Lab 4

Tasks to be done in this Lab:

1. Design and implement an overlapping sequence detector using Mealy FSMand verify its functionality using testbench.
2. Design a clock pulse generator.
3. Test the functionality of the design on the Basys 3 Board (remote access) using VIO IP.

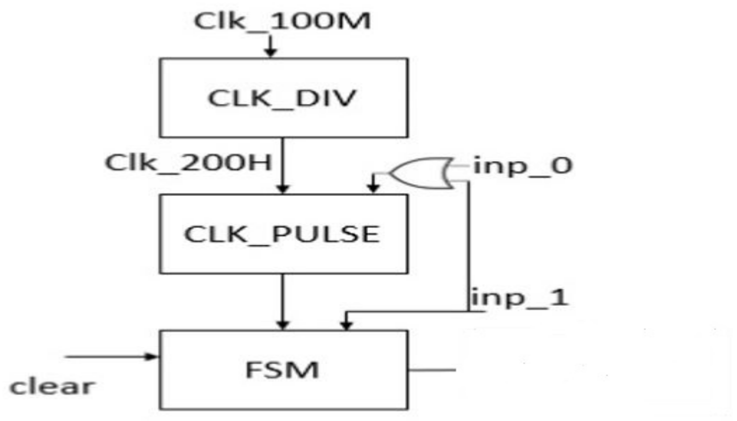
Topics to Explore:1) Mealy and Moore FSM implementation, 2) Use of 3 Always blocks, 3) Difference between overlapping and non-overlapping

In this Lab, we will look into how to implement Mealy FSM using Verilog. As an example, we will be designing a 11011 sequence detector with an overlap.

Note: Please revise the Mealy and Moore FSM and state diagrams as studied in Digital Circuits Course.

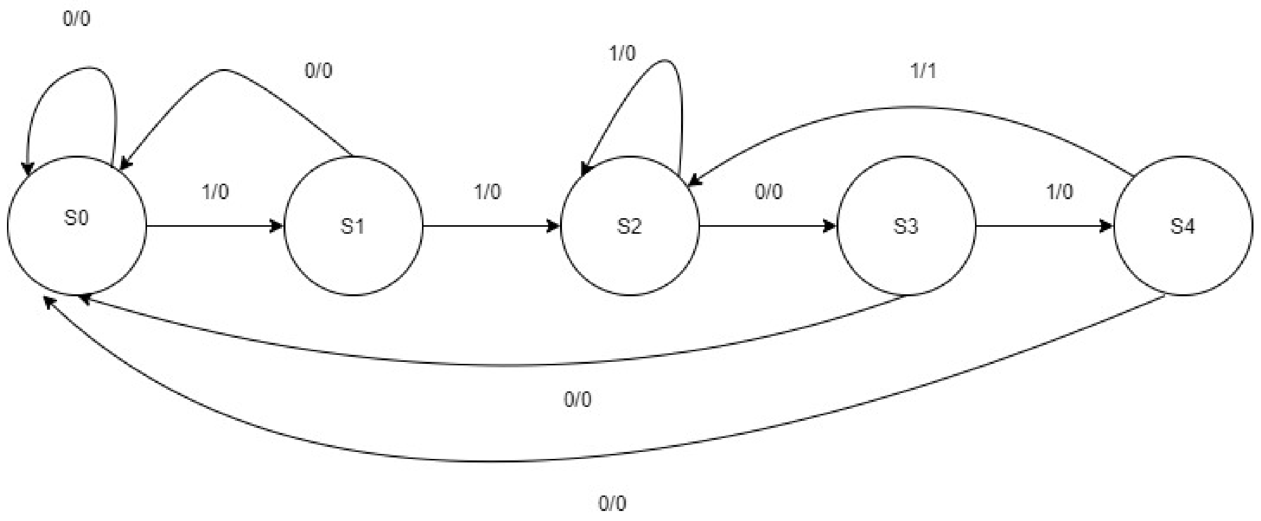
- **Design details:**
  - ❖ The input sequence will be given with the help of two push buttons.
  - ❖ One of the push buttons will correspond to an input of '1' and the other to an input of '0'.
  - ❖ Use a clean pulse from buttons to sample the inputs of the sequence detector FSM module.
  - ❖ Whenever the desired sequence is detected, the out pin should go high.
  - ❖ Add clear functionality with push-button.
  - ❖ Display the current state on the VIO.
  - ❖ The out pin should remain low whenever the input sequence is invalid.

The first thing we need to consider is the **clock pulse generation** for the FSM. We will use a 200 Hz clock to generate a clock pulse, which transitions from 0 to 1 whenever any of the data push buttons is pressed. The following block diagram will give you an abstract view of the implementation(all the ports are not shown).



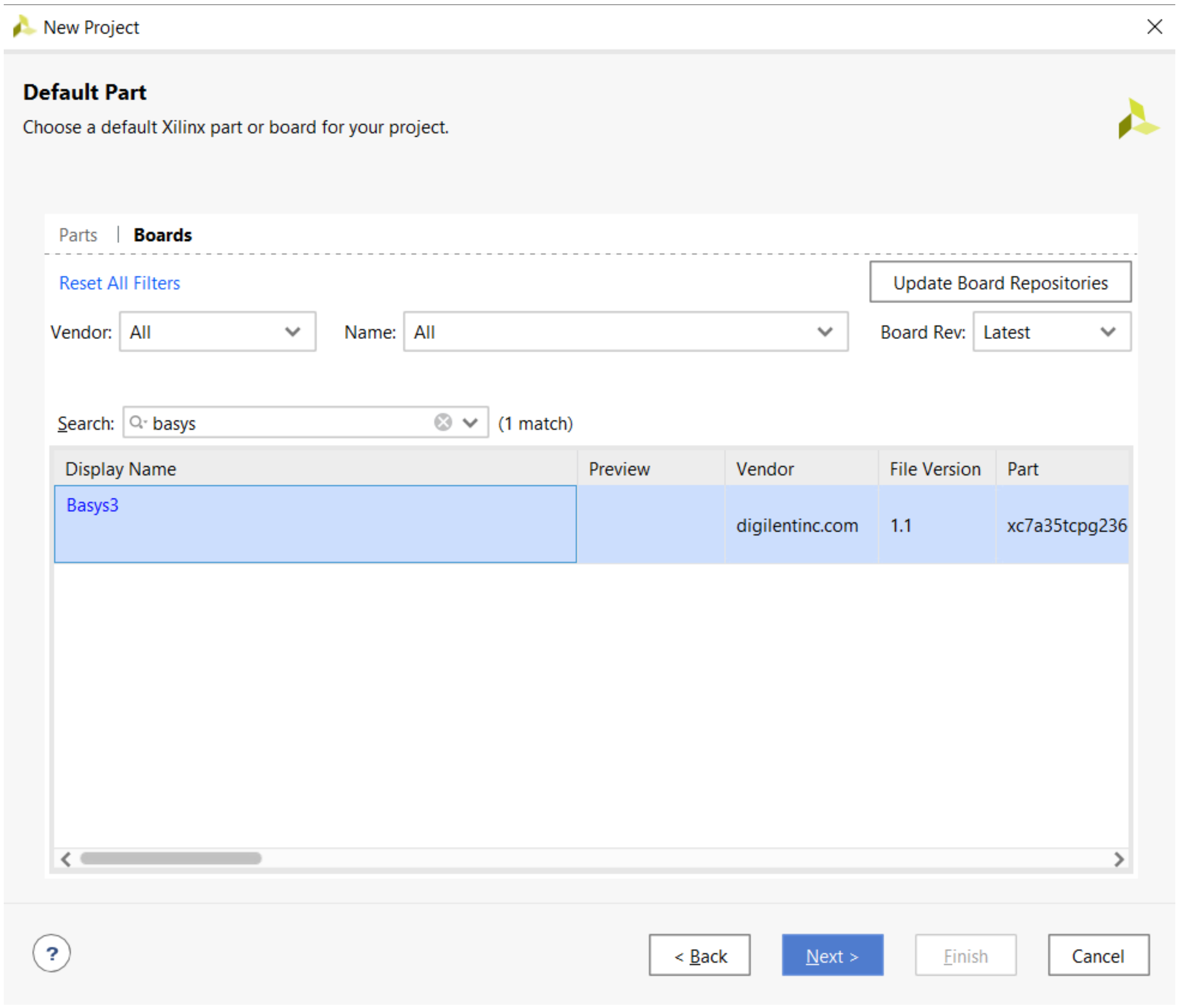
FSM Implementation

The state diagram for the Mealy type sequence detector(11011) with overlap is shown below:



Step 1:Createthe Top Module for the Sequence Detector, add and instantiate the clocking IP and clock divider module.

1. Create a New Project.
2. Select Project Type as RTL project.
3. In the Add Sources window, first, we will create the Verilog file for the top module(top_seq.v).
4. You don't need to add constraints at this time; we will do it in the later stages.
5. Add the board as **Basys3,** as shown in the screenshot below


New Project

Default Part

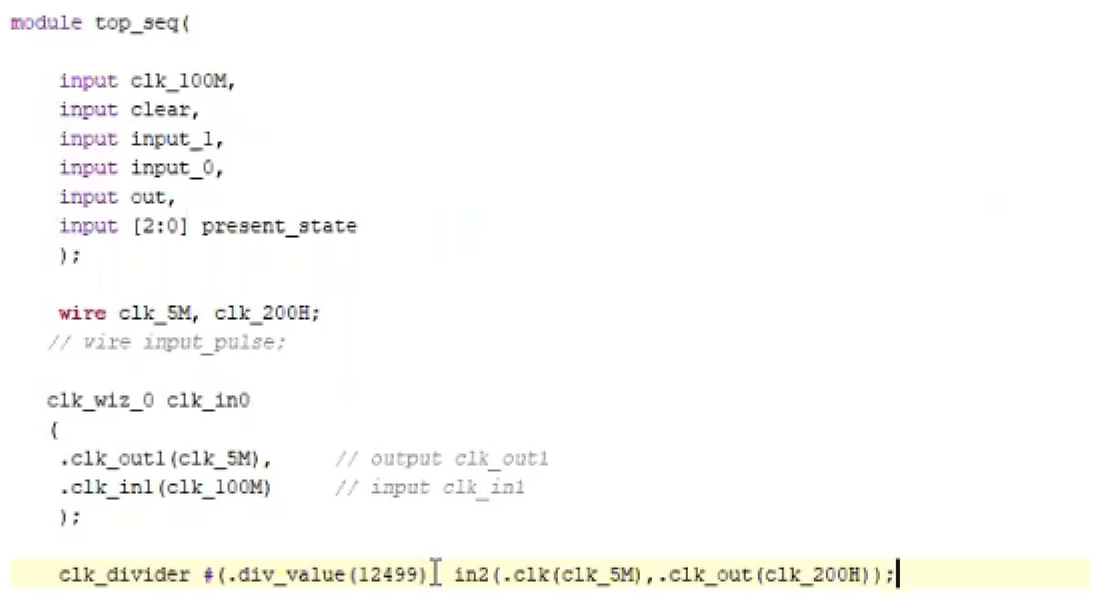
Choose a default Xilinx part or board for your project.

Vendor: All   Name: All   Board Rev: Latest

Search: basys (1 match)

Display Name	Preview	Vendor	File Version	Part
Basys3		digilentinc.com	1.1	xc7a35tcpg236

6. Make the port declaration, two input ports(clk_100M,clear,inp_0,inp_1) of 1 bit each, and one output port "out" of 1 bit and one output port present_state of 3 bits.
7. Add clocking IP and instantiate it.
8. Add the clock divider module we made in Lab 2. While adding design sources, you can click on add file instead of creating a file to add an already available module.
9. After completing Step 6, your project must have a module top_seq, which should have the module and port declaration and **Clocking IP and clock divider instantiation. It should look as follows:-**

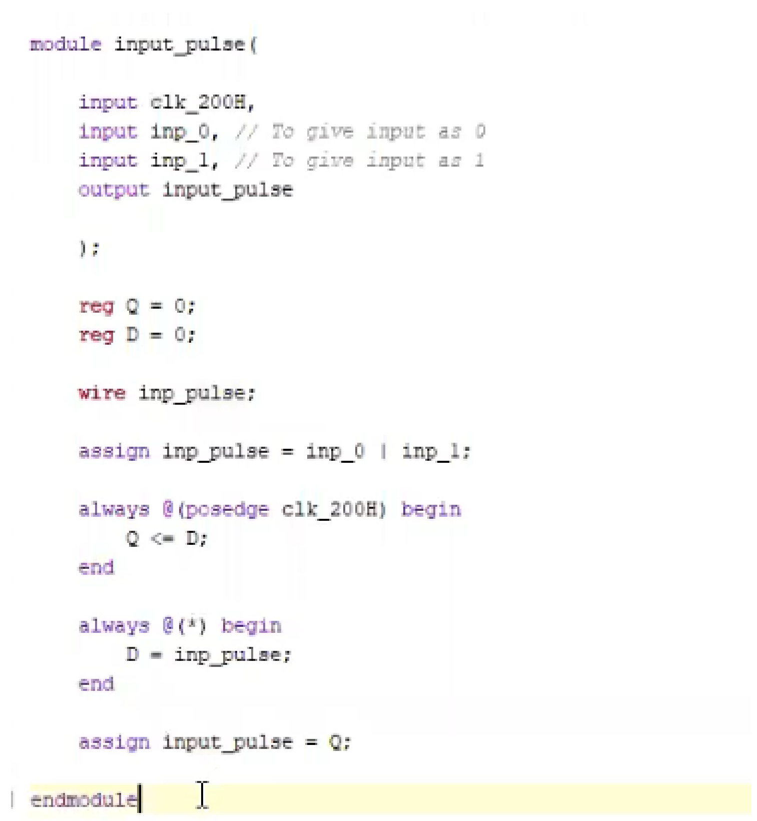


```
module top_seq(

    input clk_100M,
    input clear,
    input input_1,
    input input_0,
    input out,
    input [2:0] present_state
    );

    wire clk_5M, clk_200H;
   // wire input_pulse;

   clk_wiz_0 clk_in0
   (
    .clk_out1(clk_5M),      // output clk_out1
    .clk_in1(clk_100M)      // input clk_in1
    );

    clk_divider #(.div_value(12499)) in2(.clk(clk_5M),.clk_out(clk_200H));
```

We have re-used the clk_divider from the previous Lab and passed the appropriate div_val.

Step 2: Add a new design source to define the functionality of the input_pulse.

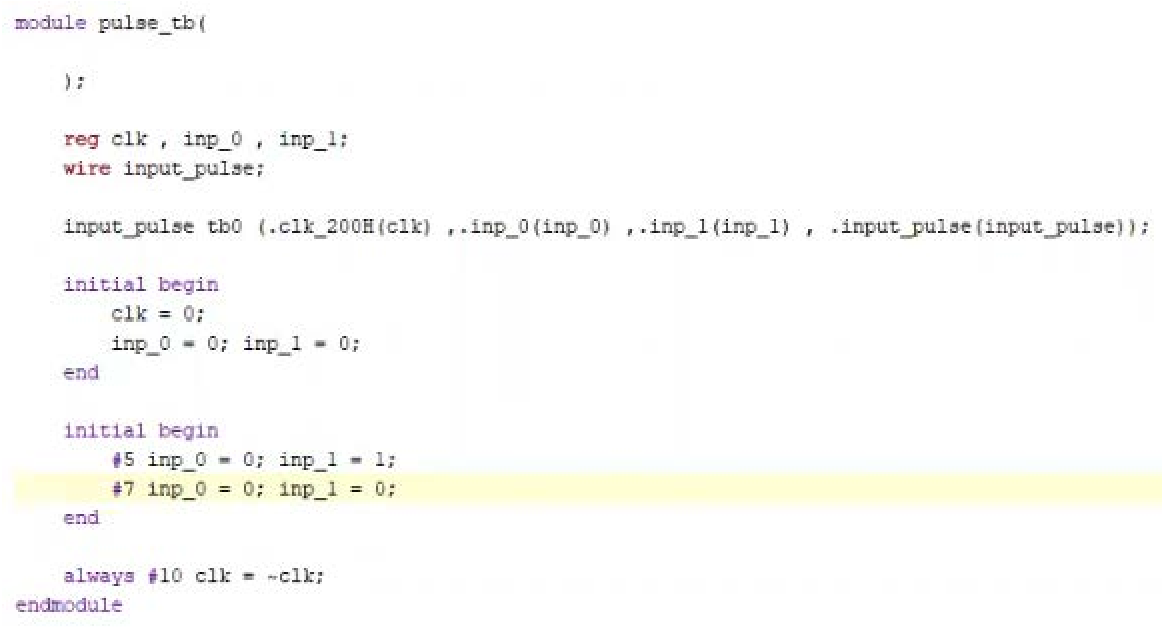
1. Now we will add the functionality of the input_pulse. To do so, add design sources and create one more Verilog file input_pulse.v. Add the three input ports (clk_200H,inp_0,inp_1) of 1 bit each and one output port input_pulse of 1 bit.



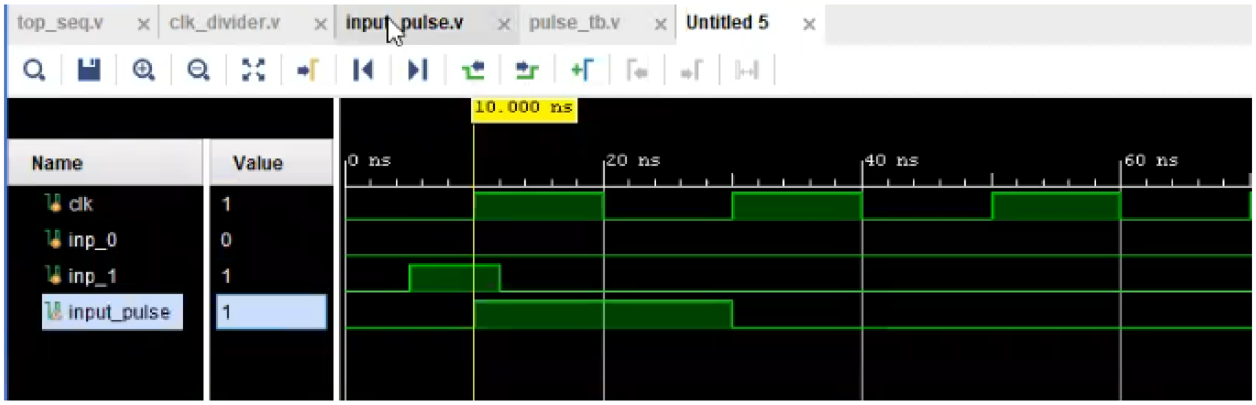
```
module input_pulse(

    input clk_200H,
    input inp_0, // To give input as 0
    input inp_1, // To give input as 1
    output input_pulse

    );

    reg Q = 0;
    reg D = 0;

    wire inp_pulse;

    assign inp_pulse = inp_0 | inp_1;

    always @(posedge clk_200H) begin
        Q <= D;
    end

    always @(*) begin
        D = inp_pulse;
    end

    assign input_pulse = Q;

endmodule
```

Before moving on to the next step, first, let's check the functionality of our input pulse generator and see how the debouncing circuit is working.

Make a testbench with the following code:



```
module pulse_tb(

    );

    reg clk , inp_0 , inp_1;
    wire input_pulse;

    input_pulse tb0 (.clk_200H(clk) ,.inp_0(inp_0) ,.inp_1(inp_1) , .input_pulse(input_pulse));

    initial begin
        clk = 0;
        inp_0 = 0; inp_1 = 0;
    end

    initial begin
        #5 inp_0 = 0; inp_1 = 1;
        #7 inp_0 = 0; inp_1 = 0;
    end

    always #10 clk = ~clk;
endmodule
```

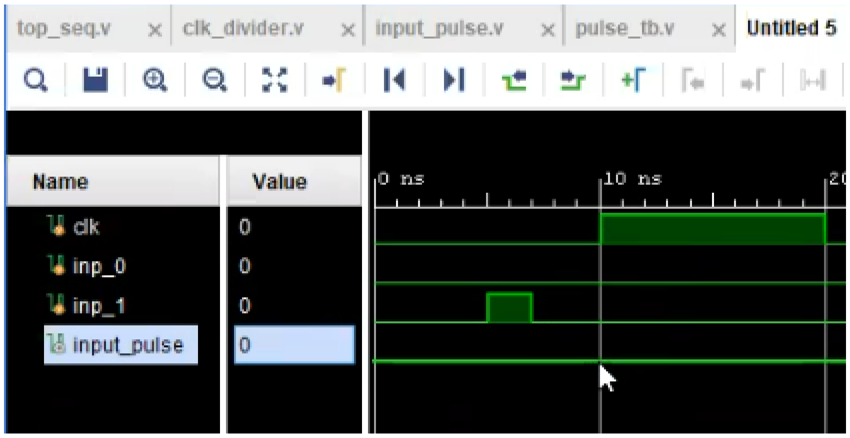


Notice the input_pulse as output.

With the following change in the testbench :-

```
initial begin
    #5 inp_0 = 0; inp_1 = 1;
    #2 inp_0 = 0; inp_1 = 0;
end
```

We can see the simulation:-



Since the time period was very low, this is not considered, and hence an input pulse would not be fed.

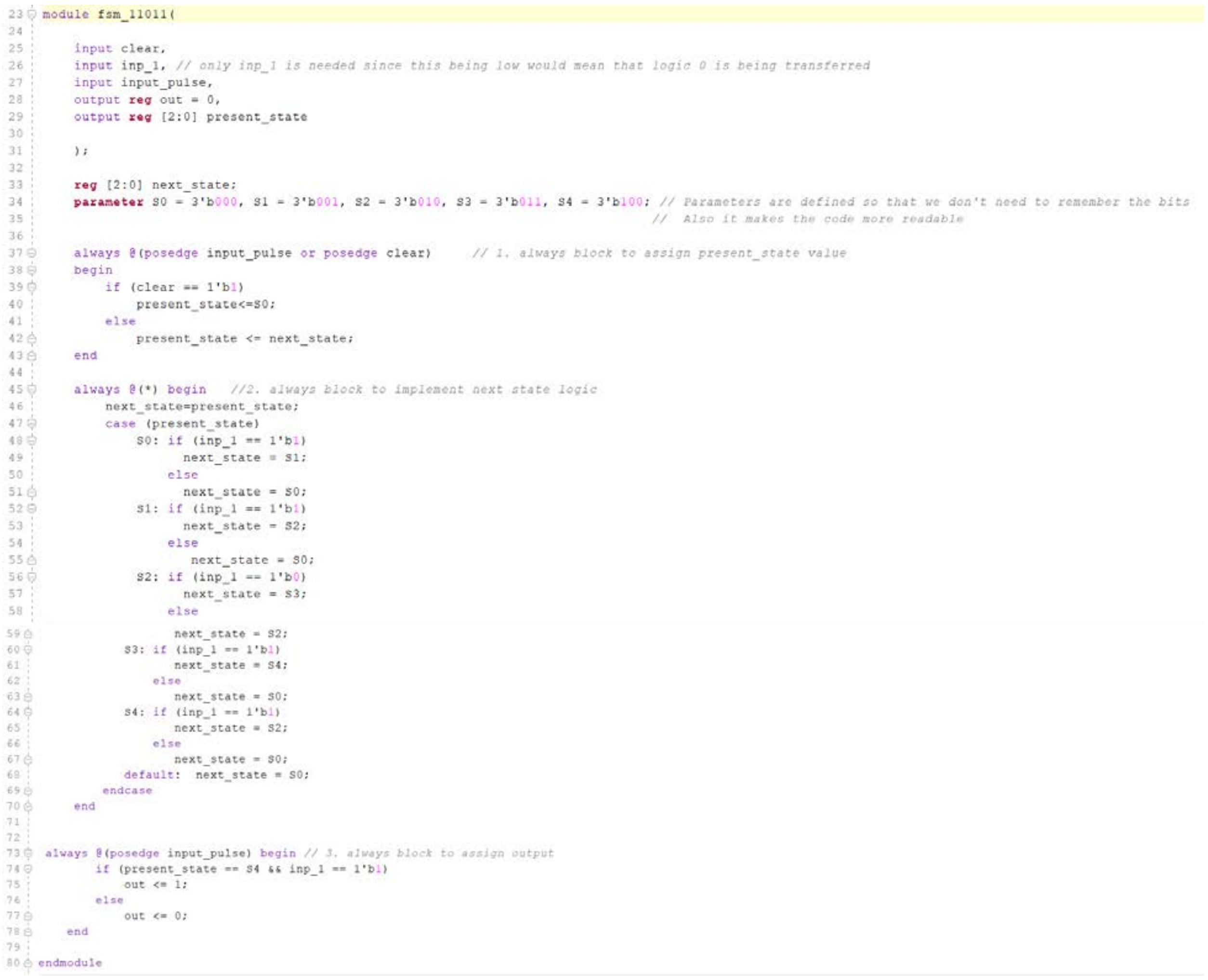
Look at the above waveform and match the results with the code.

Step 3: Create a module to define the functionality of the FSM

An FSM implementation in Verilog must be done in the way described in the following text. An FSM module will contain three always block.

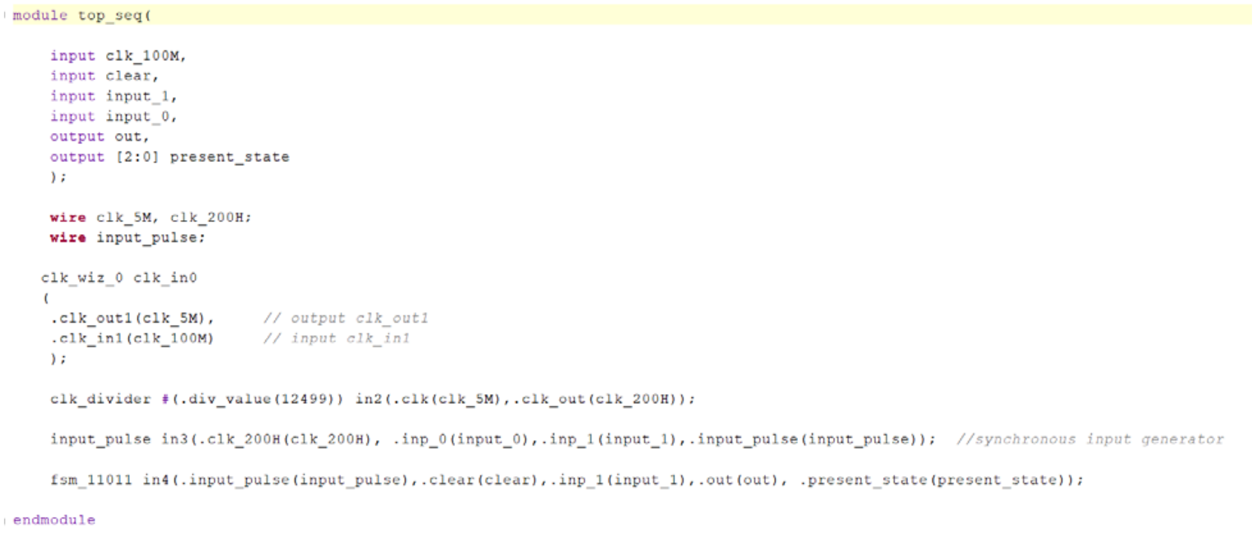
1. One always block for assigning the present state (sequential block)
2. One always block for implementing the next state logic (combinational block)→ Written using the state diagram.
3. One always block for assigning the output.

Please go through the following code and try finding these always blocks.



```verilog
module fsm_11011(

    input clear,
    input inp_1, // only inp_1 is needed since this being low would mean that logic 0 is being transferred
    input input_pulse,
    output reg out = 0,
    output reg [2:0] present_state

    );

    reg [2:0] next_state;
    parameter S0 = 3'b000, S1 = 3'b001, S2 = 3'b010, S3 = 3'b011, S4 = 3'b100; // Parameters are defined so that we don't need to remember the bits
                                                                               //  Also it makes the code more readable

    always @(posedge input_pulse or posedge clear)     // 1. always block to assign present_state value
    begin
        if (clear == 1'b1)
            present_state<=S0;
        else
            present_state <= next_state;
    end

    always @(*) begin   //2. always block to implement next state logic
        next_state=present_state;
        case (present_state)
            S0: if (inp_1 == 1'b1)
                   next_state = S1;
                 else
                   next_state = S0;
            S1: if (inp_1 == 1'b1)
                   next_state = S2;
                 else
                    next_state = S0;
            S2: if (inp_1 == 1'b0)
                   next_state = S3;
                 else
                   next_state = S2;
            S3: if (inp_1 == 1'b1)
                   next_state = S4;
                 else
                   next_state = S0;
            S4: if (inp_1 == 1'b1)
                   next_state = S2;
                 else
                   next_state = S0;
            default:  next_state = S0;
        endcase
    end


always @(posedge input_pulse) begin // 3. always block to assign output
        if (present_state == S4 && inp_1 == 1'b1)
            out <= 1;
        else
            out <= 0;
    end

endmodule
```

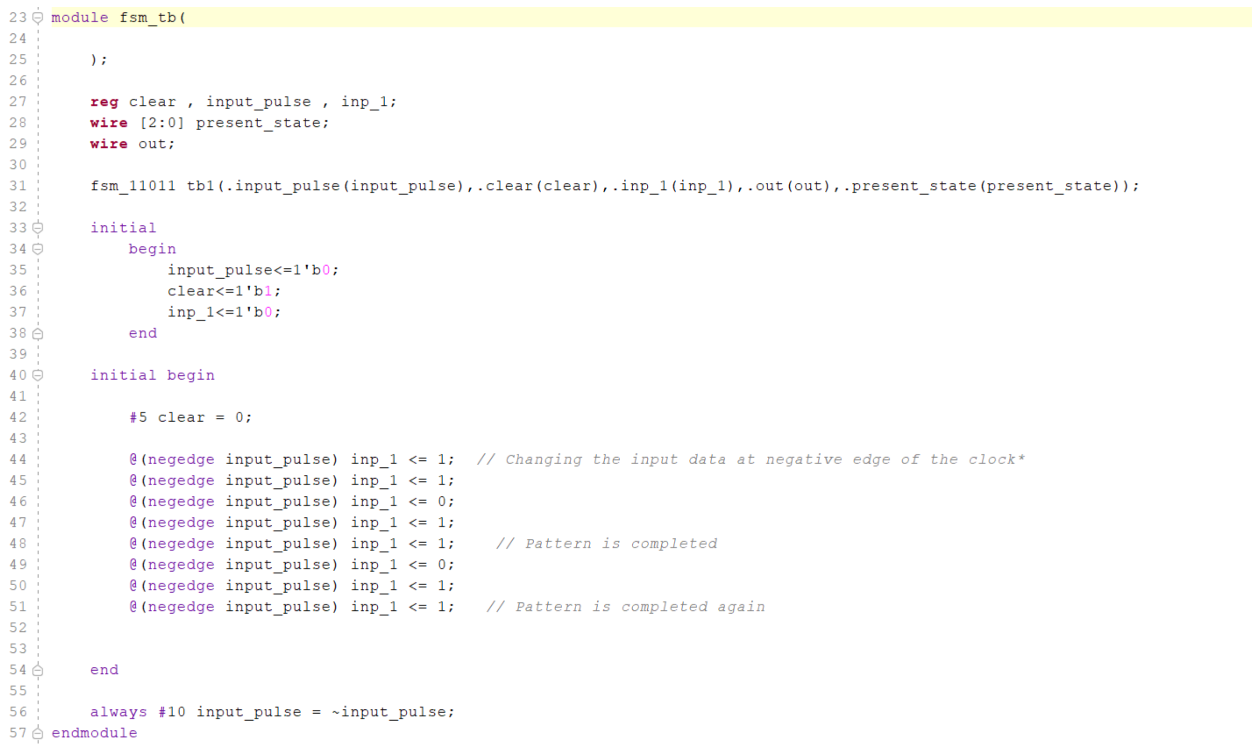
Next, we need to instantiate the FSM and clk_pulse module in our top module.



```verilog
module top_seq(

    input clk_100M,
    input clear,
    input input_1,
    input input_0,
    output out,
    output [2:0] present_state
    );

    wire clk_5M, clk_200H;
    wire input_pulse;

   clk_wiz_0 clk_in0
   (
    .clk_out1(clk_5M),     // output clk_out1
    .clk_in1(clk_100M)     // input clk_in1
    );

    clk_divider #(.div_value(12499)) in2(.clk(clk_5M),.clk_out(clk_200H));

    input_pulse in3(.clk_200H(clk_200H), .inp_0(input_0),.inp_1(input_1),.input_pulse(input_pulse));  //synchronous input generator

    fsm_11011 in4(.input_pulse(input_pulse),.clear(clear),.inp_1(input_1),.out(out), .present_state(present_state));

endmodule
```

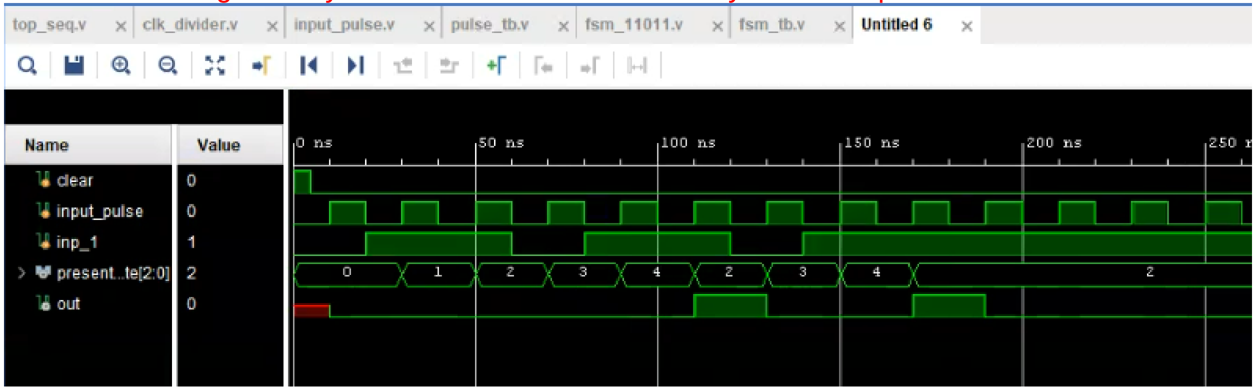
Step 4: Test the functionality of the Sequence Detector using testbench.

1. Add a simulation source and write the testbench code, as shown below:



```
module fsm_tb(

    );

    reg clear , input_pulse , inp_1;
    wire [2:0] present_state;
    wire out;

    fsm_11011 tb1(.input_pulse(input_pulse),.clear(clear),.inp_1(inp_1),.out(out),.present_state(present_state));

    initial
        begin
            input_pulse<=1'b0;
            clear<=1'b1;
            inp_1<=1'b0;
        end

    initial begin

        #5 clear = 0;

        @(negedge input_pulse) inp_1 <= 1;  // Changing the input data at negative edge of the clock*
        @(negedge input_pulse) inp_1 <= 1;
        @(negedge input_pulse) inp_1 <= 0;
        @(negedge input_pulse) inp_1 <= 1;
        @(negedge input_pulse) inp_1 <= 1;    // Pattern is completed
        @(negedge input_pulse) inp_1 <= 0;
        @(negedge input_pulse) inp_1 <= 1;
        @(negedge input_pulse) inp_1 <= 1;   // Pattern is completed again


    end

    always #10 input_pulse = ~input_pulse;
endmodule
```

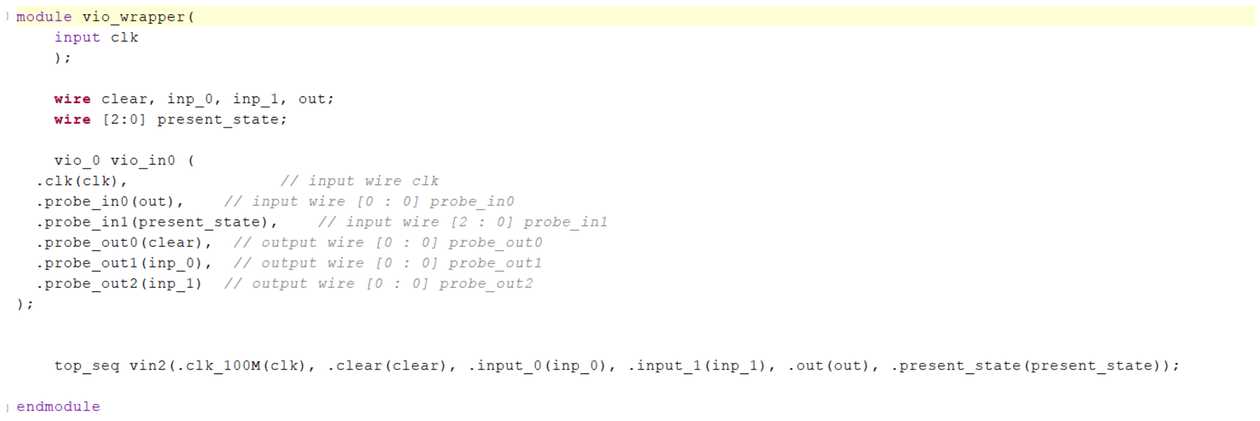
Note: We have instantiated the fsm_11011 module here and not the top_seq module. The reason being we only need to check the functionality of the Sequence detector.



2. Run the simulation and check the functionality of the sequence detector.

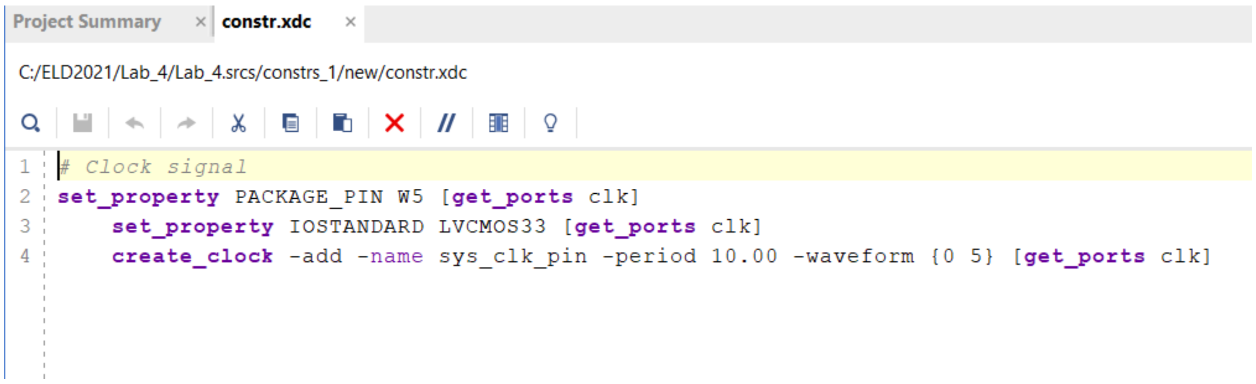
Step 5: Add VIO IP to your design to test it on hardware.

1. Create a vio_wrapper in the project in a similar way as we did in Lab2 and Lab 3.Thevio_wrapper module will look like, as shown below:



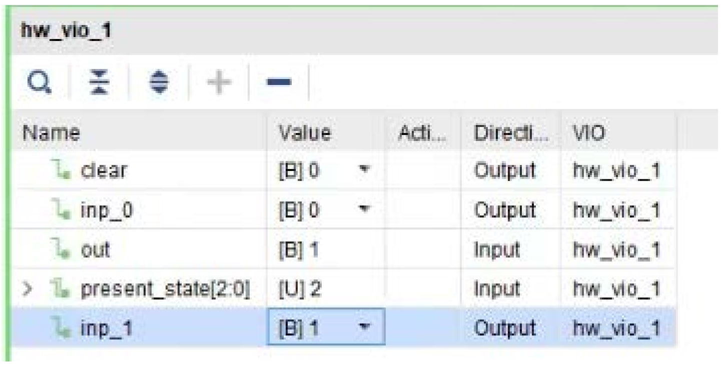
```
module vio_wrapper(
    input clk
    );

    wire clear, inp_0, inp_1, out;
    wire [2:0] present_state;

    vio_0 vio_in0 (
  .clk(clk),                // input wire clk
  .probe_in0(out),      // input wire [0 : 0] probe_in0
  .probe_in1(present_state),     // input wire [2 : 0] probe_in1
  .probe_out0(clear),   // output wire [0 : 0] probe_out0
  .probe_out1(inp_0),   // output wire [0 : 0] probe_out1
  .probe_out2(inp_1)   // output wire [0 : 0] probe_out2
);


    top_seq vin2(.clk_100M(clk), .clear(clear), .input_0(inp_0), .input_1(inp_1), .out(out), .present_state(present_state));

endmodule
```

2. Add the constraints file.



```
# Clock signal
set_property PACKAGE_PIN W5 [get_ports clk]
    set_property IOSTANDARD LVCMOS33 [get_ports clk]
    create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports clk]
```

3. Generate the bitstream.
4. Connect to the remote device using the steps mentioned in Lab 2.
5. Program the device. Once the programming is done, refresh the VIO probes.
6. Give the input sequence through inp_0 and inp_1 and check if the present_state is changing accordingly, and finally, the output should go high once the pattern 110011 is completed. **Note that you can use buttons by changing the input mode, but we have done the simulation with normal input pulses.**



Name	Value	Acti...	Directi...	VIO
clear	[B] 0		Output	hw_vio_1
inp_0	[B] 0		Output	hw_vio_1
out	[B] 1		Input	hw_vio_1
present_state[2:0]	[U] 2		Input	hw_vio_1
inp_1	[B] 1		Output	hw_vio_1

Deliverables for Lab 4 Submission(Graded):

1. .bit file and .ltx file of the design designs.
2. PDF with code, testbench, simulation screenshot, VIO, and ILA output screenshot for the designs.