ECE 5722 Homework 1

By Bruce Huynh

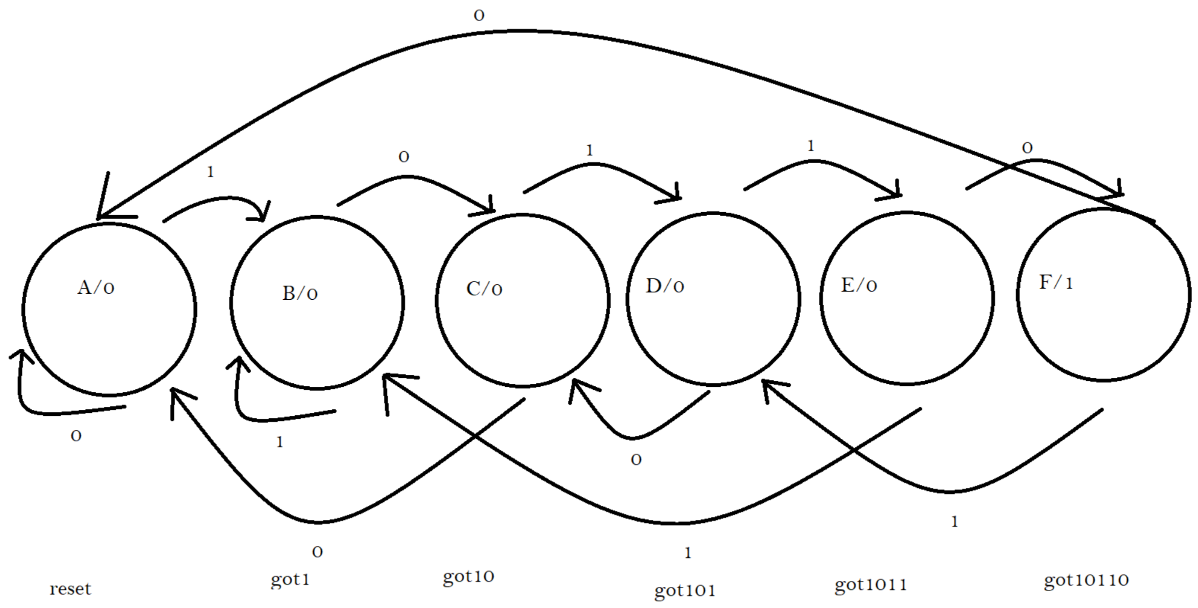
9/5/2021

Introduction:

In this assignment we were tasked to build a sequence detector that would detect the sequence: “10110” and output high on w if it detected it. Afterwards, a counter would be initiated that counts 32 clock cycles and flows back into the sequence detector to tell it to start finding the sequence once again.

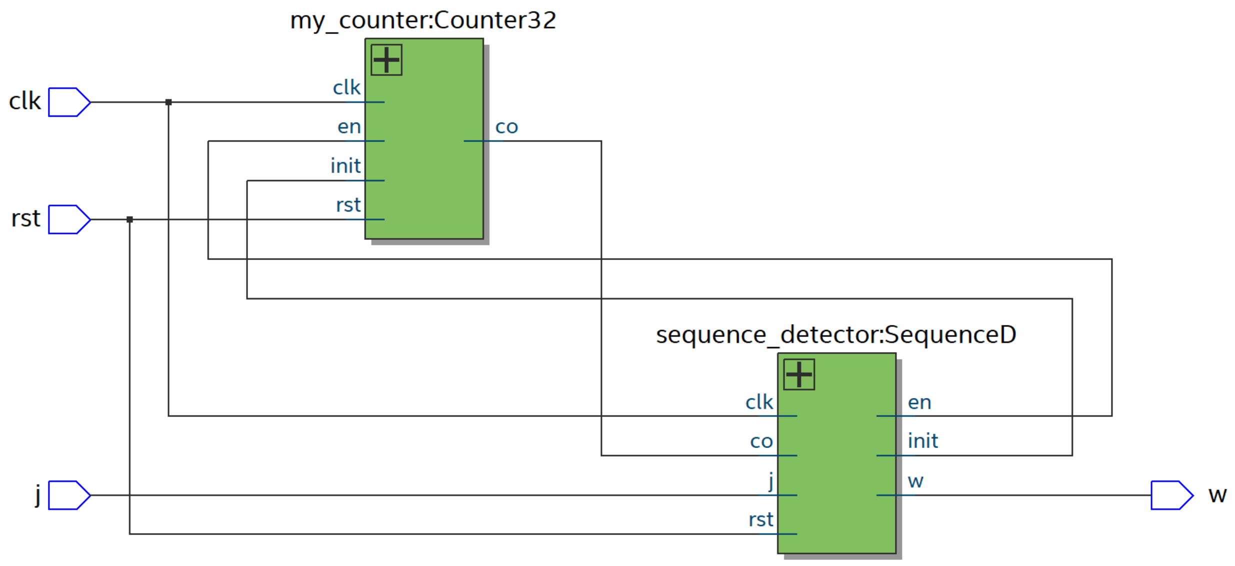
A.) State Diagram:

To easily find inputs and outputs necessary for the sequence detector, a state diagram is needed. At each state, it can receive a 1 or 0 which leads it to the next state. Knowing this, I mapped out each state.



B.)

The following is the top-level RTL design:



The accompanying code for the top-level module:

```
module moore_with_counter(input j, clk, rst, output w);

     wire detector_en_to_counter;
     wire detector_init_to_counter;
     wire counter_clockout_to_detector;

     my_counter Counter32 (.clk (clk), .rst (rst), .en
(detector_en_to_counter), .init (detector_init_to_counter), .co
(counter_clockout_to_detector));

     sequence_detector SequenceD (.j (j), .clk (clk), .rst (rst),
.co (counter_clockout_to_detector), .w (w), .init
(detector_init_to_counter), .en (detector_en_to_counter));

endmodule
```

Code for the sequence detector:

```
module sequence_detector(input j, clk, rst, co, output w, init, en);

     parameter [4:0] reset = 3'b000, got1 = 3'b001, got10 = 3'b010,
got101 = 3'b011, got1011 = 3'b100, got10110 = 3'b101;
     reg[4:0] pres_state, next_state;
     reg[7:0] firsttime; //variable to detect first j input
     reg pulse, outputon;
     //pulse enables the module to run
    //outputon is the output of the system that is kept high until
    //the module is initiated again


     initial firsttime = 0;

     always @ (posedge clk) begin
          firsttime = firsttime + 1;
          //detects either the clock out from the counter or the
          //first time j is detected
          if (co == 1 || firsttime == 1) begin
               pulse = 1;
               outputon = 0;
               end
          if (next_state == got10110) begin
               pulse = 0;
               outputon = 1;
               End
          //no longer the first time after this if statement
          if (firsttime == 8'b11111111)

               firsttime = 8'b00000010;

     end
     //state machine logic
     always @ (pres_state or j) begin : combinational
     next_state = 0;
```

```
      if (pulse == 1) begin
           case (pres_state)

                reset: begin
                if (j == 1'b1)
                next_state = got1;
                else
                next_state = reset;
                end

                got1: begin
                if (j == 1'b0)
                next_state = got10;
                else
                next_state = got1;
                end

                got10: begin
                if (j == 1)
                next_state = got101;
                else
                next_state = reset;
                end

                got101: begin
                if (j == 1)
                next_state = got1011;
                else
                next_state = got10;
                end

                got1011: begin
                if (j == 0)
                next_state = got10110;
                else
                next_state = got1;
                end

                got10110: begin
                if (j == 0)
                next_state = reset;
                else
                next_state = got101;
                end

                default next_state = reset;

           endcase
      end

      end

      always @ (posedge clk or posedge rst) begin
           if (rst)
                pres_state <= reset;
           else
                pres_state <= next_state;
      end

      assign w = outputon;
      assign init = (pres_state == got10110) ? 1 : 0; //tells the
counter when to start

      assign en = w; //enables the counter

endmodule
```

Code for the counter:

```
module my_counter (input clk, rst, en, init, output co);

reg [5:0] current;
reg pulse;
```

```
always @ (posedge clk) begin

      if (init)
            pulse = 1'b1; //keeps counter

      if (current == 6'b100000)
            pulse = 1'b0; //counter is off after 32 clock cycles
end

always @ (posedge clk) begin

      if (rst || current == 6'b100000)
            current <= 6'b000000;
      else if (en && pulse)
            current = current + 1'b1; //counting mechanism

end


assign co = (current == 6'b100000) ? 1 : 0; //counter output

endmodule
```

C.) Testbenches

For each of the smaller modules, I made testbenches for them. Very similar testbenches will be used in the top-level module.

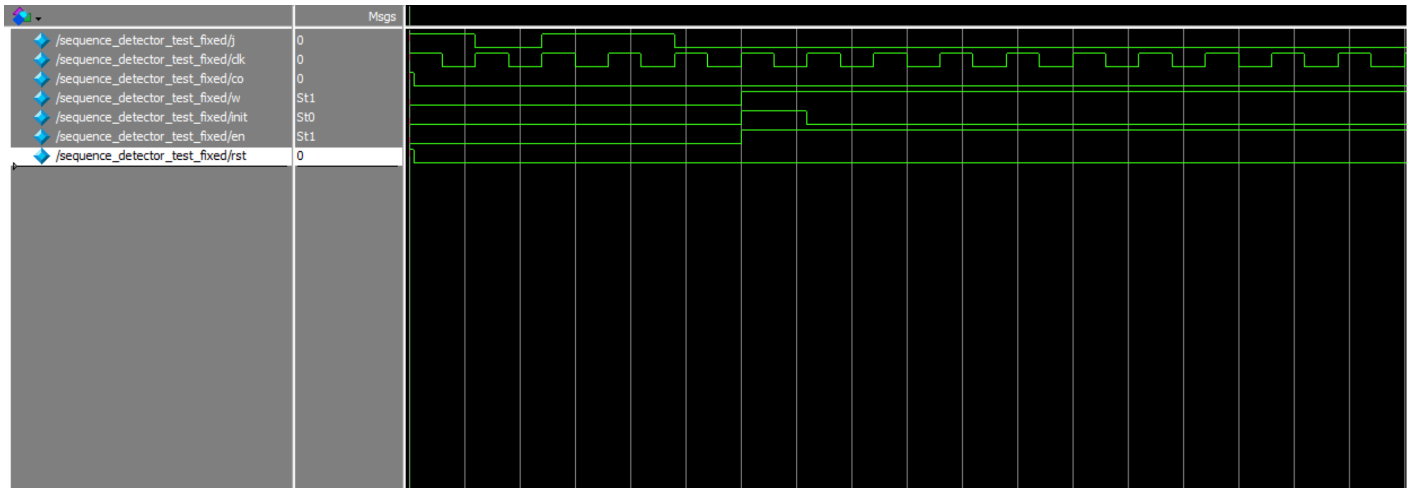
Sequence detector (fixed "10110") test:

```
module sequence_detector_test_fixed;

      reg j, clk, rst, co;
      wire w, init, en;

      sequence_detector uut (j, clk, rst, co, w, init, en);
```

```
    initial begin

          co = 1'b1; clk = 1'b1; rst = 1'b1; j = 1'b1;

    end

    initial #5 rst = 1'b0;
    initial #5 co = 1'b0;
    always #30 clk = ~clk;



    always begin
    j = 1'b1; #60; j = 1'b0; #60; j = 1'b1; #60; j = 1'b1; #60; j = 1'b0; #60; #1200; co = 1'b1; #2; co = 1'b0;
    end
```

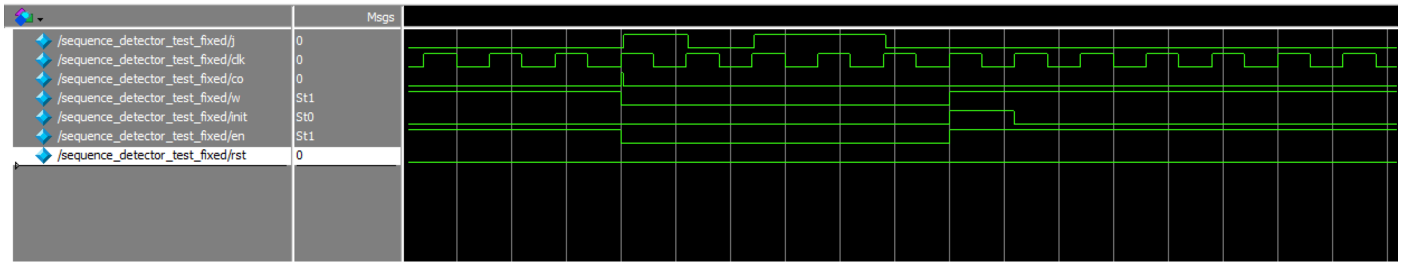
Waveform:

(order: j, clk, co, w, init, en, rst)



```
                                   Msgs
/sequence_detector_test_fixed/j    0
/sequence_detector_test_fixed/clk  0
/sequence_detector_test_fixed/co   0
/sequence_detector_test_fixed/w    St1
/sequence_detector_test_fixed/init St0
/sequence_detector_test_fixed/en   St1
/sequence_detector_test_fixed/rst  0
```

The module is initialized with a small pulse from *co* (clock out from the counter). When the module detects “10110”, *init* is pulsed while *w* and *en* is driven high.


/sequence_detector_test_fixed/j
/sequence_detector_test_fixed/clk
/sequence_detector_test_fixed/co
/sequence_detector_test_fixed/w
/sequence_detector_test_fixed/init
/sequence_detector_test_fixed/en
/sequence_detector_test_fixed/rst
Msgs
0
0
0
St1
St0
St1
0

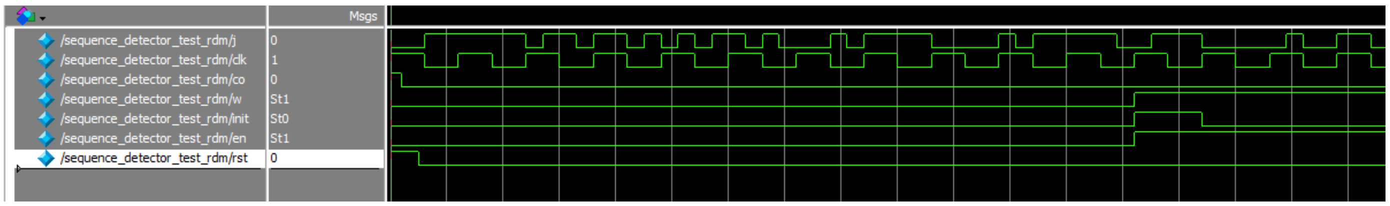
When *co* is pulsed again, the cycle continues like seen above.

Sequence detector with random j inputs test:

```
module sequence_detector_test_rdm;

     reg j, clk, rst, co;
     wire w, init, en;

     sequence_detector uut (j, clk, rst, co, w, init, en);

     initial begin

          co = 1'b1; clk = 1'b1; rst = 1'b1; j = 1'b0;

     end

     initial #25 rst = 1'b0;
     initial #10 co = 1'b0;

     always #30 clk = ~clk;
     always #15 j = $random;


endmodule
```

Waveform:

(order: j, clk, co, w, init, en, rst)


Msgs
/sequence_detector_test_rdm/j 0
/sequence_detector_test_rdm/clk 1
/sequence_detector_test_rdm/co 0
/sequence_detector_test_rdm/w St1
/sequence_detector_test_rdm/init St0
/sequence_detector_test_rdm/en St1
/sequence_detector_test_rdm/rst 0

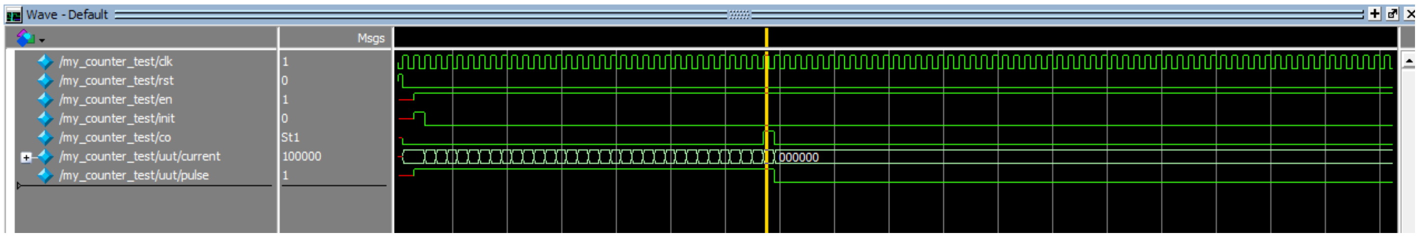
The module is receiving random inputs and waits until it gets “10110” in which it pulses *init* and turns *w* and *en* high.

Counter test:

```
module my_counter_test;
      reg clk, rst, en, init;
      reg current, pulse;
      wire co;
      my_counter uut (clk, rst, en, init, co);

      initial begin
      clk = 1'b0;
      rst <= 1'b1;
      #5 rst <= 1'b0;
      #10 en <= 1'b1;
      init = 1'b1;
      end

      always @ (posedge clk) begin
      #20 init = 1'b0;
      #2000;
      #20 init = 1'b1;
      #20 init = 1'b0;
      end


always #5 clk = ~clk;


endmodule
```

Waveform:

(order: clk, rst, en, init, co, current, pulse)



	Msgs
/my_counter_test/clk	1
/my_counter_test/rst	0
/my_counter_test/en	1
/my_counter_test/init	0
/my_counter_test/co	St1
/my_counter_test/uut/current	100000
/my_counter_test/uut/pulse	1

When we have a constant *en* and an *init* pulse, the clock will begin to count. It will count up to 32 (8’b10000000) where it will pulse *co* and then stop counting until it is initiated again.

Top-level testbench (moore_with_counter):

Like mentioned earlier, the top-level module uses the same test as the sequence detector as seen below:

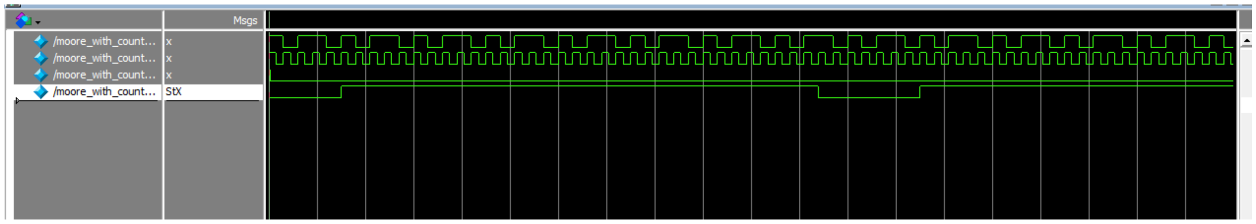
Fixed pattern:

```
module moore_with_counter_test_fixed;

     reg j, clk, rst;
     wire w;

     moore_with_counter uut (j, clk, rst, w);


     initial begin

          clk = 1'b1; rst = 1'b1; j = 1'b1;

     end

     initial #5 rst = 1'b0;
     always #30 clk = ~clk;
```

```
    always begin
    j = 1'b1; #60; j = 1'b0; #60; j = 1'b1; #60; j = 1'b1; #60; j =
1'b0; #60;
    end

endmodule
```

Waveform:

(order: j, clk, rst, w)



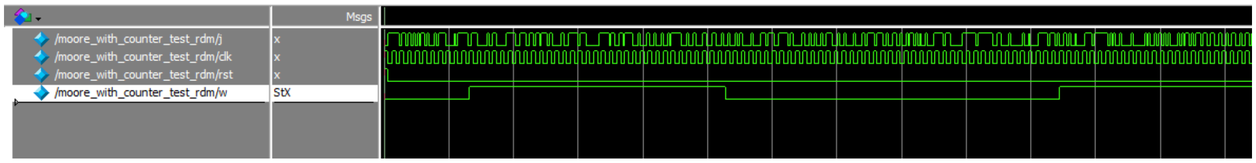
As seen in the waveform, when the first j input is received, the module will start looking for the "10110" sequence. Once it has found it, *w* gets turned on and stays on for 32 clock cycles. After that, *w* turns off and the search begins once again. The pattern above is fixed and shows *w* being high twice. If you count the clock pulses after *w* gets turned on high, you will be able to count 32 clock pulses.

Random pattern:

```
module moore_with_counter_test_rdm;

    reg j, clk, rst, co;
    wire w;

    moore_with_counter uut (j, clk, rst, w);

    initial begin

        clk = 1'b1; rst = 1'b1; j = 1'b0;
    end

    initial #25 rst = 1'b0;

    always #30 clk = ~clk;
    always #15 j = $random;


endmodule
```

Waveform:

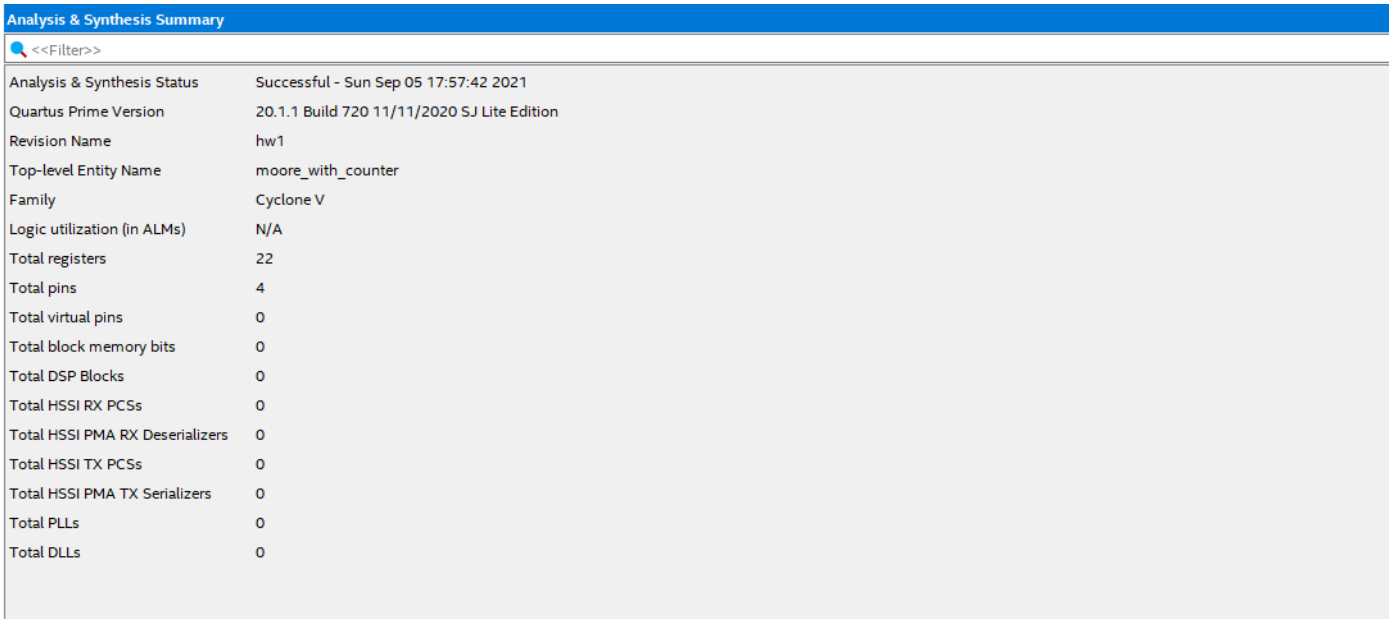
(order: j, clk, rst, w)



```
                              Msgs
/moore_with_counter_test_rdm/j    x
/moore_with_counter_test_rdm/clk  x
/moore_with_counter_test_rdm/rst  x
/moore_with_counter_test_rdm/w    StX
```

Like earlier, the module will detect “10110” and output high on *w* when it does. *W* remains high until 32 counts where it will go back to 0. The search will continue until the sequence is detected again. The waveform above shows this happening twice with random j inputs. If you count the clock pulses after *w* gets turned on high, you will be able to count 32 clock pulses.

Synthesis:



Analysis & Synthesis Summary

<<Filter>>

Analysis & Synthesis Status	Successful - Sun Sep 05 17:57:42 2021
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	hw1
Top-level Entity Name	moore_with_counter
Family	Cyclone V
Logic utilization (in ALMs)	N/A
Total registers	22
Total pins	4
Total virtual pins	0
Total block memory bits	0
Total DSP Blocks	0
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0
Total DLLs	0