Garrett Tashiro

EE 371

November 19, 2021

Lab 4 Report

Procedure

Lab four was comprised of two tasks: the first task was to implement a bit counter from an ASMD chart that was given and make the control and datapath for the bit counter. The bit counter would take in an 8-bit data and count the number of 1's that was in the data. The second task was to design an ASMD chart, control, and datapath for the binary search algorithm. For this task, we had to create a 32x8 RAM module and search the RAM module for an 8-bit piece of data using the control logic and datapath that we created from our ASMD chart. If the data was found, we would display the address on at which the data was at in the RAM on HEX1 and HEX0, and LEDR[9] would light up. If the value was not found, not address would display on the HEX displays, and LEDR[8] would light up. This lab gave me an understanding about how to create an ASMD chart, and translate the chart into control and datapath logic.

Task #1

The first task we were given was to implement a bit counter algorithm in SystemVerilog from an ASMD chart that was given to us on the lab document. We had to create the control logic and datapath for the ASMD chart, and count the number of bits that were 1 in an 8-bit piece of data that we put into the system. For the given ASMD chart I changed a couple of things to make the chart a little bit more descriptive of what my code would be. My ASMD chart can be seen below in Figure 1.

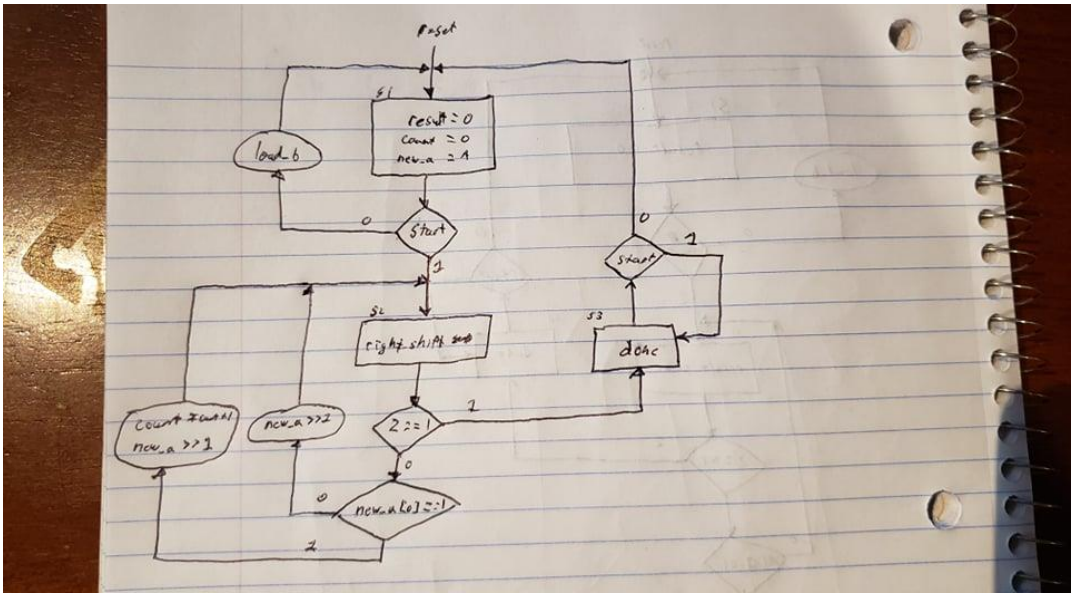

Fig. 1: ASMD Chart for Bit Counting

The way I approached this first task was first by reading over the information in the “Introduction” section of the lab doc, as well as reviewing the lecture slides from class. I wanted to make sure I had a clear understanding of how an ASMD chart worked, and what parts of the chart would be control logic, and what parts were going to be datapath logic. Once I had an understanding of how an ASMD chart worked, and how to translate it to code, I read over task 1. I knew that I was going to need to have a module for control, datapath, and one for HEX display. I first started by translating the ASMD chart into an FSM in my control module. I wanted to have three states based on the three rectangles in the ASMD chart, so I created the three states and named them s1, s2, s3. I knew that I needed to have flags as outputs that would tell the datapath what to do based upon what state the FSM was in. The assignment of the flags can be seen below in Figure 2.

```
////Assignment of outputs////

//result_shift is 1 if in s2 state
assign result_shift = (ps == s2);

//done is 1 if in s3 state
assign done = (ps == s3);

//load_b is 1 if in s1 state and start is 0
assign load_b = ((ps == s1) && !start) ? 1'b1 : 1'b0;
```

Fig. 2: Assignment of Flags in Control

With the assignments, I knew that each one was going to be different based upon what state the FSM was in. I had load_b only be set to 1’b1 if the FSM was in state s1, and if the 1-bit input signal start was set to 0. This would tell the datapath to not cycle through the 8-bit input data yet, and keep updating to what the 8-bit input data was each clock cycle, as well as setting the count to 0. The value of load_b was checked in an if statement along with the value of reset and was inside of an always_ff block in the datapath_BC. Once start is set to 1, the FSM will transition to s2, which then sets result_shift to 1. If result_shift was 1, then the datapath would check to see if the less significant bit of the temp logic, which was originally set to the 8-bit data that was brought in, to see if the value was a 1, or a 0. If the bit is a 1, then count would increment by one. After that, the 8-bit temp logic would be bit shifted to the right by 1 bit. The code for this always_ff can be seen below in Figure 3, and the full code for datapath_BC.sv can be found in Appendix 1.C.

```
always_ff @(posedge clk) begin
    if(reset || load_b) begin
        count <= 0;
        new_a <= A;
    end

    else if(result_shift) begin
        if(new_a[0] == 1) begin
            count <= count + 4'd1;
        end
        new_a <= new_a >> 1;
    end
end
```

Fig. 3: Datapath Logic for Bit Counter

After I had the control and datapath done I moved onto creating the module for the hex_display. This module was fairly simple, and I actually had most of the code from previous labs. I just changed a couple variable names, as well as deleted a couple of things and it was finished. The code for the hex_display module can be found in Appendix 1.E. With the hex_display module finished I moved onto the DE1_SoC

module. Since there would be inputs from switches and a key, I knew that I was going to need a double DFF module, so I went to my lab two files and got my paramDFF.sv file. This was to prevent metastability on the signals from the inputs. The full code for paramDFF can be found in Appendix 3.A since this module is used in both tasks. The DE1_SoC was a fairly simple module to do, although I did have an issue at first with timing from the inputs to the module when resetting the system. The inputs would reset a cycle late, and I fixed this by just putting 1'b0 as the input to all the resets instead of passing reset into reset. The full code for the DE1_SoC module can be seen in Appendix 1.G.

I reason as to why I setup my code the way I did is because it was because it made the most sense. I followed the ASMD chart, which was fairly easy to follow and translate to code once I knew how to do so. I chose to have the control and datapath be separate modules so I didn't get confused with the where the different variables were needed. I chose to split those modules and just pass flags from one module to another so they worked together. The 1-bit logics load_b, result_shift, and done were outputs from the control module and passed as inputs to the datapath module. These inputs would assist the datapath in doing what it needed to do. The 1-bit output z in datapath was passed into the control module in order to tell the control that the data that was passed, checked for 1's and shifted, was now 0'ed out and the FSM would transition into the done state. 1-bit output done logic in the control module would be set to 1 once in this state, and the value is passed to datapath in order to set the 4-bit value of result, which is the number of 1's that were in the data that was passed in by the user using switches. The hex_display module was setup using an always_comb block, and had 4-bit result passed in from the datapath module. This value would determine what was displayed on HEX0. For the DE1_SoC, I hierarchically called hex_display, paramDFF, bitCounter, and datapath_BC and connected the ports to one another, and tested to make sure that the correct values were being passed into each module, that the outputs were updating correctly, and that the FSM was transitioning through its state correctly.

Task #2

For the second task, we had to create an ASMD chart for the binary search algorithm, and then create the control logic and datapath for the binary search algorithm that would search a 32x8 RAM module for a desired piece of data. How I did this was I first started by reading over task 2 to see what was being asked of me. I then watching a video on YouTube as a refresher to see how the binary search algorithm worked as I didn't fully remember exactly how the algorithm worked. Once I knew how the algorithm worked, I went onto drawing my ASMD chart. I figured that I could get the whole system, and all the logic in three states. I Followed along with how the ASMD chart from task 1 was done. I start with drawing the first state, and setting values for my 5-bit values front, mid, and last all to 0. I draw a line down to a diamond that was for 1-bit input start. If start was 0, then 1-bit output loads would be 1. If start was 1, then system would move onto the second state, s2. While in s2 I knew that I first wanted to check if 8-bit target, the data value at the address in the RAM at which mid was pointing to, was equal to 8-bit A, the desired data being searched for. I drew a diamond with this test right below the rectangle for the second state. If the two values were equal, then a line goes from the diamond to the third state, s3, and 1-bit logic is updated to 1. If this test is false, then a line goes to the next diamond/test. I next knew that I wanted to check to see if 5-bit logic front was greater than, or equal to 5-bit logic last. If front is greater than, or equal to then a line goes from the diamond that holds this test to an oval that has 1-bit not_found set to 1, and then over to s3. If this test was false, then a line is drawn to the next diamond for the third test. For the third test I checked if 8-bit value A was greater than 8-bit value target. If this was true, then you leave the diamond and go to an oval which does updates: front =mid+1,

and mid = (first + last) / 2. If this test was false, then the ASMD went onto the fourth, and final diamond and test. This diamond was the last case, so it would always be true. From the diamond, a line is drawn to an oval. Inside the oval there are two updates: last = mid – 1, and mid = (first + last) / 2. If either of the first tests were true, then the ASMD would be at s3. Coming out of s3 was a line to a diamond that check if start was 0. If start was 0, then you go back to s1. If not, then a line goes from the diamond back into s3. This can all be seen below in my ASMD chart in Figure 4.

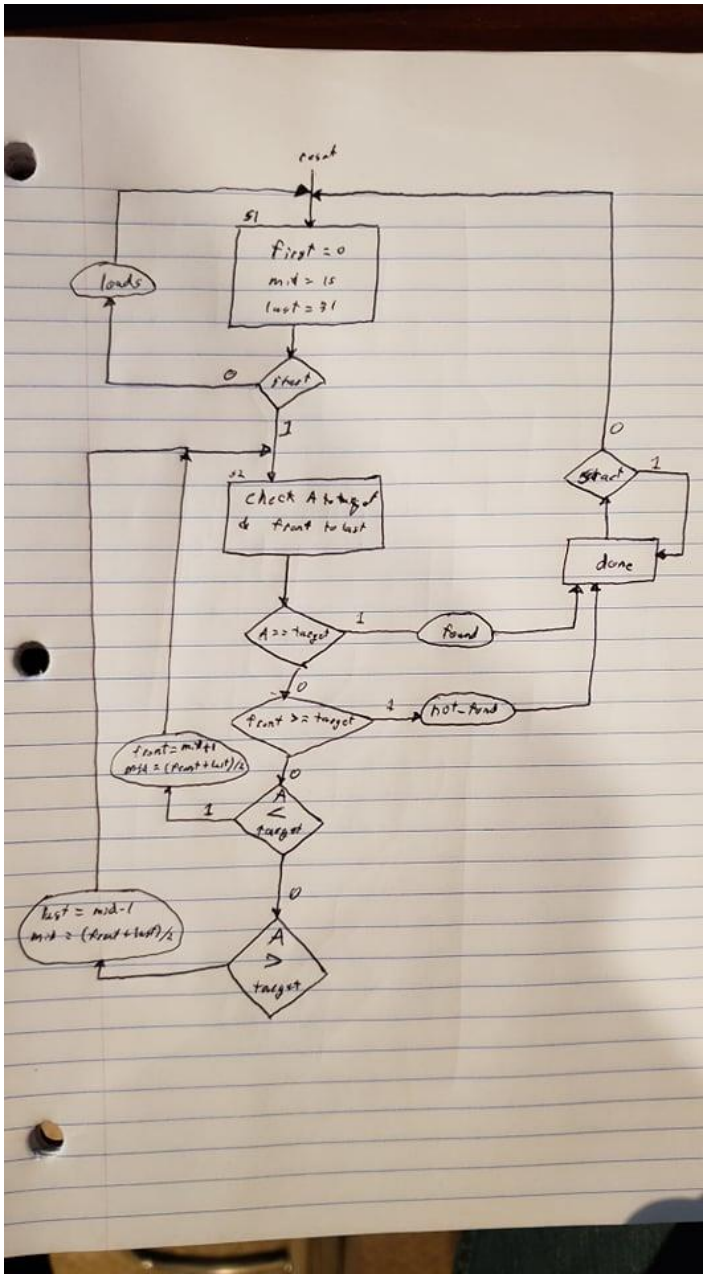

Fig. 4: ASMD Chart for Binary Search Algorithm

Once I had the ASMD chart done for task 2 I started working on the control logic by following along with my chart. I first created the states that I knew I would need: s1, s2, s3. I had a similar setup as task 1's s1, which tested if 1-bit input start was 1 or 0. If start was equal to 1 then the FSM would transition to s2. If it was 0, then it stayed in s1. The second state, s2, had an if statement that tested if 8-bit input A was

equal to 8-bit input target OR if 5-bit input front was greater than, or equal to 5-bit input last. This if statement took care of two of the diamonds in the ASMD chart, and I put them in the same if statement since they both would have the FSM transition to s3. If this test was false, then the FSM would stay in s2. This else statement handled the last two diamonds of the ASMD chart. For the third state, s3, I had similar logic to task 1's s3, where if start was 0 then the FSM would transition to s1, otherwise it would stay in s3. The FSM from binarysearch_Control can be seen below in Figure 5, and the complete code for this module can be seen in Appendix 2.A.

```
always_comb begin
   case(ps)
      s1: begin
         if(!start) begin
            ns = s1;
         end

         else begin
            ns = s2;
         end
      end

      s2: begin
         if((A == target) || (front >= last)) begin
            ns = s3;
         end

         else begin
            ns = s2;
         end
      end

      s3: begin
         if(!start) begin
            ns = s1;
         end

         else begin
            ns = s3;
         end
      end
   endcase
end
```

Fig. 5: Binary Search Control Logic FSM

Once I had the FSM setup, I assigned the output flags. The 1-bit logic loads was the exact same as loads_b from task 1, which was if the FSM was in state s1, and start was 0, then loads would be 1, otherwise it would be 0. The 1-bit logic found is set to 1 if the state is s3, and if 8-bit logic A is equal to 8-bit logic target, otherwise it is 0. The 1-bit logic not_found is set to 1 if the state is s3, and if 8-bit logic A is not equal to 8-bit logic target, otherwise it is 0. The 1-bit logic lt is set to 1 if the state is s2, and if 8-bit value A is less than 8-bit value target. The 1-bit logic gt is set to 1 if the state is s2, and if 8-bit value A is greater than 8-bit value target. The assignments for these values can be seen below in Figure 6.

```
//Assign load to 1 if in s1 and start is 0.
//Assign found to 1 if in s3 and A == target.
//Assign not_found to 1 if in s3 and A!= target.
//Assign lt to 1 if in s2 and A < target.
//Assign gt to 1 if in s2 and A > target.
assign loads     = ((ps == s1) && !start) ? 1'b1 : 1'b0;
assign found     = ((ps == s3) && (A == target)) ? 1'b1 : 1'b0;
assign not_found = ((ps == s3) && (A != target)) ? 1'b1 : 1'b0;
assign lt        = ((ps == s2) && (A < target)) ? 1'b1 : 1'b0;
assign gt        = ((ps == s2) && (A > target)) ? 1'b1 : 1'b0;
```

Fig. 6: Assigning Outputs in Binary Search Control

After I had everything done for the control logic, I moved onto the datapath for binary search. The datapath used 1-bit loads, lt, and gt to determine if the front and mid pointer needed to shift to the right, or if the mid and last pointer needed to shift to the left. Inside an always_ff there was an if

statement that checked if reset or loads was 1. If either of them was 1, then 5-bit front was set to 5'd0, 5-bit mid was set to 5'd16, and 5-bit last was set to 5d'31. After the if statement was two else if statements. The first else if checked if lt was 1. If it was, then last will update to mid – 1. The second else if statement checked if gt was 1. If it was, then front will update to mid + 1. After all cases, mid is updated to (front + last) / 2. This logic can be seen below in Figure 7, and the full code for binarysearch_Datapath can be found in Appendix 2.C.

```
//This always_ff sets 5-bit outputs front, mid,
//and last. If reset or loads is 1, then front
//is set to 0, mid is set to 16, and last is set
//to 31. If reset and loads are both 0, then the
//control will send flags for less than, or
//greater than depending on where the value lies
//compared to the data at memeray address mid.
//Shifting happens depending on which flag is
//raised, and then mid is always half of what
//the sum of front and last is.
always_ff @(posedge clk) begin
   if(reset || loads) begin
      front <= 5'b00000;
      mid   <= 5'b10000;
      last  <= 5'b11111;
   end

   else if(lt) begin
      last <= mid - 5'd1;
   end

   else if(gt) begin
      front <= mid + 5'd1;
   end

   mid  <= (front + last) / 2;
end
```

Fig. 6: Updating Output Logic for Binary Search Datapath

With the datapath done, I moved onto the hex_display module. I copied over the module from task 1, and made some slight changes. Since the address for which 5-bit mid was pointing to was being passed, I had 4-bit logic lower set to the lower 4-bits of mid's value, and then 1-bit logic upper set to the most significant bit of the value of mid. I had an always_comb with a case for lower, which assigned 7-bit led0 to a value that would display 1-15 in hex on HEX0, and a second case statement for upper that set 7-bit led1 to display 0-1 on HEX1. I put these case statement inside of an if statement that tested if found was equal to 1. If it was, then the display would update with the address. If found was 0, then the displays would be blank. This was a fairly simple module to create since I had most of the code already done.

With binarysearch_Control, binarysearch_Datapath, hex_display, and paramDFF done I moved onto DE1_SoC. I setup the DE1_SoC module using hierarchical calls to the module previously mentioned and I made a testbench. The testbench check to see if the system would iterate through the RAM array properly, if it would find a value in the RAM array, or if it wouldn't find the value. The reason as to why I did this task the way I did was because it was logical. The ASMD chart made the control logic and datapath fairly easy to write. I made the slight changes needed for hex_display so that it would accommodate a larger input value, and if the data was found in the array. If the data was not found, then I didn't want to display anything, and just have the HEX displays be blank. I had LEDR[8] = not_found, and LEDR[9] = found in the DE1_SoC. This was part of the task, so I made sure that an LED would light up if either found, or not_found was equal to 1.

Top-Level Block Diagrams

Task #1

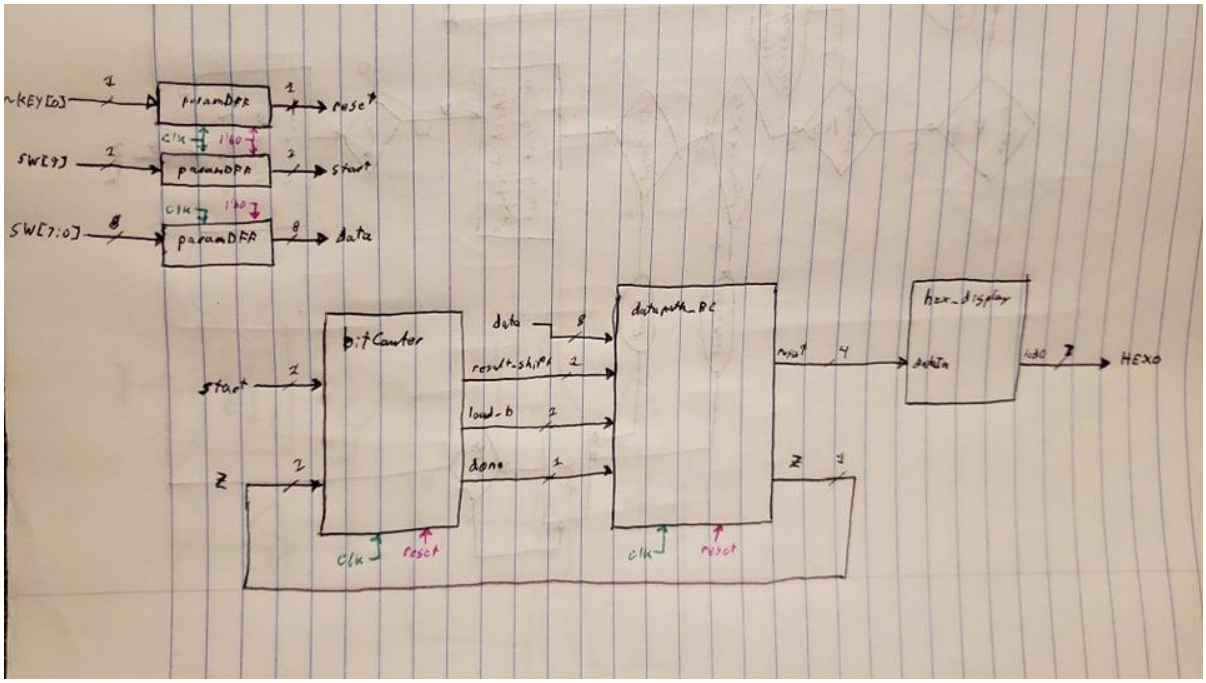

Fig. 7: DE1_SoC Block Diagram for Task 1

Task #2

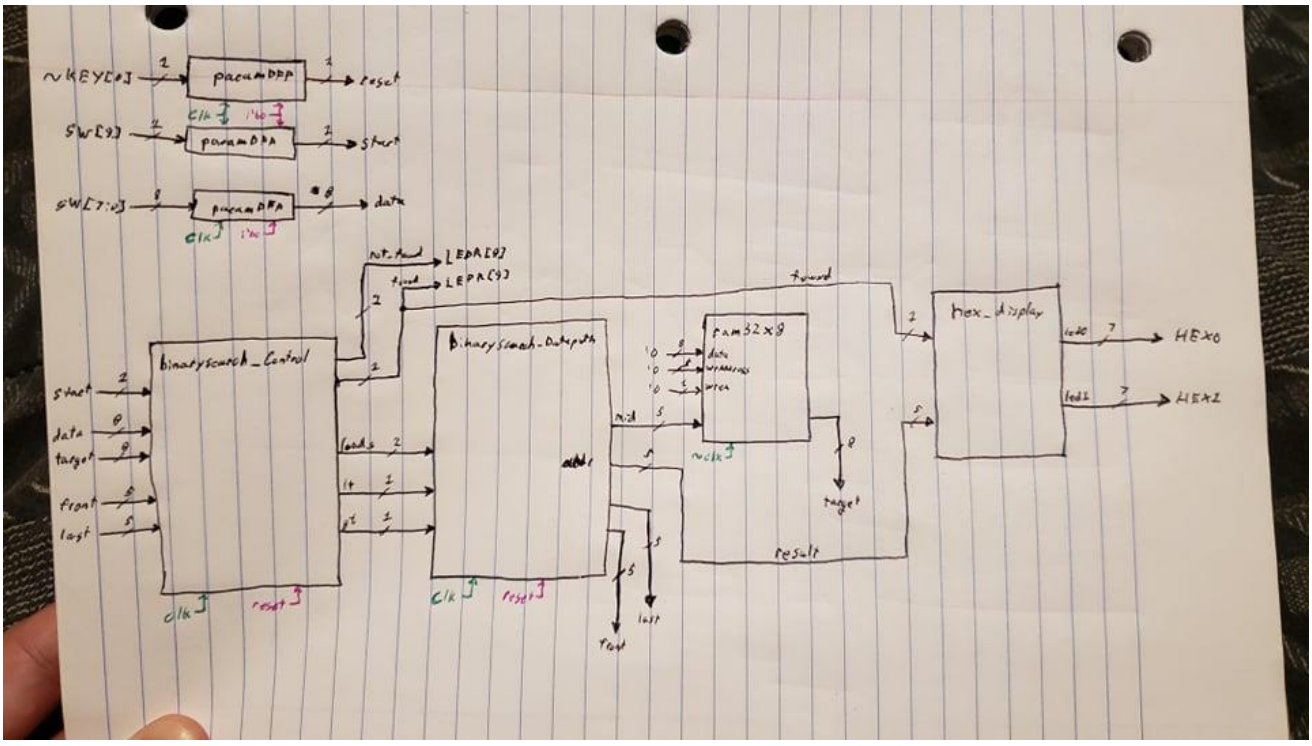

Fig. 8: DE1_SoC Block Diagram for Task 2

Results

Task #1

The first testbench I made for task 1 was for bitCounter. For this testbench I first started by setting 1-bit start and 1-bit z to 0. I then set reset high for a clock cycle, then low. After that, I set start, and z to 1 for two clock cycles. I then set z to 0 for 2 clock cycles, followed by setting start to 0 for two clock cycles. This was to see how the FSM would transition and if it would update correctly, along with other outputs in the system. After that, I set reset high, then low. I set start to 1 for 8 clock cycles to see if 1-bit output right_shift would go to 1. Then, I set start to 0, and set z to 1 for a clock cycle. This was to see if 1-bit output done would update by itself, without start being 1. Finally, I set z to 0 for 4 clock cycles to see if 1-bit done, and load_b would update correctly. The results from these tests were as expected. Load_b is 1 as long the FSM is in s1. If the FSM was in s1 and start was set to 1, then the FSM would transition to s2. If z was 1 while in s2, then the FSM will transition to s3, and done will go to 1. While in s2, right_shift does go to 1 as well. This can be seen below in Figure 9.

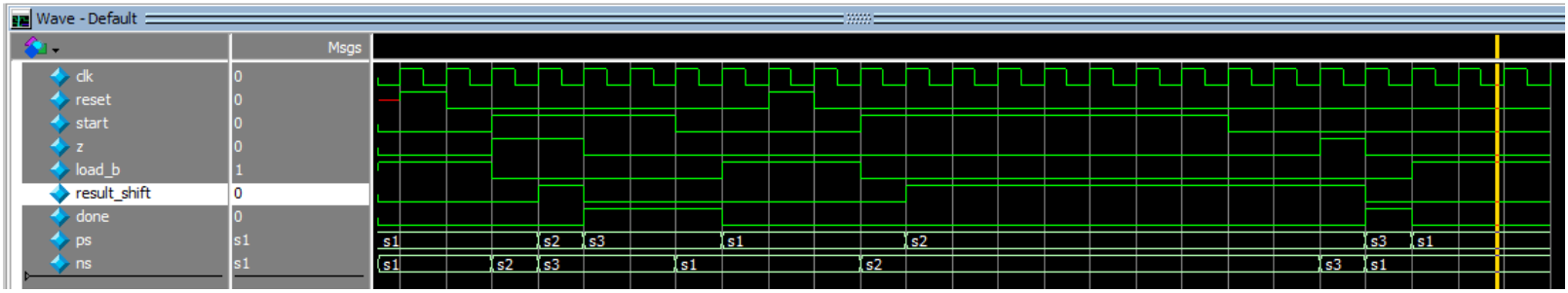

Fig. 9: bitCounter ModelSim Simulation

The second testbench I made for task 1 was for datapath_BC. For this testbench I set 8-bit input A to a non-zero value, 1-bit load_b, right_shift, and done all to 0. I then set reset high then low. I set right_shift to 1 for 8 clock cycles, then low. This was done to see if when new_a was all zeros, that z would go to 1. Next, I set done to 1 for one clock cycle to see if result would update correctly. I then changed the value of A to another non-zero value, set load_b to 1 for a clock cycle, then low to see if the system would behave the same is if reset was set high, then low. I then set right_shift high for 8 clock cycles, then low to see if values would update correctly. I set done to 1 for a clock cycle, then 0. The results for these tests were as expected. Upon reset or if load_b is 1, new_a is set to A, and count is set to 0. When right_shift is 1, count updates whenever the least significant bit of new_a is a 1, and then new_a is shifted properly. Once new_a is equal to 0, then z does go to 1, and if done is set to 1, then result updates to what count is. This can be seen below in Figure 10.

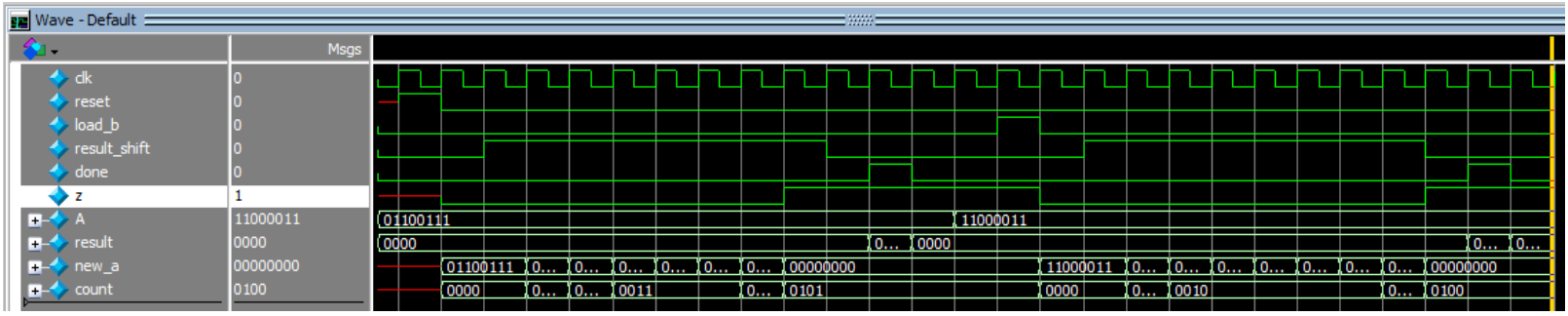

Fig. 10: datapath_BC ModelSim Simulation

Next, I made the testbench for hex_display. For this testbench I made a for loop. I had the for loop iterate from 0 to 15 and set 4-bit dataIn to the for-loop value i. This was to see if the 7-bit output led0 would update with the correct values depending on what value dataIn was. The results for this test was as expected. If dataIn changes, then led0 will update accordingly to display the correct numbers to HEX0. This can be seen below in Figure 11.

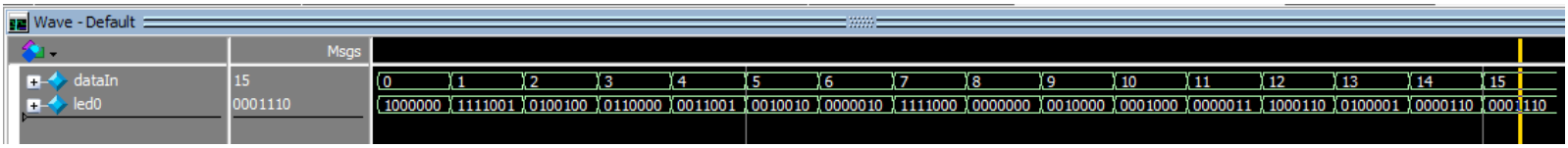

Fig. 11: hex_display ModelSim Simulation

Once I had the hex_display module done, I brought in my paramDFF module from a previous lab. For paramDFF testbench, I set press to four different values to see if out would update correctly. The results from this test were as expected, out updated to the correct value, and did it after two clock cycles as well. This can be seen in Figure 12 below.

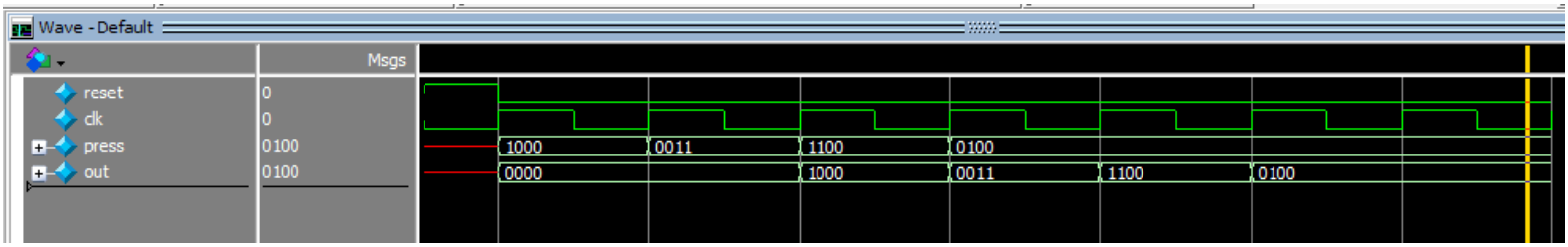

Fig. 12: paramDFF ModelSim Simulation

The last testbench that I made was for DE1_SoC. For this testbench, I first set data, SW7-0, to 8'b00110011. I then set start (SW9) to 0 for a clock cycle. After that, I set reset, KEY0, low then high to reset the system. I then set start high for 10 clock cycles. Next, I set start low for 5 clock cycles. From there I set start high again for 11 clock cycles. After those 11 clock cycles I reset the system again by setting KEY0 low then high, and held the high value for 15 clock cycles. I then set data to 8'b11111111 for 4 clock cycles without changing other values. Then I had start go low for 2 clock cycles, then high again. I changed data again, and this time to 8'b00011111. These values were held for 15 clock cycles. Lastly, I set start to 0 for 8 clock cycles, then high for 15 clock cycles. The results from these tests were as expected. The 1-bit logic load_b is 1 until 1-bit value start goes to 1, then load_b goes to 0. While start is 1, result_shift is 1, up until 1-bit z updates and then result_shift goes to 0. After z is 1, then done updates to 1 as well, and result updates to the number of 1's correctly. Once start is set to 0, then load_b goes back to 1. Reset works in the system as well, and resets the result back to '0. Also, if the system is finished while start is still high, it will retain the values until start is set low again. This can be seen below in Figure 13.

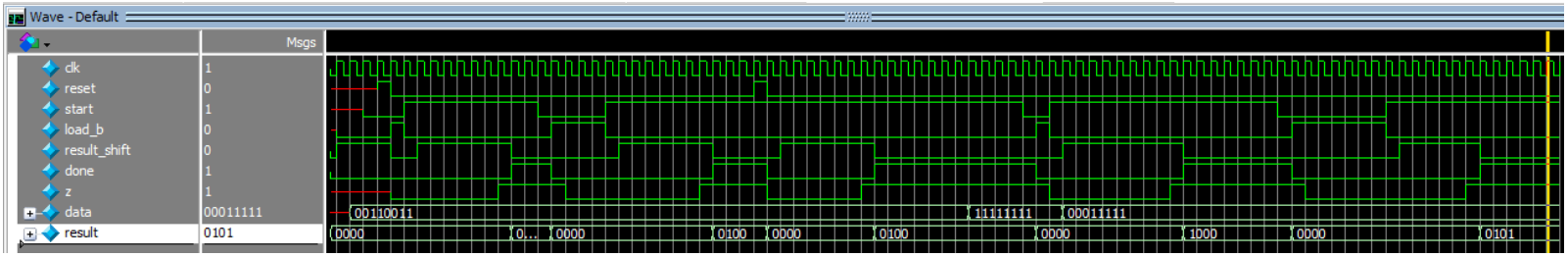

Fig. 13: DE1_SoC ModelSim Simulation

Task #2

The first testbench I made for task 2 was for binarysearch_Control. The testbench first sets start to 0 and then sets reset high then low. While start is 0, I first tested if the FSM would stay in s1 properly and if the outputs wouldn't update. I made 5-bit front less than 5-bit last for one clock cycle, front equal to last for one clock cycle, then I made front greater than last for one clock cycle. I then set 8-bit A to be less than 8-bit target for one clock cycle, then A to be greater than target, and finally A to be equal to target all for one clock cycle, respectfully. I then change front to be less than last, A to be less than target, and set start to 1. With start set to 1, I repeated my tests for A and target and did three tests of A less than target. After that, I had three tests in which A was greater than target. After that I had one test with A equal to target. I then set start to 0, and set A to be less than target. I set start to 1, and then set front to be equal to last. Then, I set start to 0, set front to be greater than target, and set start to 1 again. I finally set reset high, then low. The results from these tests were as expected. While in s1 and start set to 0, front, last, A, and target can be any value without changing any of the outputs, or causing a state transition in the FSM. Once start is set to 1 and A less than target, lt will be 1. If A is greater than target, then gt will be 1. If A and target are equal, then found will be 1, and the FSM will go to s3. Setting start to 0 at this point does the proper state change, and then setting it to 1 with front greater than, or equal to last, then the FSM transitions to s3 and not_found is set to 1. This can be seen below in Figure 14.

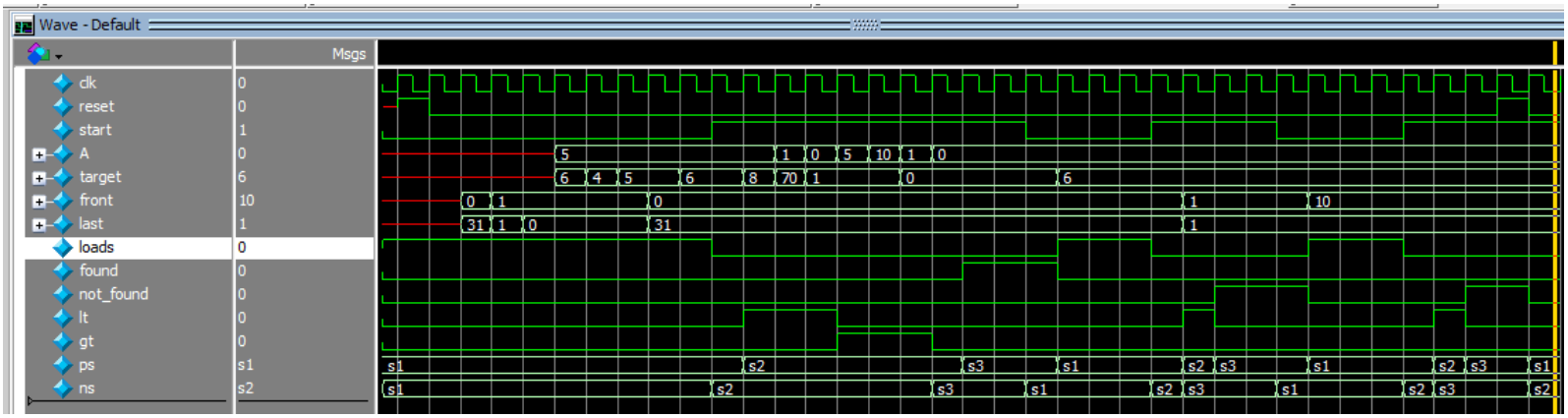

Fig. 14: binarysearch_Control ModelSim Simulation

Next, I moved onto the testbench for binarysearch_Datapath. I started this testbench off by first setting 1-bit loads, lt, and gt to 0. I then set reset high, then low. After that, I set lt to 1 for 8 clock cycles, then set it back to 0. I set loads to 1, then 0. After that, I set gt to 1 for 10 clock cycles, then to 0. This was followed by setting loads to 1, then 0. I then set reset high, then low. Next, I set lt high, then low twice in a row, followed by setting gt high, then low twice. The results from these tests were as expected. When lt is set to 1, 5-bit mid, and last are both updated correctly to the value of 5-bit front. When loads is set

to 1 then 0, front, mid, and last all get set to their starting values. When gt is 1, front and mid increment correctly up to the value of loads. Changing lt, and gt after reset/loads causes front, mid, and last update correctly. This can be seen below in Figure 15.

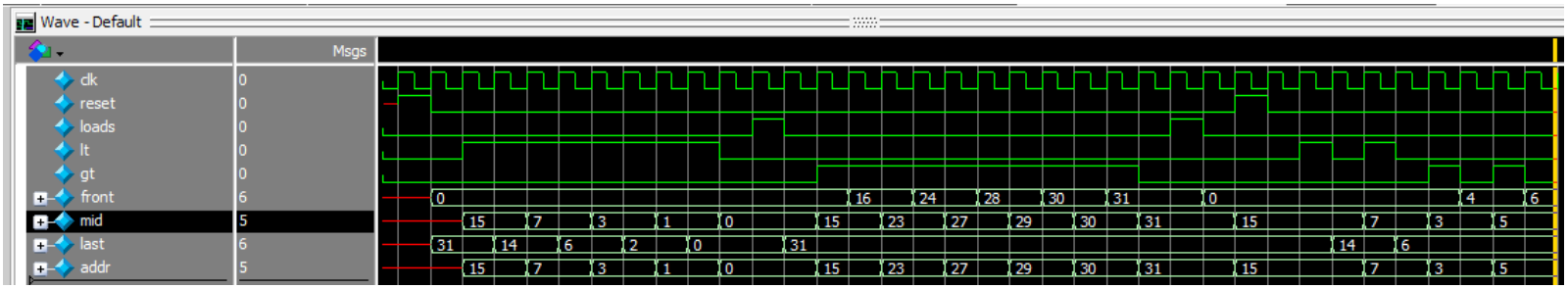

Fig. 15: binarysearch_Datapath ModelSim Simulation

Once I finished binarysearch_Datapath I moved onto hex_display. I first started these tests by setting 1-bit found to 0, and used a for loop that had int i go from 0-17 and set 5-bit dataIn to the value of i. I then repeated this test with a second for loop, but I set found to 1 prior to the second for loop. The results from this test were as expected. When found is set to 0, 7-bit led0 and led1 don't change at all. If found is set to 1, then the values for led0 and led1 update accordingly. This can be seen below in Figure 18.

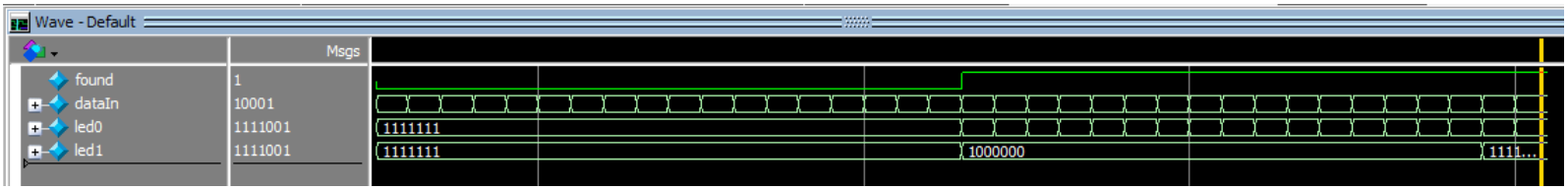

Fig. 18: hex_display ModelSim Simulation

Once I had the hex_display module done, I brought in my paramDFF module. For paramDFF testbench, I set press to four different values to see if out would update correctly. The results from this test were as expected, out updated to the correct value, and did it after two clock cycles as well. This can be seen in Figure 19 below.

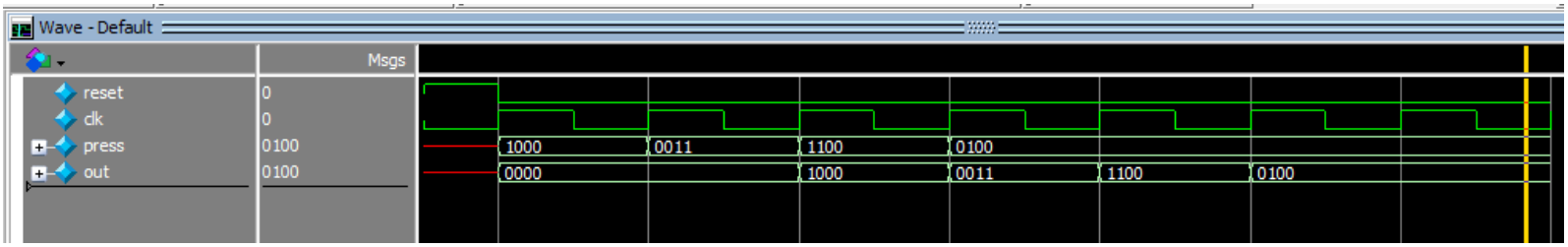

Fig. 19: paramDFF ModelSim Simulation

The last testbench I did for task 2 was for the DE1_SoC module. I started off by first setting 8-bit data to a value that was in the RAM module, setting 1-bit start to 0, and setting reset high for one clock cycle, then low for 14 clock cycles. I then set start to 1 for 18 clock cycles then low for 2. After this, I change the value of data to a value that was not in the RAM. I then set start to 1 for 20 clock cycles, then low for 2 clock cycles. I repeated this test with start a second time. Afterwards, I changed data to a value that was in the RAM again. I set reset high for 2 clock cycles, then low for 14, all while start was 0. I then set start to 1 for 20 clock cycles, then low for 2. The results from these tests were mostly as expected. The values for 5-bit front, mid, and last all stay at their starting value upon startup if start is 0. Once start is

set 1 then front, mid, and last all update accordingly until they are all equal, and the value is found. The value for data can change, and the system can start over if start is set low, then high. If a value is in the RAM, then found will output a 1. If a value is not in the RAM, then not_found will output a 1. This can be seen below in Figure 20. The one unexpected thing that happened can be seen in Figure 21. When a value is found, there is a clock cycle delay between when front, mid, and last are all equal, and when found updates. When the values are equal for one clock cycle not_found is 1, but then goes to 0. This isn't observed while on the board due to the high clock rate.

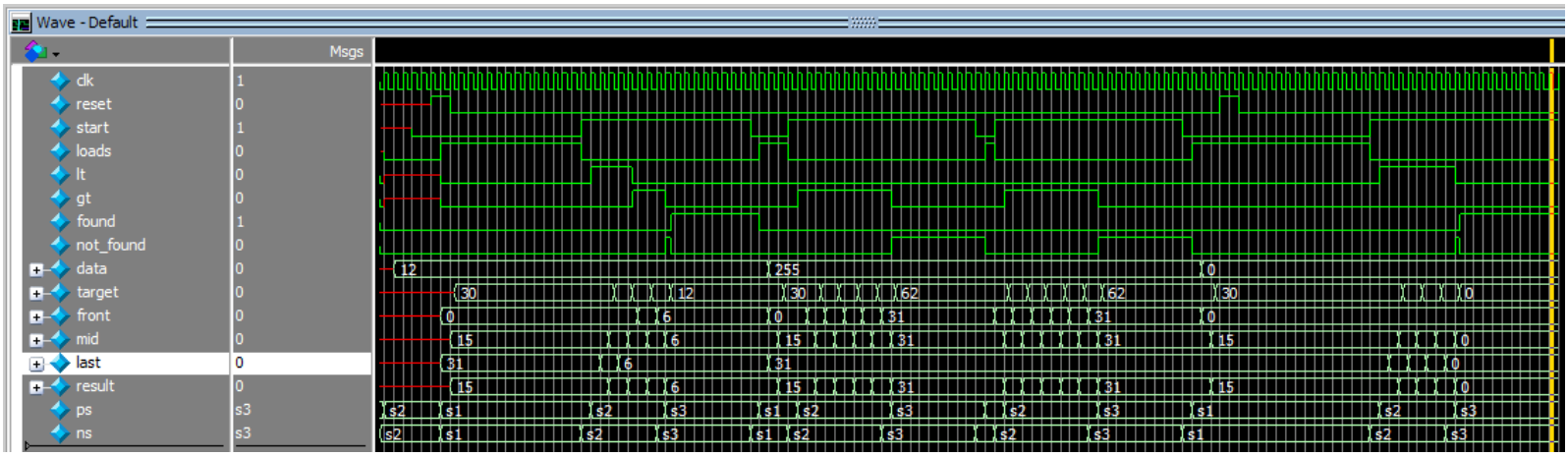

Fig. 20: DE1_SoC ModelSim Simulation

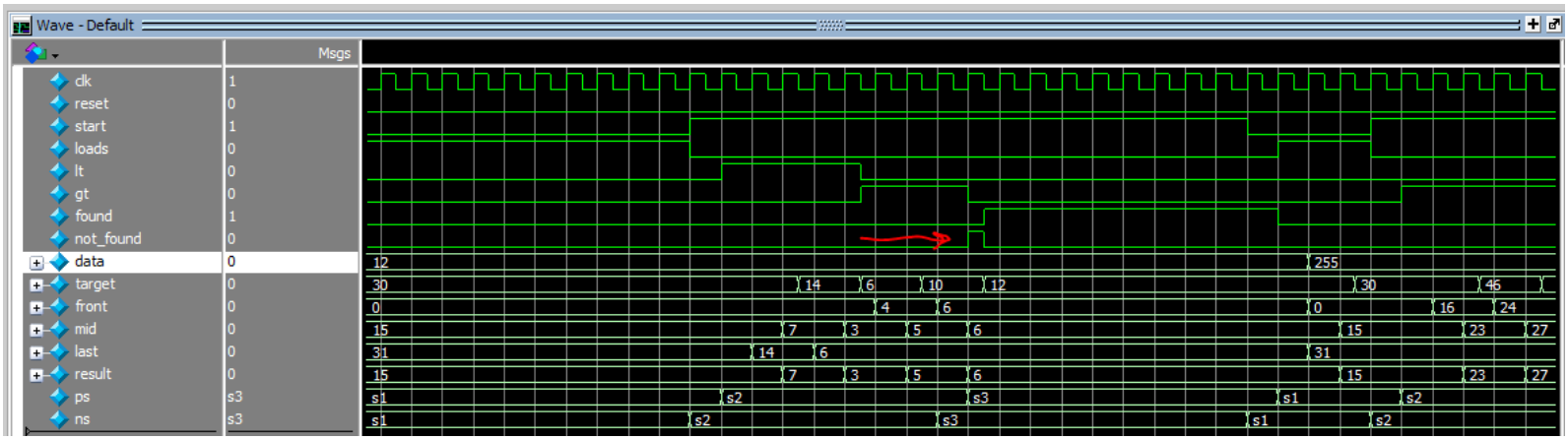

Fig. 21: DE1_SoC ModelSim Simulation Unexpected Result

Final Project

The goal of this lab was to learn how to create, and use ASMD charts to implement algorithms, such as a bit counting algorithm and binary search algorithm, on the DE1_SoC board using SystemVerilog. In this lab we learned how to read and create an ASMD chart and implement a bit counting algorithm from it with control logic and a datapath, as well as creating an ASMD chart for a binary search algorithm and implementing it with control logic and a datapath. The most challenging part of this lab was getting my inputs from the board to update properly after passing them through double DFFs. Upon reset, values were staying the same for one clock cycle, then reset, which messed with the whole system. This was

the first time that I have worked with an ASMD chart by having to write code from one that is given, or even create one and write the code from the one that I made.

In the end, I was able to produce the results that I wanted and I believe are sufficient and cover the requirements for lab 4.

Appendix

1.A) bitCounter.sv

```
1  //Garrett Tashiro
2  //November 15, 2021
3  //EE 371
4  //Lab 4, Task 1
5
6  //bitCounter has 1-bit clk, reset, start, and z as
7  //inputs and returns 1-bit load_b, result_shift, and done
8  //as outputs. This module is the control module for a bit
9  //counter. It has the FSM for the system, and will use inputs
10 //start and z to control state changing. Assignments for
11 //outputs are based upon the state the system is in.
12 module bitCounter(clk, reset, z, start, result_shift, load_b, done);
13       input logic              clk, reset, start, z;
14       output logic             load_b, result_shift, done;
15
16       //enumerated states for the FSM
17       enum{s1, s2, s3} ps, ns;
18
19       //This always_comb has the FSM for the bit counter. The FSM
20       //will stay in s1 as long as start is low. If start is high,
21       //the FSM will transition to s2. Once in s2, as long as 'z'
22       //is low, the FSM will stay in s2, which sets result_shift
23       //and shifts the bits in the datapath. If 'z' is 1, the FSM
24       //transitions to s3. The FSM stays in s3 as long as start is
25       //1, otherwise the FSM transitions to s1. While in s1 and
26       //!start, output load_b is 1. In s3, done is 1.
27       always_comb begin
28          case(ps)
29             s1: begin
30                if(!start) begin
31                   ns = s1;
32                end
33
34                else begin
35                   ns = s2;
36                end
37             end
38
39             s2: begin
40                if(z == 1) begin
41                   ns = s3;
42                end
43
44                else begin
45                   ns = s2;
46                end
47             end
48
49             s3: begin
50                if(start) begin
51                   ns = s3;
52                end
53
54                else begin
55                   ns = s1;
56                end
57             end
58          endcase
59       end
60
61       //This always_ff is to update ps. Upon reset
62       //ps is set to s1. If reset is 0, then ps is
63       //set to ns.
64       always_ff @(posedge clk) begin
65          if(reset) begin
66             ps <= s1;
67          end
68
69          else begin
70             ps <= ns;
71          end
72       end
73
74       ////Assignment of outputs////
75
76       //result_shift is 1 if in s2 state
77       assign result_shift = (ps == s2);
78
79       //done is 1 if in s3 state
80       assign done = (ps == s3);
81
82       //load_b is 1 if in s1 state and start is 0
83       assign load_b = ((ps == s1) && !start) ? 1'b1 : 1'b0;
84 endmodule
```

1.B) bitCounter.sv (testbench)

```
86  //bitCounter_testbench tests for expected, unexpected, and edgecase
87  //behavior. This testbench first sets 1-bit inputs start and z to 0.
88  //Reset is then set high for one clock cycle, then low. Next, start
89  //and z are both set to 1 for 2 clock cycles. Z is then set to 0 for
90  //two clock cycles. Then start is set to 0 for two clock cycles. This
91  //was done to see the behavior of the outputs, and if they would update
92  //properly. Next, reset is set high then low. Start is then set high
93  //for 8 clock cycles, and then low for 2. After that, z is set high for
94  //one clock cycle, then low for 4 clock cycles. This was checking if
95  //states and outputs would update properly as well.
96  module bitCounter_testbench();
97        logic       clk, reset, start, z;
98        logic       load_b, result_shift, done;
99
100       bitCounter dut(.clk, .reset, .z, .start, .result_shift, .load_b, .done);
101       parameter clk_PERIOD = 100;
102       initial begin
103             clk <= 0;
104             forever #(clk_PERIOD/2) clk <= ~clk;// Forever toggle the clk
105       end
106
107       initial begin
108          start <= 0; z <= 0;           repeat(1)   @(posedge    clk);
109          reset <= 1;                   repeat(1)   @(posedge    clk);
110          reset <= 0;                   repeat(1)   @(posedge    clk);
111          start <= 1; z <= 1;           repeat(2)   @(posedge    clk);
112          start <= 1; z <= 0;           repeat(2)   @(posedge    clk);
113          start <= 0; z <= 0;           repeat(2)   @(posedge    clk);
114          reset <= 1;                   repeat(1)   @(posedge    clk);
115          reset <= 0;                   repeat(1)   @(posedge    clk);
116          start <= 1;                   repeat(8)   @(posedge    clk);
117          start <= 0;                   repeat(2)   @(posedge    clk);
118          z <= 1;                       repeat(1)   @(posedge    clk);
119          z <= 0;                       repeat(4)   @(posedge    clk);
120
121          $stop; // End the simulation.
122       end
123 endmodule
```

1.C) datapath_BC.sv

```
1   //Garrett Tashiro
2   //November 15, 2021
3   //EE 371
4   //Lab 4, Task 1
5
6
7   //datapath_BC is a parameterized module that has 1-bit clk,
8   //reset, load_b, result_shift, done, and 8-bit A (parameterized
9   //variable set to 8) as inputs and returns 1-bit z, and 4-bit
10  //result as outputs. This module implements the datapath for a
11  //bit counter to count the number of 1's in a certain set of data
12  //being passed in.
13  module datapath_BC #(parameter width = 8)(clk, reset, A, load_b, result_shift, done, z, result);
14        input logic                 clk, reset;
15        input logic                 load_b, result_shift, done;
16        input logic [width - 1: 0] A;
17        output logic                z;
18        output logic [3:0]          result;
19
20        //logic to hold value of A that is orignially
21        //passed in so the data can be shifted to
22        //count the number of 1's in the data passed.
23        //4-bit logic count to hold the count of number
24        //of 1's that the data has.
25        logic [width - 1: 0] new_a;
26        logic [3:0] count;
27
28        //Assign output z to not or new_a
29        assign z = ~|new_a;
30
31        //Assign result to be 0 if done is 0,
32        //or for result to equal the count if
33        //done is 1 using a conditional operator.
34        assign result = (done) ? count : 4'd0;
```

```
35
36          //This always_ff block has the logic for the
37          //datapath. If reset or load_b are 1, then
38          //count is set to 0 and new_a is set to the
39          //input data. Else if right_shift is 1, if
40          //the data in the 0'th place of a is a 1 then
41          //increase count by 1, and always shift new_a
42          //to the right by 1.
43          always_ff @(posedge clk) begin
44             if(reset || load_b) begin
45                count <= 0;
46                new_a <= A;
47             end
48
49             else if(result_shift) begin
50                if(new_a[0] == 1) begin
51                   count <= count + 4'd1;
52                end
53                new_a <= new_a >> 1;
54             end
55          end
56       endmodule
```

1.D) datapath_BC.sv (testbench)

```
58    //datapath_BC_testbench tests for expected, unexpected, and edgecase
59    //behavior. This testbench starts by setting A to a non_zero value,
60    //then load_b, result_shift, and done to 0. Reset is set high then
61    //low. right_shift is set high for 8 clock cycles, and then low for one
62    //cycle. The 8-bit input value of A is changed to a different non-zero
63    //value, and load_b is set high for a clock cycle, then it is set low.
64    //result_shift is set high for 8 clock cycles, then low for 1, and then
65    //done is set high for a clock cycle then low again. This testbench was
66    //done to be sure value were updating correctly based upon the input
67    //values being passed to the system, and that the output values were getting
68    //updated correctly after the input values were cahnged.
69    module datapath_BC_testbench();
70          logic clk, reset, load_b, result_shift, done, z;
71          logic [7:0] A;
72          logic [3:0] result;
73
74          datapath_BC #(8) dut(.clk, .reset, .A, .load_b, .result_shift, .done, .z, .result);
75
76          parameter clk_PERIOD = 100;
77          initial begin
78                clk <= 0;
79                forever #(clk_PERIOD/2) clk <= ~clk;// Forever toggle the clk
80          end
81
82          initial begin
83             A <= 8'b01100111; load_b <= 0; result_shift <= 0; done <= 0; repeat(1)   @(posedge    clk);
84             reset <= 1;           repeat(1)   @(posedge    clk);
85             reset <= 0;           repeat(1)   @(posedge    clk);
86             result_shift <= 1;    repeat(8)   @(posedge    clk);
87             result_shift <= 0;    repeat(1)   @(posedge    clk);
88             done <= 1;            repeat(1)   @(posedge    clk);
89             done <= 0;            repeat(1)   @(posedge    clk);
90             A <= 8'b11000011;     repeat(1)   @(posedge    clk);
91             load_b <= 1;          repeat(1)   @(posedge    clk);
92             load_b <= 0;          repeat(1)   @(posedge    clk);
93             result_shift <= 1;    repeat(8)   @(posedge    clk);
94             result_shift <= 0;    repeat(1)   @(posedge    clk);
95             done <= 1;            repeat(1)   @(posedge    clk);
96             done <= 0;            repeat(1)   @(posedge    clk);
97
98             $stop; // End the simulation.
99          end
100   endmodule
```

1.E) hex_display.sv

```
1  //Garrett Tashiro
2  //November 15, 2021
3  //EE 371
4  //Lab 4, Task 1
5
6  //hex_display takes 4-bit dataIn as an input and returns
7  //7-bit led0 as an output. This module takes the data
8  //of number of 1's counted and then updates HEX0 display
9  //with the number of 1's.
10 module hex_display(dataIn, led0);
11       input logic [3:0]    dataIn;
12       output logic [6:0]   led0;
13
14       //This always_comb takes 4-bit dataIn as the case parameter
15       //and assigns led0 to values to display 0-F in hex to hex
16       //disaply 0 based upon the value of dataIn.
17       always_comb begin
18          case(dataIn)
19             4'd0: begin
20                led0 = 7'b1000000;    //0
21             end
22
23             4'd1: begin
24                led0 = 7'b1111001;    //1
25             end
26
27             4'd2: begin
28                led0 = 7'b0100100;    //2
29             end
30
31             4'd3: begin
32                led0 = 7'b0110000;    //3
33             end
34
35             4'd4: begin
36                led0 = 7'b0011001;    //4
37             end
38
39             4'd5: begin
40                led0 = 7'b0010010;    //5
41             end
42
43             4'd6: begin
44                led0 = 7'b0000010;    //6
45             end
46
47             4'd7: begin
48                led0 = 7'b1111000;    //7
49             end
50
51             4'd8: begin
52                led0 = 7'b0000000;    //8
53             end
54
55             4'd9: begin
56                led0 = 7'b0010000;    //9
57             end
58
59             4'd10: begin
60                led0 = 'b0001000;     //A
61             end
62
63             4'd11: begin
64                led0 = 7'b0000011;    //b
65             end
66
67             4'd12: begin
68                led0 = 7'b1000110;    //C
69             end
70
71             4'd13: begin
72                led0 = 7'b0100001;    //d
73             end
74
75             4'd14: begin
76                led0 = 7'b0000110;    //E
77             end
78
79             4'd15: begin
80                led0 = 7'b0001110;    //F
81             end
82
83             default: begin
84                led0 = 7'bx;
85             end
86          endcase
87       end
88 endmodule
```

1.F) hex_display.sv (testbench)

```
90    //hex_display_testbench tests for expected and unexpected
91    //behavior. This testsbench uses a for loop and assigns
92    //dataIn to decimal values 0-15 to see how the output led0
93    //behaves.
94    module hex_display_testbench();
95          logic [3:0] dataIn;
96          logic [6:0] led0;
97
98          hex_display dut(.dataIn, .led0);
99
100         integer i;
101
102         initial begin
103            for(i = 0; i < 16; i++) begin
104               dataIn = i;  #10;
105            end
106         end
107   endmodule
```

1.G) DE1_SoC.sv

```
1     //Garrett Tashiro
2     //November 15, 2021
3     //EE 371
4     //Lab 4, Task 1
5
6
7     //DE1_SoC is the top level module for Lab 4, Task 1. This module implements
8     //a bit counter to count the number of 1's in data that is passed to it with
9     //switches. The module uses hierarchical calls to paramDFF, bitCounter,
10    //datapath_BD, and hex_display. The DE1_SoC module takes inputs from SW[7:0],
11    //SW[9], and KEY[0]. The module will take in an 8-bit piece of data using
12    //SW[7:0], and if SW[9] is equal to one, then the system will count the number
13    //of bits int the 8-bits passed from the input that are 1. Once the system has
14    //finished counting all the bits, LEDR[9] will light up saying that the system
15    //is done and the number of ones that were counted will display on HEX0.
16    module DE1_SoC(HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, LEDR, SW, CLOCK_50);
17          input logic           CLOCK_50;
18          input logic [3:0]     KEY;
19          input logic [9:0]     SW;
20          output logic [6:0]    HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
21          output logic [9:0]    LEDR;
22
23          //Assign HEX1-HEX5 to be blank
24          assign HEX1 = 7'b1111111;
25          assign HEX2 = 7'b1111111;
26          assign HEX3 = 7'b1111111;
27          assign HEX4 = 7'b1111111;
28          assign HEX5 = 7'b1111111;
29
30          //Create 1-bit logic for reset, start, clk,
31          //load_b, result_shift, done, and z. Then
32          //create 8-bit logic for data and 4-bit logic
33          //for result.
34          logic reset, start, clk;
35          logic load_b, result_shift, done, z;
36          logic [7:0] data;
37          logic [3:0] result;
38
```

```
39          //Assign clk to CLOCK_50
40          //and LEDR[9] to the value of done.
41          assign clk = CLOCK_50;
42          assign LEDR[9] = done;
43
44          //paramDFF reset_dff is a parameterized module that has its
45          //parameter value set to 1. The module has 1-bit clk, 0 as reset,
46          //and KEY[0] as inputs and returns 1-bit reset as an output. This
47          //module is two DFFs in series and prevents metastability. The
48          //output value from this module is passed as an input to bitCounter,
49          //and datapath_BC.
50          paramDFF #(.itsy(1)) reset_dff(.clk(clk), .reset(1'b0), .press(~KEY[0]), .out(reset));
51
52          //paramDFF start_signal is a parameterized module that has its
53          //parameter value set to 1. The module has 1-bit clk, 0 as reset,
54          //and SW[9] as inputs and returns 1-bit start as an output. This
55          //module is two DFFs in series and prevents metastability. The
56          //output value from this module is passed as an input to bitCounter.
57          paramDFF #(.itsy(1)) start_signal(.clk(clk), .reset(1'b0), .press(SW[9]), .out(start));
58
59          //paramDFF input_data is a parameterized module that has its
60          //parameter value set to 8. The module has 1-bit clk, 0 as reset,
61          //and SW[7:0] as inputs and returns 8-bit data as an output. This
62          //module is two DFFs in series and prevents metastability. The
63          //output value from this module is passed as an input to bitCounter.
64          paramDFF #(.itsy(8)) input_data(.clk(clk), .reset(1'b0), .press(SW[7:0]), .out(data));
65
66          //bitCounter control has 1-bit clk, reset, start, and z as inputs
67          //and returns 1-bit result_shift, load_b, and done as outputs.
68          //This module is the control logic for the bit counter for this system,
69          //and has teh FSM inside of it. The outputs from bitCounter are passed to
70          //datapath_BC as inputs.
71          bitCounter control(.clk(clk), .reset(reset), .z, .start(start), .result_shift, .load_b, .done);
72
73          //datapath_BC datapath has 1-bit clk, reset, result_shift, load_b, done,
74          //and 8-bit data as inputs and returns 1-bit z, and 4-bit results as outputs.
75          //This module is teh datapath logic for the bit counter. This module does the
76          //shifting of the bits, checks if a bit is 1 or 0 in the data that was passed,
77          //and updates the count accordingly. Once the all the data that was passed is
78          //now 0's, z is set to 1, result is equal to the count.
79          datapath_BC #(.width(8)) datapath(.clk(clk), .reset(reset), .A(data), .load_b, .result_shift, .done, .z, .result(result));
80
81          //hex_display hexy has 4-bit result as and inputs and returns 7-bit value
82          //to HEX0. This module takes teh number of 1's counted in the data being
83          //input and then displays it to HEX0.
84          hex_display hexy(.dataIn(result), .led0(HEX0));
85
86      endmodule
```

1.H) DE1_SoC.sv (testbench)

```
88      //DE1_SoC_testbench tests for expected, unexpected, and edgecase behavior.
89      //This testbench first sets the input data to a value, and sets start (SW[9])
90      //to 0. The system is reset by setting KEY[0] low for a clock cycle, then high.
91      //start is then set high for 10 clock cycles to check if the system counts the
92      //number of 1's in the data being passed in. After that, start is set low for 5
93      //cycles to see if the values would update properly and the states would transition
94      //properly as well. Start is then set high for 11 clock cycles to see if values
95      //and states update properly again. With start still at 1, reset is set high then
96      //low. The value for data coming in is changed. Start is then set low for two clock
97      //cycles and then high again. This is done so it would cause state tranistions. While
98      //the number of 1's are beign counted the data input is changed to see if that affects
99      //the current procces. Start is et low for 8 clock cycles and then high for 15 to check
100     //for proper state transitions and values updating.
101     module DE1_SoC_testbench();
102           logic CLOCK_50;
103           logic [3:0] KEY;
104           logic [9:0] SW;
105           logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
106
107           DE1_SoC dut(.HEX0, .HEX1, .HEX2, .HEX3, .HEX4, .HEX5, .KEY, .SW, .CLOCK_50);
108
109           parameter CLOCK_PERIOD=100;
110           initial begin
111              CLOCK_50 <= 0;
112              forever #(CLOCK_PERIOD/2) CLOCK_50 <= ~CLOCK_50; // Forever toggle the clock
113           end
114
115           initial begin
116              SW[7:0] <= 8'b00110011;    repeat(1)   @(posedge    CLOCK_50);
117              SW[9] <= 0;                repeat(1)   @(posedge    CLOCK_50);
118              KEY[0] <= 0;               repeat(1)   @(posedge    CLOCK_50);
119              KEY[0] <= 1;               repeat(1)   @(posedge    CLOCK_50);
120              SW[9] <= 1;                repeat(10)   @(posedge    CLOCK_50);
121              SW[9] <= 0;                repeat(5)   @(posedge    CLOCK_50);
122              SW[9] <= 1;                repeat(11)   @(posedge    CLOCK_50);
123              KEY[0] <= 0;               repeat(1)   @(posedge    CLOCK_50);
124              KEY[0] <= 1;               repeat(15)   @(posedge    CLOCK_50);
125              SW[7:0] <= 8'b11111111;    repeat(4)   @(posedge    CLOCK_50);
126              SW[9] <= 0;                repeat(2)   @(posedge    CLOCK_50);
127              SW[9] <= 1;                repeat(1)   @(posedge    CLOCK_50);
128              SW[7:0] <= 8'b00011111;    repeat(1)   @(posedge    CLOCK_50);
129              SW[9] <= 1;                repeat(15)   @(posedge    CLOCK_50);
130              SW[9] <= 0;                repeat(8)   @(posedge    CLOCK_50);
131              SW[9] <= 1;                repeat(15)   @(posedge    CLOCK_50);
132
133              $stop; // End the simulation.
134           end
135     endmodule
```

2.A) binarysearch_Control.sv

```
1  //Garrett Tashiro
2  //November 17, 2021
3  //EE 371
4  //Lab 4, Task 2
5
6  //binarysearch_Control is a parameterized module that has
7  //1-bit clk, reset, start, 8-bit A, target, 5-bit front,
8  //and last as inputs and returns 1-bit loads, found,
9  //not_found, lt, gt as outputs. This module implements the
10 //control for binary search algorithm. The module has the
11 //FSM that sets flags depending on the input values, and
12 //determines if data that is wanted is in the RAM or not.
13 module binarysearch_Control #(parameter w = 8)(clk, reset, start, A, front, last,
14                                                target, loads, found, not_found, lt, gt);
15       input logic             clk, reset, start;
16       input logic [w - 1 : 0] A, target;
17       input logic [4:0]       front, last;
18       output logic            loads, found, not_found, lt, gt;
19
20       //Three states for the FSM
21       enum{s1, s2, s3} ps, ns;
22
23       //This always_comb implements an FSM for the binary
24       //search control. The FSM has three states: s1, s2,
25       //and s3. The FSM starts in s1 and stays in this state
26       //until the start signal is 1, then it transitions to
27       //s2. While in s2, there is an if statement that tests
28       //if the data at the address we are pulling from is the
29       //data we want OR if the front pointer is greater than
30       //or equal to the last pointer. If either of those are
31       //true, transition to s3. Otherwise stay in s2. The only
32       //way to transition out of s3 to s1 is if start is 0,
33       //otherwise you stay in s3.
34       always_comb begin
35          case(ps)
36             s1: begin
37                if(!start) begin
38                   ns = s1;
39                end
40
41                else begin
42                   ns = s2;
43                end
44             end
45
46             s2: begin
47                if((A == target) || (front >= last)) begin
48                   ns = s3;
49                end
50
51                else begin
52                   ns = s2;
53                end
54             end
55
56             s3: begin
57                if(!start) begin
58                   ns = s1;
59                end
60
61                else begin
62                   ns = s3;
63                end
64             end
65          endcase
66       end
67 endmodule
```

```
68          //This always_ff is to update ps.
69          //if reset is 1, then ps is set
70          //to s1. Else, ps is set to ns.
71          always_ff @(posedge clk) begin
72             if(reset) begin
73                ps <= s1;
74             end
75
76             else begin
77                ps <= ns;
78             end
79          end
80
81          //Assign load to 1 if in s1 and start is 0.
82          //Assign found to 1 if in s3 and A == target.
83          //Assign not_found to 1 if in s3 and A!= target.
84          //Assign lt to 1 if in s2 and A < target.
85          //Assign gt to 1 if in s2 and A > target.
86          assign loads     = ((ps == s1) && !start) ? 1'b1 : 1'b0;
87          assign found     = ((ps == s3) && (A == target)) ? 1'b1 : 1'b0;
88          assign not_found = ((ps == s3) && (A != target)) ? 1'b1 : 1'b0;
89          assign lt        = ((ps == s2) && (A < target)) ? 1'b1 : 1'b0;
90          assign gt        = ((ps == s2) && (A > target)) ? 1'b1 : 1'b0;
91    endmodule
```

2.B) binarysearch_Control.sv (testbench)

```
93
94    //binarysearch_Control_testbench tests for expected, unexpected,
95    //and edgecase behavior. The testbench first sets start to 0 and
96    //resets. With start at 0, the system is in s1, so tests for if
97    //A > target, A < target, A == target, front > last, front < last,
98    //and front == last are done to see if any of the output flags
99    //change even though in s1. Next, start is set to 1, and three
100   //tests are done in which A < target. After that, there are three
101   //tests for A > target, and then one test for if A == target. This
102   //was to see if lt, gt, and found were changing accordingly. After
103   //this, set A < target and start to 0 and then 1 to have the FSM
104   //transition. Two tests are done to see if found > last triggers
105   //not_found, and if first == last triggers nor_found. Reset at the
106   //end to see if ps is set to s1.
107   module binarysearch_Control_testbench();
108         logic       clk, reset, start;
109         logic [7:0] A, target;
110         logic [4:0] front, last;
111         logic       loads, found, not_found, lt, gt;
112
113         binarysearch_Control #(.w(8)) dut(.clk, .reset, .start, .A, .front, .last, .target,
114                                           .loads, .found, .not_found, .lt, .gt);
115
116         parameter clk_PERIOD = 100;
117         initial begin
118               clk <= 0;
119               forever #(clk_PERIOD/2) clk <= ~clk;// Forever toggle the clk
120         end
121
122
```

```
123         initial begin
124            start <= 0;                  repeat(1)   @(posedge    clk);
125            reset <= 1;                  repeat(1)   @(posedge    clk);
126            reset <= 0;                  repeat(1)   @(posedge    clk);
127
128            //Start isn't at 1. Still in s1. Check if outputs update correctly.
129            front <= 5'd0; last <= 5'd31; repeat(1)   @(posedge    clk);
130            front <= 5'd1; last <= 5'd1;  repeat(1)   @(posedge    clk);
131            front <= 5'd1; last <= 5'd0;  repeat(1)   @(posedge    clk);
132            A <= 8'd5; target <= 8'd6;    repeat(1)   @(posedge    clk);
133            A <= 8'd5; target <= 8'd4;    repeat(1)   @(posedge    clk);
134            A <= 8'd5; target <= 8'd5;    repeat(1)   @(posedge    clk);
135
136            //Setting inputs to values that won't trigger flags. Start to 1.
137            front <= 5'd0; last <= 5'd31; repeat(1)   @(posedge    clk);
138            A <= 8'd5; target <= 8'd6;    repeat(1)   @(posedge    clk);
139            start <= 1;                  repeat(1)   @(posedge    clk);
140
141            //A < target while in s2
142            A <= 8'd5; target <= 8'd8;    repeat(1)   @(posedge    clk);
143            A <= 8'd1; target <= 8'd70;   repeat(1)   @(posedge    clk);
144            A <= 8'd0; target <= 8'd1;    repeat(1)   @(posedge    clk);
145
146            //A > target while in s2
147            A <= 8'd5; target <= 8'd1;    repeat(1)   @(posedge    clk);
148            A <= 8'd10; target <= 8'd1;   repeat(1)   @(posedge    clk);
149            A <= 8'd1; target <= 8'd0;    repeat(1)   @(posedge    clk);
150
151            //A == target. Wait 3 cycles to see behavior.
152            A <= 8'd0; target <= 8'd0;    repeat(3)   @(posedge    clk);
153
154            //start to 0, back to s1
155            start <= 0;                  repeat(1)   @(posedge    clk);
156            A <= 8'd0; target <= 8'd6;    repeat(3)   @(posedge    clk);
```

```
157
158         //Start to 1, and check for state change with front >= last
159         start <= 1;                 repeat(1)   @(posedge    clk);
160         front <= 5'd1; last <= 5'd1; repeat(3)   @(posedge    clk);
161         start <= 0;                 repeat(1)   @(posedge    clk);
162         front <= 5'd10; last <= 5'd1; repeat(3)   @(posedge    clk);
163         start <= 1;                 repeat(3)   @(posedge    clk);
164
165         //Test reset from s3
166         reset <= 1;                 repeat(1)   @(posedge    clk);
167         reset <= 0;                 repeat(1)   @(posedge    clk);
168
169         $stop; // End the simulation.
170     end
171 endmodule
```

2.C) binarysearch_Datapath.sv

```
 1  //Garrett Tashiro
 2  //November 17, 2021
 3  //EE 371
 4  //Lab 4, Task 2
 5
 6  //binarysearch_Datapath has 1-bit clk, reset, loads, lt,
 7  //and gt as inputs and returns 5-bit front, mid, last, and
 8  //addr as outputs. This module implements the datapath for
 9  //the binary search algo. Depending on what the flag inputs
10  //are will determine if front, last, and mid will change to
11  //shift the search around the array.
12  module binarysearch_Datapath(clk, reset, loads, lt, gt, front, mid, last, addr);
13      input logic             clk, reset;
14      input logic             loads, lt, gt;
15      output logic [4:0]      front, mid, last, addr;
16
17      //Assign addr to mid to send to hex_display.
18      assign addr = mid;
19
20      //This always_ff sets 5-bit outputs front, mid,
21      //and last. If reset or loads is 1, then front
22      //is set to 0, mid is set to 16, and last is set
23      //to 31. If reset and loads are both 0, then the
24      //control will send flags for less than, or
25      //greater than depending on where the value lies
26      //compared to the data at memeray address mid.
27      //Shifting happens depending on which flag is
28      //raised, and then mid is always half of what
29      //the sum of front and last is.
30      always_ff @(posedge clk) begin
31          if(reset || loads) begin
32              front <= 5'b00000;
33              mid   <= 5'b10000;
34              last  <= 5'b11111;
35          end
36
37          else if(lt) begin
38              last <= mid - 5'd1;
39          end
40
41          else if(gt) begin
42              front <= mid + 5'd1;
43          end
44
45          mid  <= (front + last) / 2;
46      end
47  endmodule
```

2.D) binarysearch_Datapath.sv (testbench)

```
49  //binarysearch_Datapath_testbench tests for expected, unexpected,
50  //and edgecase behavior. The testbench first sets loadsm lt, and gt
51  //to 0 and resets. After the reset, lt is set to 1 for 8 cycles to
52  //see if mid and last will equal front. After that, lt is set to 0
53  //and loads is set high then low to reset the system. gt is then
54  //set high for 10 clock cycles to see if front and mid will increment
55  //up and equal last. After that reset is set high then low. lt is
56  //set low then high twice, and gt is set low than twice to see if
57  //the system can accpect both lt, and gt correctly.
58  module binarysearch_Datapath_testbench();
59        logic clk, reset, loads, lt, gt;
60        logic [4:0] front, mid, last, addr;
61
62        binarysearch_Datapath dut(.clk, .reset, .loads, .lt, .gt, .front, .mid, .last, .addr);
63
64        parameter clk_PERIOD = 100;
65        initial begin
66              clk <= 0;
67              forever #(clk_PERIOD/2) clk <= ~clk;// Forever toggle the clk
68        end
69
70        initial begin
71           loads <= 0; lt <= 0; gt <= 0;    repeat(1)   @(posedge    clk);
72           reset <= 1;                      repeat(1)   @(posedge    clk);
73           reset <= 0;                      repeat(1)   @(posedge    clk);
74           lt <= 1;                         repeat(8)   @(posedge    clk);
75           lt <= 0;                         repeat(1)   @(posedge    clk);
76           loads <= 1;                      repeat(1)   @(posedge    clk);
77           loads <= 0;                      repeat(1)   @(posedge    clk);
78
79           gt <= 1;                         repeat(10)  @(posedge    clk);
80           gt <= 0;                         repeat(1)   @(posedge    clk);
81           loads <= 1;                      repeat(1)   @(posedge    clk);
82           loads <= 0;                      repeat(1)   @(posedge    clk);
83
84            reset <= 1;                       repeat(1)   @(posedge    clk);
85            reset <= 0;                       repeat(1)   @(posedge    clk);
86
87            lt <= 1;                          repeat(1)   @(posedge    clk);
88            lt <= 0;                          repeat(1)   @(posedge    clk);
89            lt <= 1;                          repeat(1)   @(posedge    clk);
90            lt <= 0;                          repeat(1)   @(posedge    clk);
91            gt <= 1;                          repeat(1)   @(posedge    clk);
92            gt <= 0;                          repeat(1)   @(posedge    clk);
93            gt <= 1;                          repeat(1)   @(posedge    clk);
94            gt <= 0;                          repeat(1)   @(posedge    clk);
95
96            $stop; // End the simulation.
97        end
98  endmodule
```

2.E) hex_display.sv

```
1  //Garrett Tashiro
2  //November 15, 2021
3  //EE 371
4  //Lab 4, Task 2
5
6  //hex_display takes 4-bit dataIn and 1-bit found as inputs a
7  //nd returns 7-bit led0, led1 as outputs. This module takes
8  //the data of the address that mid is pointing to, and if
9  //found is 1, then the address will be displayed on HEX1 and
10 //HEX0, otherwise HEX0 and HEX1 are blank.
11 module hex_display(dataIn, found, led0, led1);
12       input logic          found;
13       input logic [4:0]    dataIn;
14       output logic [6:0]   led0, led1;
15
16       //logic for upper bit of dataIn as well
17       //logic lower for lower 4 bits of dataIn
18       logic upper;
19       logic [3:0] lower;
20
21       //Assign lower to the lower 4 bits of dataIn
22       //Assing the most significant bit of dataIn to upper.
23       assign lower = dataIn[3:0];
24       assign upper = dataIn[4];
25
26       //This always_comb has an if statement to check if found
27       //is 1. If it is, there are two case statements: lower,
28       //and upper. Case statement for lower sets led0 to the
29       //respective value of the lower bits of dataIn to display
30       //0-F. Case statement for upper sets led1 to the respective
31       //value of the upper bit of dataIn to 0 or 1. Else statement
32       //for if the data is not found, and in this case both led0
33       //and led1 are going to be blank on the HEX displays.
34       always_comb begin
35          if(found) begin
36             case(lower)
37                4'd0: begin
38                   led0 = 7'b1000000;   //0
39                end
40
41                4'd1: begin
42                   led0 = 7'b1111001;   //1
43                end
44
45                4'd2: begin
46                   led0 = 7'b0100100;   //2
47                end
```

```
48
49                4'd3: begin
50                   led0 = 7'b0110000;   //3
51                end
52
53                4'd4: begin
54                   led0 = 7'b0011001;   //4
55                end
56
57                4'd5: begin
58                   led0 = 7'b0010010;   //5
59                end
60
61                4'd6: begin
62                   led0 = 7'b0000010;   //6
63                end
64
65                4'd7: begin
66                   led0 = 7'b1111000;   //7
67                end
68
69                4'd8: begin
70                   led0 = 7'b0000000;   //8
71                end
72
73                4'd9: begin
74                   led0 = 7'b0010000;   //9
75                end
76
77                4'd10: begin
78                   led0 = 'b0001000;    //A
79                end
80
81                4'd11: begin
82                   led0 = 7'b0000011;   //b
83                end
```

```
84
85              4'd12: begin
86                  led0 = 7'b1000110;    //C
87              end
88
89              4'd13: begin
90                  led0 = 7'b0100001;    //d
91              end
92
93              4'd14: begin
94                  led0 = 7'b0000110;    //E
95              end
96
97              4'd15: begin
98                  led0 = 7'b0001110;    //F
99              end
100
101             default: begin
102                 led0 = 7'bx;
103             end
104         endcase
105
106         case(upper)
107             1'b0: begin
108                 led1 = 7'b1000000;    //0
109             end
110
111             1'b1: begin
112                 led1 = 7'b1111001;    //1
113             end
114
115             default: begin
116                 led1 = 7'bx;
117             end
118         endcase
119     end
120
121     else begin
122         led0 = 7'b1111111;
123         led1 = 7'b1111111;
124     end
125 end
126 endmodule
```

2.F) hex_display.sv (testbench)

```
128 //hex_display_testbench tests for expected and unexpected
129 //behavior. This testbench first sets found to 0 then uses
130 //a for loop to change dataIn by setting it to 'i'. The loop
131 //goes from 0-17. After that, found is set to 1, and there
132 //is a second for loop that goes from 0-17. This is to check
133 //if the HEX displays will be updating according to the value
134 //of input found.
135 module hex_display_testbench();
136       logic found;
137       logic [4:0] dataIn;
138       logic [6:0] led0, led1;
139
140       hex_display dut(.dataIn, .found, .led0, .led1);
141
142       integer i;
143
144       initial begin
145           found <= 0;
146           for(i = 0; i < 18; i++) begin
147               dataIn = i;  #10;
148           end
149
150           found <= 1;
151           for(i = 0; i < 18; i++) begin
152               dataIn = i;  #10;
153           end
154       end
155 endmodule
```

2.G) ram32x8.v

```
// synopsys translate_off
`timescale 1 ps / 1 ps
// synopsys translate_on
module ram32x8 (
    clock,
    data,
    rdaddress,
    wraddress,
    wren,
    q);

    input   clock;
    input [7:0]  data;
    input [4:0]  rdaddress;
    input [4:0]  wraddress;
    input   wren;
    output   [7:0]  q;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_off
`endif
    tri1    clock;
    tri0    wren;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_on
`endif

    wire [7:0] sub_wire0;
    wire [7:0] q = sub_wire0[7:0];

    altsyncram  altsyncram_component (
              .address_a (wraddress),
              .address_b (rdaddress),
              .clock0 (clock),
              .data_a (data),
              .wren_a (wren),
              .q_b (sub_wire0),
              .aclr0 (1'b0),
              .aclr1 (1'b0),
              .addressstall_a (1'b0),
              .addressstall_b (1'b0),
              .byteena_a (1'b1),
              .byteena_b (1'b1),
              .clock1 (1'b1),
              .clocken0 (1'b1),
              .clocken1 (1'b1),
              .clocken2 (1'b1),
              .clocken3 (1'b1),
              .data_b ({8{1'b1}}),
              .eccstatus (),
              .q_a (),
              .rden_a (1'b1),
              .rden_b (1'b1),
              .wren_b (1'b0));
```

```
    defparam
        altsyncram_component.address_aclr_b = "NONE",
        altsyncram_component.address_reg_b = "CLOCK0",
        altsyncram_component.clock_enable_input_a = "BYPASS",
        altsyncram_component.clock_enable_input_b = "BYPASS",
        altsyncram_component.clock_enable_output_b = "BYPASS",
        altsyncram_component.init_file = "my_array.mif",
        altsyncram_component.intended_device_family = "Cyclone V",
        altsyncram_component.lpm_type = "altsyncram",
        altsyncram_component.numwords_a = 32,
        altsyncram_component.numwords_b = 32,
        altsyncram_component.operation_mode = "DUAL_PORT",
        altsyncram_component.outdata_aclr_b = "NONE",
        altsyncram_component.outdata_reg_b = "UNREGISTERED",
        altsyncram_component.power_up_uninitialized = "FALSE",
        altsyncram_component.ram_block_type = "M10K",
        altsyncram_component.read_during_write_mode_mixed_ports = "DONT_CARE",
        altsyncram_component.widthad_a = 5,
        altsyncram_component.widthad_b = 5,
        altsyncram_component.width_a = 8,
        altsyncram_component.width_b = 8,
        altsyncram_component.width_byteena_a = 1;


endmodule
```

2.H) DE1_SoC.sv

```
1  //Garrett Tashiro
2  //November 15, 2021
3  //EE 371
4  //Lab 4, Task 2
5
6  //DE1_SoC is the top level module for Lab 4, Task 2. This module
7  //implements a binary search algorithm on a ram module that is
8  //32x8 to search for data inside the module. This module uses
9  //hierarchical calls to paramDFF, binarysearch_Control, ram32x8,
10 //binarysearch_Datapath, and hex_display. The DE1 takes data input
11 //from SW[7:0] and checks the ram for that value. KEY[0] controls
12 //reset, and SW[9] is the switch for the start signal. If the data
13 //is found inside the ram, then LEDR[9] will light up and the address
14 //for the data will display on HEX0 and HEX1. If the data is not
15 //found then LEDR[8] will light up, and HEX0 and HEX1 will be blank.
16 module DE1_SoC(HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, LEDR, SW, CLOCK_50);
17       input logic          CLOCK_50;
18       input logic [3:0]    KEY;
19       input logic [9:0]    SW;
20       output logic [6:0]   HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
21       output logic [9:0]   LEDR;
22
23       //Set HEX2-HEX5 to be blank
24       assign HEX2 = 7'b1111111;
25       assign HEX3 = 7'b1111111;
26       assign HEX4 = 7'b1111111;
27       assign HEX5 = 7'b1111111;
28
29       //logic for reset, start, clk,
30       //loads, lt, gt, found, not_found,
31       //data, target, front, mid, last,
32       //and result.
33       logic reset, start, clk;
34       logic loads, lt, gt;
35       logic found, not_found;
36       logic [7:0] data, target;
37       logic [4:0] front, mid, last, result;
38
```

```
38
39       //Assign clk to CLOCK_50,
40       //LEDR[9] to the value of found
41       //LEDR[8] to the value of not_found
42       assign clk = CLOCK_50;
43       assign LEDR[9] = found;
44       assign LEDR[8] = not_found;
45
46       //paramDFF reset_dff is a parameterized module that has its
47       //parameter value set to 1. The module has 1-bit clk, 0 as reset,
48       //and KEY[0] as inputs and returns 1-bit reset as an output. This
49       //module is two DFFs in series and prevents metastability. The
50       //output value from this module is passed as an input to bitCounter,
51       //and datapath_BC.
52       paramDFF #(.itsy(1)) reset_dff(.clk(clk), .reset(1'b0), .press(~KEY[0]), .out(reset));
53
54       //paramDFF start_signal is a parameterized module that has its
55       //parameter value set to 1. The module has 1-bit clk, 0 as reset,
56       //and SW[9] as inputs and returns 1-bit start as an output. This
57       //module is two DFFs in series and prevents metastability. The
58       //output value from this module is passed as an input to
59       //binarysearch_Control.
60       paramDFF #(.itsy(1)) start_signal(.clk(clk), .reset(1'b0), .press(SW[9]), .out(start));
61
62       //paramDFF input_data is a parameterized module that has its
63       //parameter value set to 8. The module has 1-bit clk, 0 as reset,
64       //and SW[7:0] as inputs and returns 8-bit data as an output. This
65       //module is two DFFs in series and prevents metastability. The
66       //output value from this module is passed as an input to
67       //binarysearch_Control.
68       paramDFF #(.itsy(8)) input_data(.clk(clk), .reset(1'b0), .press(SW[7:0]), .out(data));
69
```

```
70          //binarysearch_Control control has 1-bit clk, reset, start, 8-bit data,
71          //5-bit front, and last as inputs and returns 1-bit loads, found,
72          //not_found, lt, and gt as outputs. The 1-bit ouputs loads, lt, and gt
73          //are passed to binarysearch_Datapath. The 1-bit output found is passed
74          //to hex_display as an input, and the output not_found is the value LEDR[8]
75          //is assigned to. This module is the control for the system and has the FSM.
76          binarysearch_Control #(.w(8)) control(.clk(clk), .reset(reset), .start(start),
77                                                .A(data), .front(front), .last(last),
78                                                .target(target), .loads(loads), .found(found),
79                                                .not_found(not_found), .lt(lt), .gt(gt));
80
81          //ram32x8 has 1-bit ~clk, 8'd0 for data, 5-bit mid, 5'd0 for wraddress, and
82          //1'b0 for wren as inputs returns 8-bit target as an output. This module implements
83          //a 32x8 RAM. Three inputs are zeroed out since we only want to read from mid
84          //addresses data and return it to target. target is passed to binarysearch_Control
85          //as an input.
86          ram32x8 sortedRAM(.clock(~clk), .data(8'd0), .rdaddress(mid),
87                            .wraddress(5'd0), .wren(1'b0), .q(target));
88
89          //binarysearch_Datapath has 1-bit clk, reset, loads, lt, and gt as inputs
90          //and returns 5-bit front, mid, last, and addr as outputs. This module is
91          //the datapath for the binary search algorithm. 5-bit outputs front, and last
92          //are passed to binarysearch_Control as inputs. 5-bit output mid is passed to
93          //both binarysearch_Control and ram32x8 as an input. 5-bit output addr is passed
94          //to hex_display as an input.
95          binarysearch_Datapath datapath(.clk(clk), .reset(reset), .loads(loads),
96                                         .lt(lt), .gt(gt), .front(front),
97                                         .mid(mid), .last(last), .addr(result));
98
99          //hex_display has 5-bit result, and 1-bit found as inputs and return
100         //7-bit outputs to HEX0 and HEX1. This module takes the addr that mid is set
101         //to, and will display the address onto HEX1 and HEX0 only if found is equal
102         //to 1. Otherwise both displays will be blank.
103         hex_display hexy(.dataIn(result), .found(found), .led0(HEX0), .led1(HEX1));
104
105 endmodule
```

2.I) DE1_SoC.sv (testbench)

```
106
107 `timescale 1 ps / 1 ps
108
109 //DE1_SoC_testbench tests for expected, unexpected, and edgecase behavior.
110 //The testbench first starts by setting SW7-0 to a value in the RAM. SW9
111 //is set to 0. KEY0 is set low then high to reset the system and this high
112 //value is then held for 14 clock cycles to see if state transitions will
113 //happen prior to having SW9 be high. SW9 is then set high for 18 clock
114 //cycles. SW9 is then set low. SW7-0 is set to a value not in RAM. SW9 is
115 //set high for 20 clock cycles, then low. SW9 is set high for 20 cycles and
116 //then low for a second time to see if the systems FSM will transition correctly.
117 //SW7-0 is then set to something in the RAM again. KEY0 is set low then high
118 //for 14 cycles to see if the system will reset correctly without a state
119 //transition. Finally, SW9 is set high for 20 clock cycles then low.
120 module DE1_SoC_testbench();
121       logic CLOCK_50;
122       logic [3:0] KEY;
123       logic [9:0] SW;
124       logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
125
126       DE1_SoC dut(.HEX0, .HEX1, .HEX2, .HEX3, .HEX4, .HEX5, .KEY, .SW, .CLOCK_50);
127
128       parameter CLOCK_PERIOD=100;
129       initial begin
130          CLOCK_50 <= 0;
131          forever #(CLOCK_PERIOD/2) CLOCK_50 <= ~CLOCK_50; // Forever toggle the clock
132       end
133
```

```
133
134         initial begin
135             SW[7:0] <= 8'b00001100;     repeat(2)   @(posedge    CLOCK_50);
136             SW[9] <= 0;                 repeat(2)   @(posedge    CLOCK_50);  //start = 0
137             KEY[0] <= 0;                repeat(2)   @(posedge    CLOCK_50);  //reset
138             KEY[0] <= 1;                repeat(14)  @(posedge    CLOCK_50);
139             SW[9] <= 1;                 repeat(18)  @(posedge    CLOCK_50); //start = 1
140             SW[9] <= 0;                 repeat(2)   @(posedge    CLOCK_50);
141             SW[7:0] <= 8'b11111111;     repeat(2)   @(posedge    CLOCK_50);
142             SW[9] <= 1;                 repeat(20)  @(posedge    CLOCK_50);
143             SW[9] <= 0;                 repeat(2)   @(posedge    CLOCK_50);
144             SW[9] <= 1;                 repeat(20)  @(posedge    CLOCK_50);
145             SW[9] <= 0;                 repeat(2)   @(posedge    CLOCK_50);
146
147             SW[7:0] <= 8'b00000000;     repeat(2)   @(posedge    CLOCK_50);
148             KEY[0] <= 0;                repeat(2)   @(posedge    CLOCK_50);  //reset
149             KEY[0] <= 1;                repeat(14)  @(posedge    CLOCK_50);
150
151             SW[9] <= 1;                 repeat(20)  @(posedge    CLOCK_50);
152             SW[9] <= 0;                 repeat(2)   @(posedge    CLOCK_50);
153
154             $stop; // End the simulation.
155         end
156     endmodule
```

Module used by both tasks

3.A) paramDFF.sv

```
1   //Garrett Tashiro
2   //October 18, 2021
3   //EE 371
4   //Lab 2, Task 2.3
5
6   //doubleD has 1-bit clk, reset, and press as inputs, and
7   //returns 1-bit out. This is a parameterized module to be able
8   //to change the number of bits being passed. This module is a double
9   //DFF (two in series) that takes the input signal from a switches, or
10  //buttons to prevent metastability.
11  module paramDFF #(parameter itsy = 1)(clk, reset, press, out);
12        input logic                 reset, clk;
13        input logic [itsy - 1:0]    press;
14        output logic [itsy - 1:0]   out;
15
16        logic [itsy - 1:0] temp1;
17
18        //always_ff replicates a double DFF. The input press goes into the
19        //first DFF and the output from the first DFF is the input for the
20        //second DFF.
21        always_ff @(posedge clk) begin
22           if (reset) begin
23              temp1 <= '0;
24              out   <= '0;
25           end
26
27           else begin
28              temp1 <= press;
29              out   <= temp1;
30         end
31        end
32  endmodule
33
```

3.B) paramDFF.sv (testbench)

```
34  //paramDFF_testbench tests the behaivor of a parameterized double DFF
35  //module. The test first sets reset to high and then low. It then tests
36  //4 different values being passed through the DFF's in series. The updated
37  //output takes two clock cycles, so wait a few clock cycles at the end.
38  module paramDFF_testbench();
39        logic          reset, clk;
40        logic [3:0]    press, out;
41
42        paramDFF #(.itsy(4)) dut(.clk, .reset, .press, .out);
43
44        parameter clk_PERIOD = 100;
45        initial begin
46              clk <= 0;
47              forever #(clk_PERIOD/2) clk <= ~clk;// Forever toggle the clk
48        end
49
50        initial begin
51           reset <= 1;                          repeat(1)   @(posedge    clk);
52           reset <= 0; press <= 4'b1000;        repeat(1)   @(posedge    clk);
53           press <= 4'b0011;                    repeat(1)   @(posedge    clk);
54           press <= 4'b1100;                    repeat(1)   @(posedge    clk);
55           press <= 4'b0100;                    repeat(1)   @(posedge    clk);
56                                                repeat(3)   @(posedge    clk);
57
58           $stop; // End the simulation.
59        end
60  endmodule
```