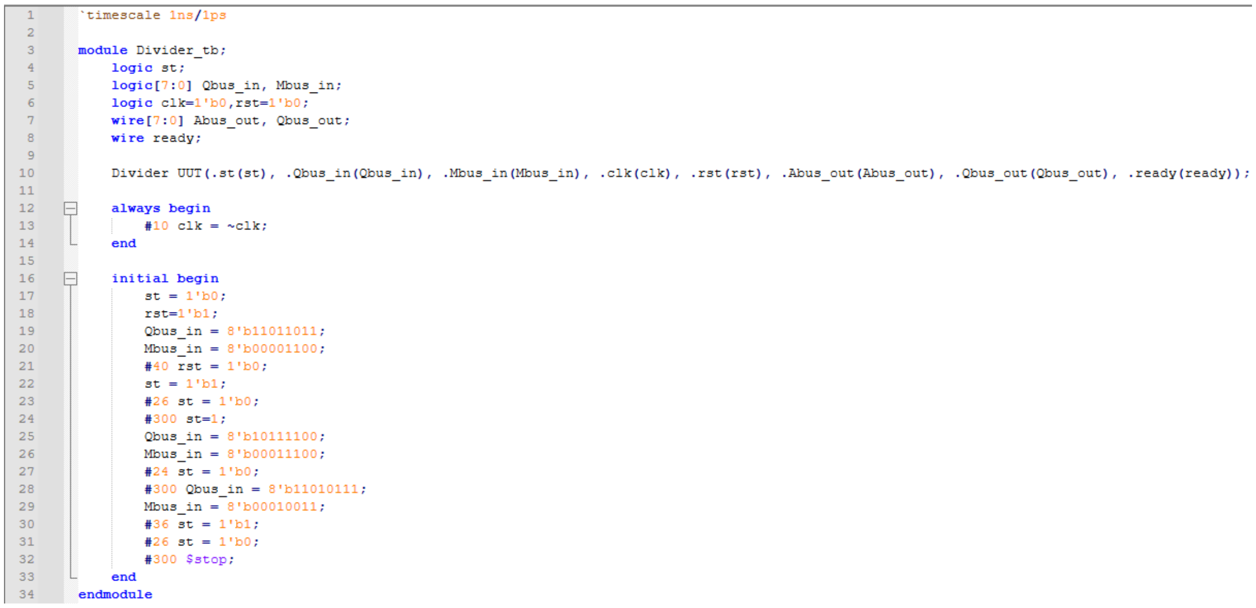
	Fmax	Restricted Fmax	Clock Name	Note
1	312.4 MHz	250.0 MHz	clk	limit due to minimum period restriction (max

This panel reports FMAX for every clock in the design, regardless of the user-specified clock periods. FMAX is only computed for paths where the source and destination registers or ports are driven by the same clock. Paths of different clocks, including generated clocks, are ignored. For

The limit on the clock is $312MHz$, and $250MHz$ is restricted. I changed my clock's period from $10ns$ to $20ns$, leading to the frequency of $\frac{1}{20\times10^{-9}} = 5 \times 10^7 = 50MHz$, 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.

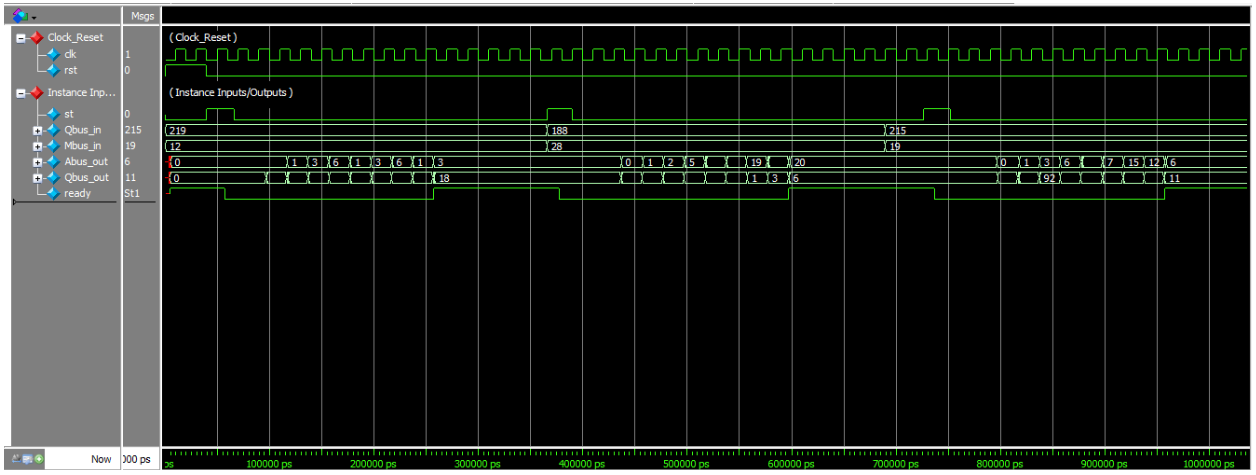
Netlist Simulations: Testbench and Waveforms

I used the testbench bellow to simulate the testbench.



```
`timescale 1ns/1ps

module Divider_tb;
    logic st;
    logic[7:0] Qbus_in, Mbus_in;
    logic clk=1'b0,rst=1'b0;
    wire[7:0] Abus_out, Qbus_out;
    wire ready;

    Divider UUT(.st(st), .Qbus_in(Qbus_in), .Mbus_in(Mbus_in), .clk(clk), .rst(rst), .Abus_out(Abus_out), .Qbus_out(Qbus_out), .ready(ready));

    always begin
        #10 clk = ~clk;
    end

    initial begin
        st = 1'b0;
        rst=1'b1;
        Qbus_in = 8'b11011011;
        Mbus_in = 8'b00001100;
        #40 rst = 1'b0;
        st = 1'b1;
        #26 st = 1'b0;
        #300 st=1;
        Qbus_in = 8'b10111100;
        Mbus_in = 8'b00011100;
        #24 st = 1'b0;
        #300 Qbus_in = 8'b11010111;
        Mbus_in = 8'b00010011;
        #36 st = 1'b1;
        #26 st = 1'b0;
        #300 $stop;
    end
endmodule
```

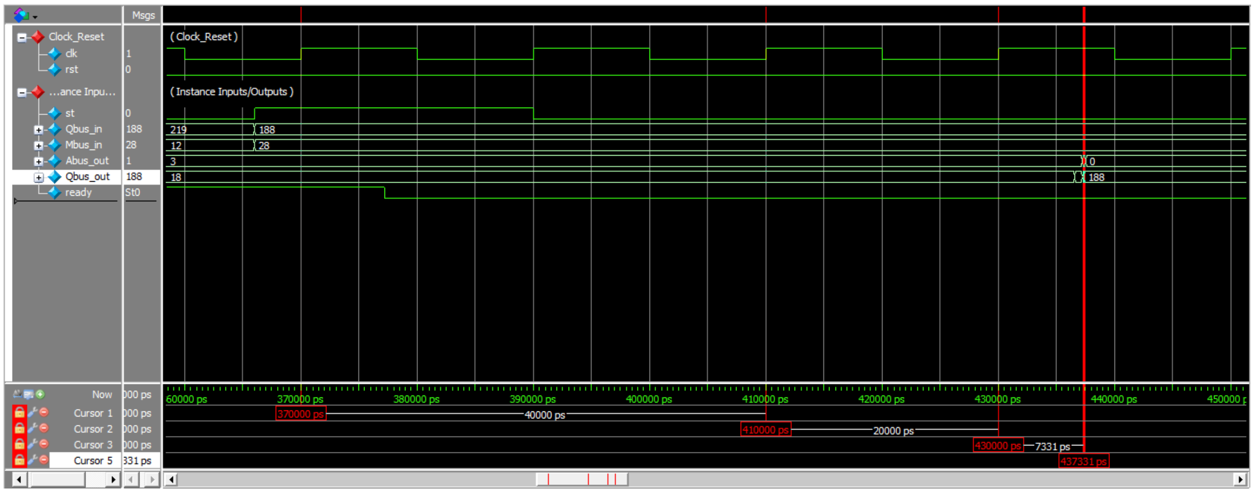
There is one thing here I need to discuss, and that is the timescale: it's set to 1ns/1ps. I needed higher resolution than 1ns for further measurements using cursors. So I used higher resolution. There is also one more thing about this testbench, and it is how I simulated it. I added two libraries "cycloneive_ver" 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 $7331ps$ 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.

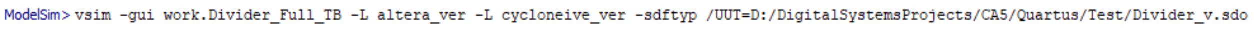
Next thing I did was simulate the netlist, alongside the original description. The images of its results are shown in the next page, alongside the testbench I used, which is very similar to the current one.

Pre/Post Synthesis Simulations

Testbench

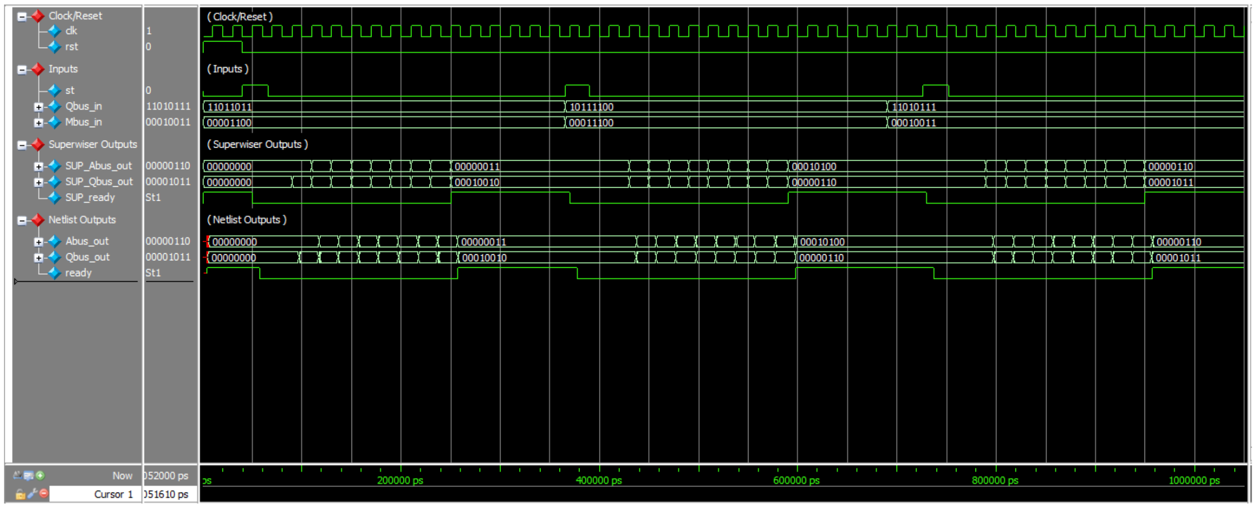
I used the testbench bellow to test the two circuits together.

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



```
ModelSim> vsim -gui work.Divider_Full_TB -L altera_ver -L cycloneive_ver -sdftyp /UUT=D:/DigitalSystemsProjects/CA5/Quartus/Test/Divider_v.sdo
```

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