Lab 4

Tasks to be done in this Lab:

1. Design and implement an overlapping sequence detector using Mealy FSM and verify its functionality using testbench.
2. Design a clock pulse generator while taking care of push-button debounce.
3. Test the functionality of the design on the Zybo Board (remote access) using VIO IP.

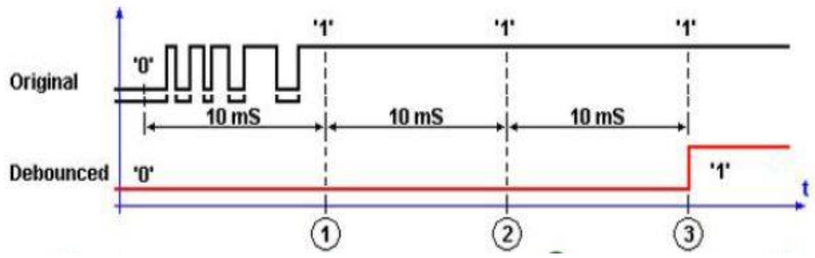
Topics to Explore: 1) Mealy and Moore FSM implementation, 2) Use of 3 Always blocks, 3) Difference between overlapping and non-overlapping, 4) Pushbutton debounce and clock pulse generation.

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.

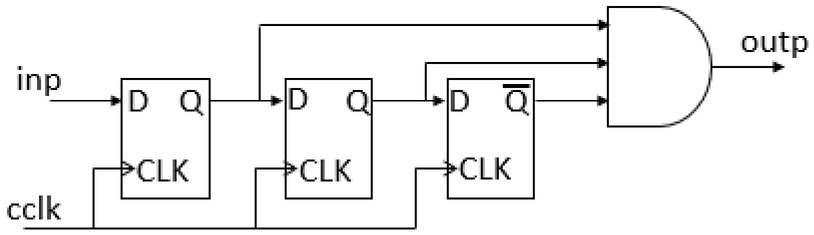
Note: Please revise the Mealy and Moore FSM and state diagrams as studied in Digital Circuits Course.

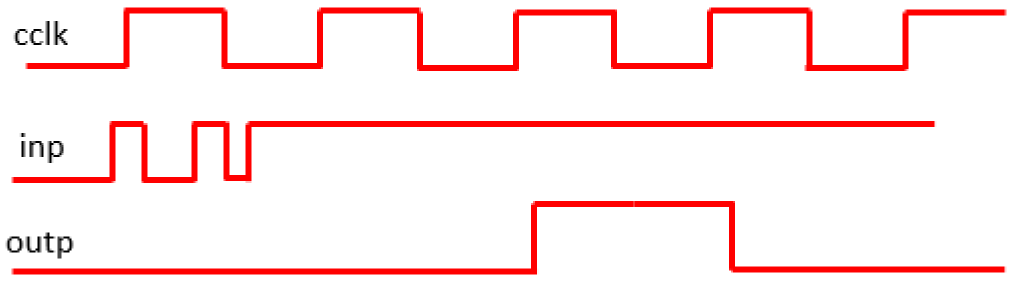
- **Design details:**
  - ❖ The input sequence will be given with the help of two push buttons.
  - ❖ One of the push buttons will correspond to an input of '1' and the other to an input of '0'.
  - ❖ Use the debouncing circuit to generate a clean pulse if any of the push buttons is pressed.
  - ❖ Use this pulse 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.

Before starting with the implementation, first, let's look into the **Push Button Debounce** as the inputs(clear,inp_0,inp_1) are taken from the push-buttons. When a push-button is pressed, it may debounce before setting its output high. In the case of "clear" input, it may not be an issue, but data inputs(inp_0 and inp_1) can be taken erroneously. A representation of the Debounce is shown in the below diagram.

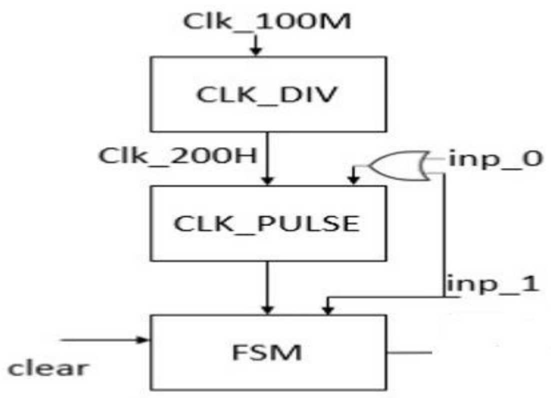


To avoid such an issue, we pass push-button inputs through debounce circuits. As shown below, we create a single pulse for every press of the push-button. As per the push-button requirement, cclk should be around 200 Hz.



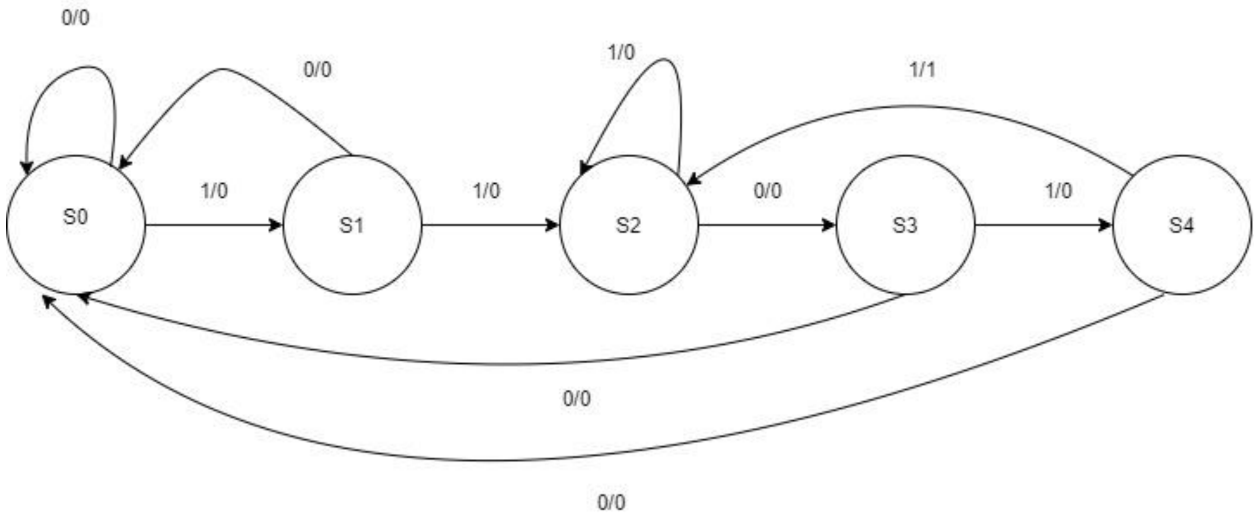


The next thing we need to consider is the clock pulse generation for the FSM. We will use a 200 Hz clock to generate a clock pulse, which has the transition from 0 to 1 whenever any one of the data push-buttons is pressed. The following block diagram will give you an abstract view of the implementation(all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



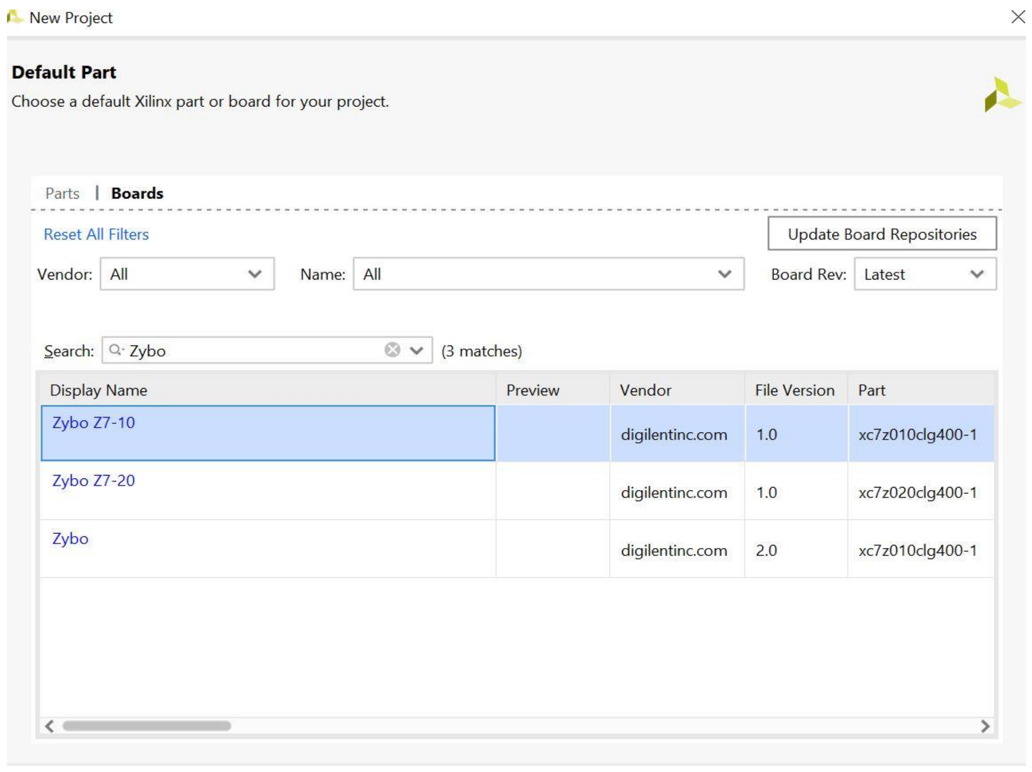
FSM Implementation

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



Step 1: Create the 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 Zybo Z7-10 as shown in the screenshot below


New Project

Default Part

Choose a default Xilinx part or board for your project.

Parts | **Boards**

Reset All Filters

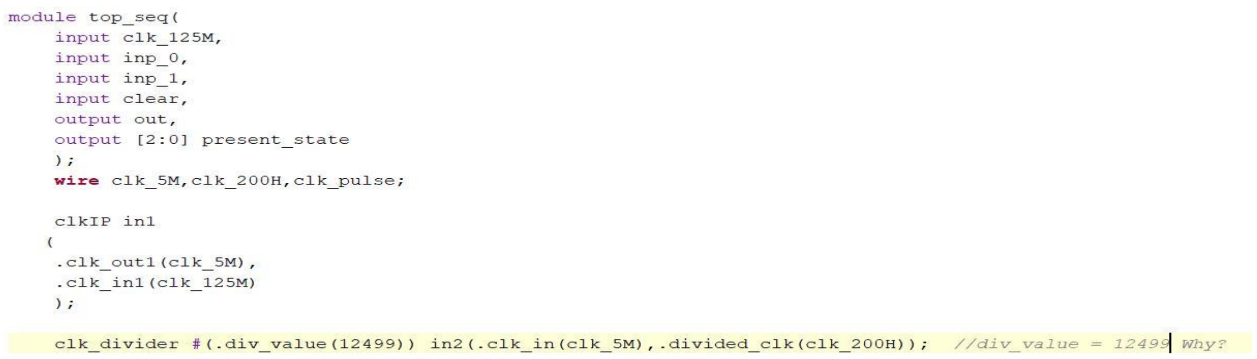
Update Board Repositories

Vendor: All Name: All Board Rev: Latest

Search: Zybo (3 matches)

Display Name	Preview	Vendor	File Version	Part
Zybo Z7-10		digilentinc.com	1.0	xc7z010clg400-1
Zybo Z7-20		digilentinc.com	1.0	xc7z020clg400-1
Zybo		digilentinc.com	2.0	xc7z010clg400-1

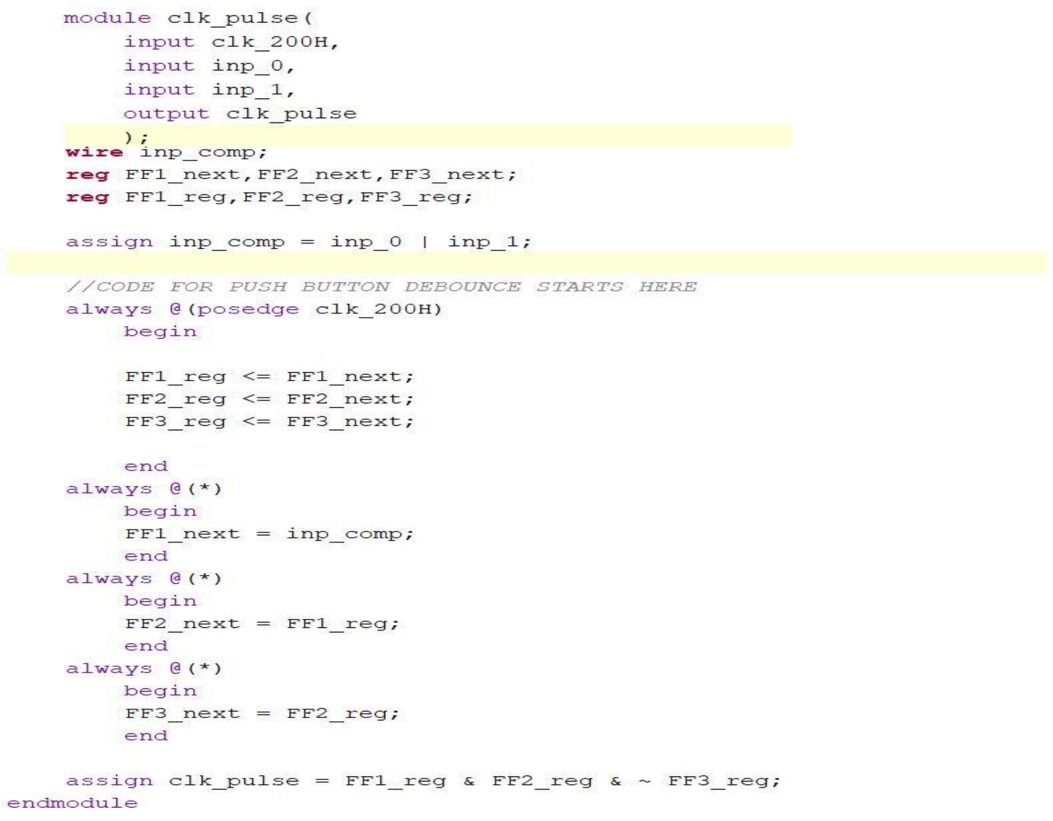
6. Make the port declaration, two input ports(clk_125M, clear,inp_0,inp_1) of 1 bit each, and one output port "out" of 1 bit and one output port present_state of 3 bits.
7. Add clocking IP and instantiate it.
8. Add the clock divider module we made in Lab 2. While adding design sources, you can click of 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 like the one shown below.



```
module top_seq(
    input clk_125M,
    input inp_0,
    input inp_1,
    input clear,
    output out,
    output [2:0] present_state
    );
    wire clk_5M,clk_200H,clk_pulse;

    clkIP in1
   (
    .clk_out1(clk_5M),
    .clk_in1(clk_125M)
    );

    clk_divider #(.div_value(12499)) in2(.clk_in(clk_5M),.divided_clk(clk_200H));  //div_value = 12499 Why?
```

Step 2: Add a new design source to define the functionality of the clock pulse generator.

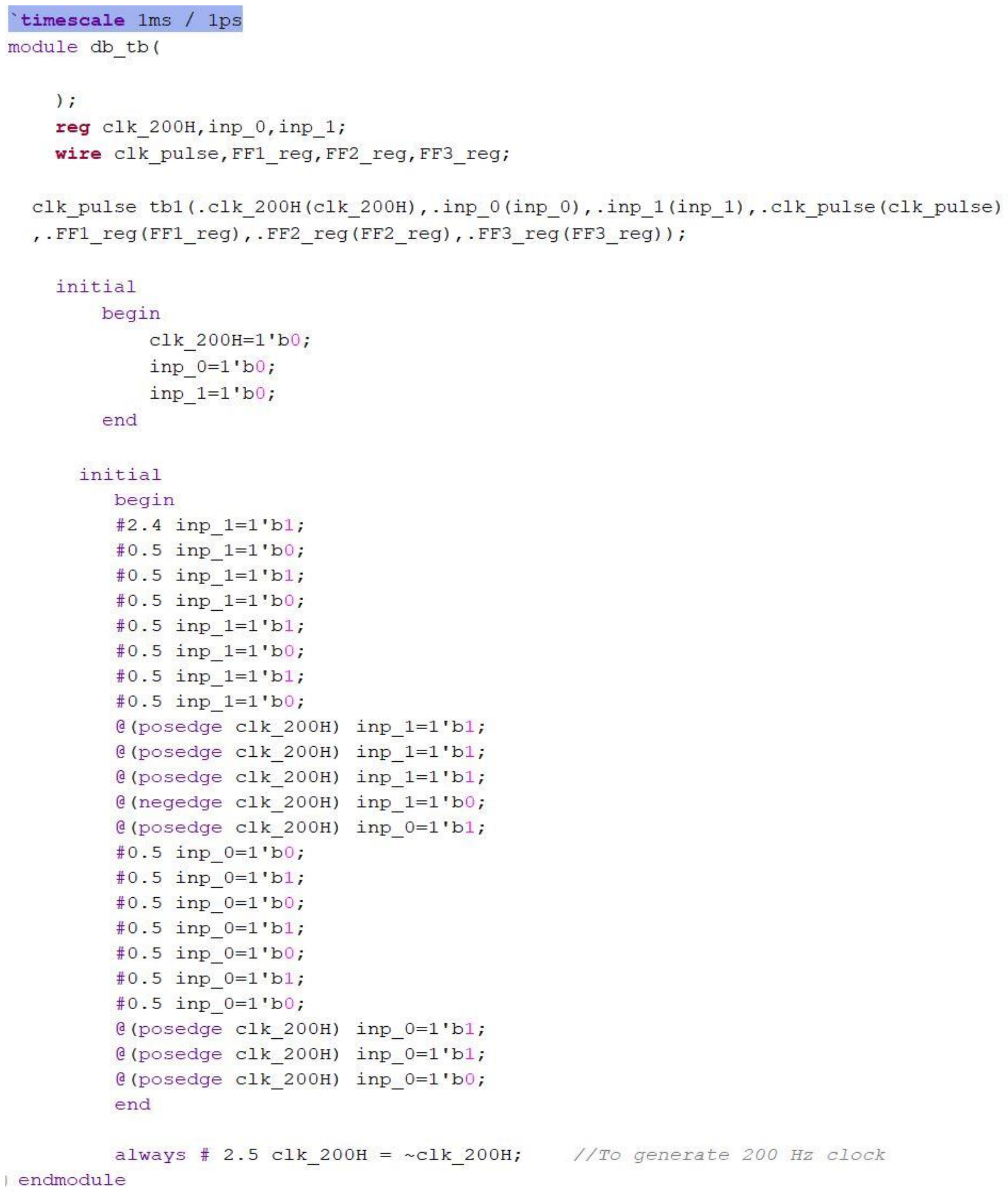
1. Now we will add the functionality of the clock pulse generator. To do so, add design sources and create one more Verilog file clk_pulse.v. Add the three input ports (clk_200H, inp_0,inp_1) of 1 bit each and one output port clk_pulse of 1 bit.



```
    module clk_pulse(
        input clk_200H,
        input inp_0,
        input inp_1,
        output clk_pulse
        );
    wire inp_comp;
    reg FF1_next,FF2_next,FF3_next;
    reg FF1_reg,FF2_reg,FF3_reg;

    assign inp_comp = inp_0 | inp_1;

    //CODE FOR PUSH BUTTON DEBOUNCE STARTS HERE
    always @(posedge clk_200H)
        begin

        FF1_reg <= FF1_next;
        FF2_reg <= FF2_next;
        FF3_reg <= FF3_next;

        end
    always @(*)
        begin
        FF1_next = inp_comp;
        end
    always @(*)
        begin
        FF2_next = FF1_reg;
        end
    always @(*)
        begin
        FF3_next = FF2_reg;
        end

    assign clk_pulse = FF1_reg & FF2_reg & ~ FF3_reg;
endmodule
```

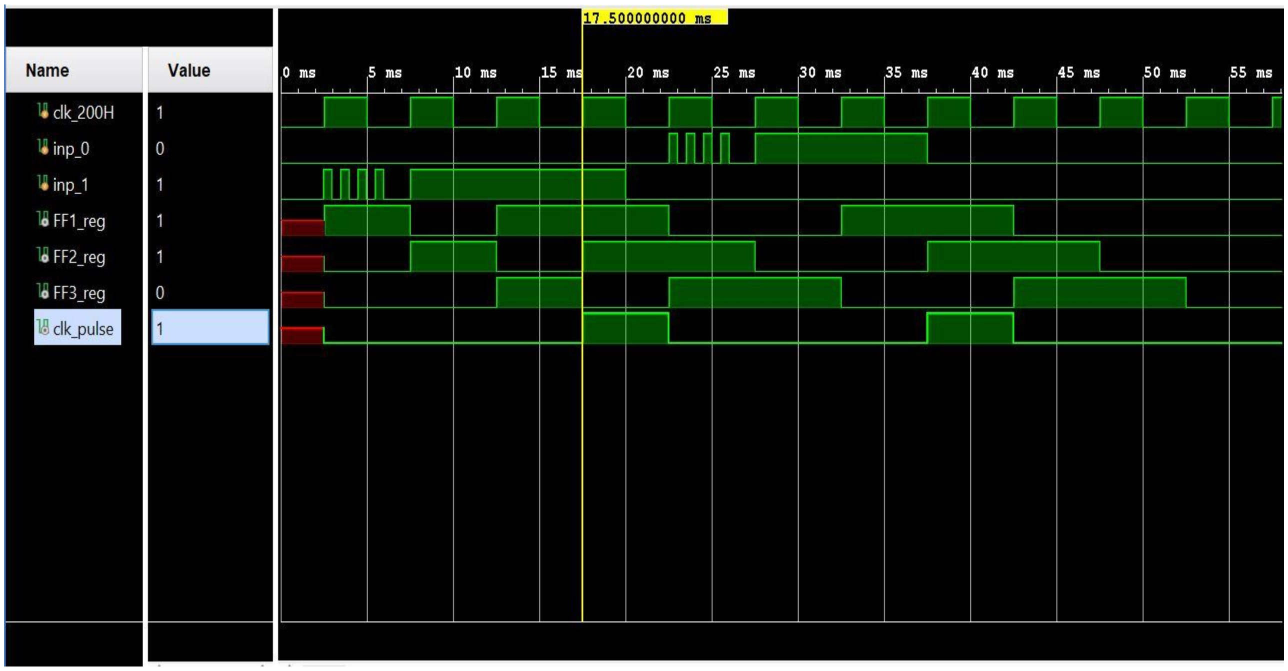
Before moving on to the next step, first, let's check the functionality of our clock pulse generator and see how the debouncing circuit is working. To see that more clearly, make FF1_reg, FF2_reg, and FF3_reg as the output ports (no need to do that in the main design. We are doing it only for getting a better insight how the clock pulse generator and debounce circuitry is working).

Make a testbench with the following code:



```
`timescale 1ms / 1ps
module db_tb(

    );
    reg clk_200H,inp_0,inp_1;
    wire clk_pulse,FF1_reg,FF2_reg,FF3_reg;

  clk_pulse tb1(.clk_200H(clk_200H),.inp_0(inp_0),.inp_1(inp_1),.clk_pulse(clk_pulse)
  ,.FF1_reg(FF1_reg),.FF2_reg(FF2_reg),.FF3_reg(FF3_reg));

    initial
        begin
            clk_200H=1'b0;
            inp_0=1'b0;
            inp_1=1'b0;
        end

      initial
         begin
         #2.4 inp_1=1'b1;
         #0.5 inp_1=1'b0;
         #0.5 inp_1=1'b1;
         #0.5 inp_1=1'b0;
         #0.5 inp_1=1'b1;
         #0.5 inp_1=1'b0;
         #0.5 inp_1=1'b1;
         #0.5 inp_1=1'b0;
         @(posedge clk_200H) inp_1=1'b1;
         @(posedge clk_200H) inp_1=1'b1;
         @(posedge clk_200H) inp_1=1'b1;
         @(negedge clk_200H) inp_1=1'b0;
         @(posedge clk_200H) inp_0=1'b1;
         #0.5 inp_0=1'b0;
         #0.5 inp_0=1'b1;
         #0.5 inp_0=1'b0;
         #0.5 inp_0=1'b1;
         #0.5 inp_0=1'b0;
         #0.5 inp_0=1'b1;
         #0.5 inp_0=1'b0;
         @(posedge clk_200H) inp_0=1'b1;
         @(posedge clk_200H) inp_0=1'b1;
         @(posedge clk_200H) inp_0=1'b0;
         end

         always # 2.5 clk_200H = ~clk_200H;     //To generate 200 Hz clock
 endmodule
```

Note: Please note that the timescale is changed to milliseconds



Name	Value
clk_200H	1
inp_0	0
inp_1	1
FF1_reg	1
FF2_reg	1
FF3_reg	0
clk_pulse	1

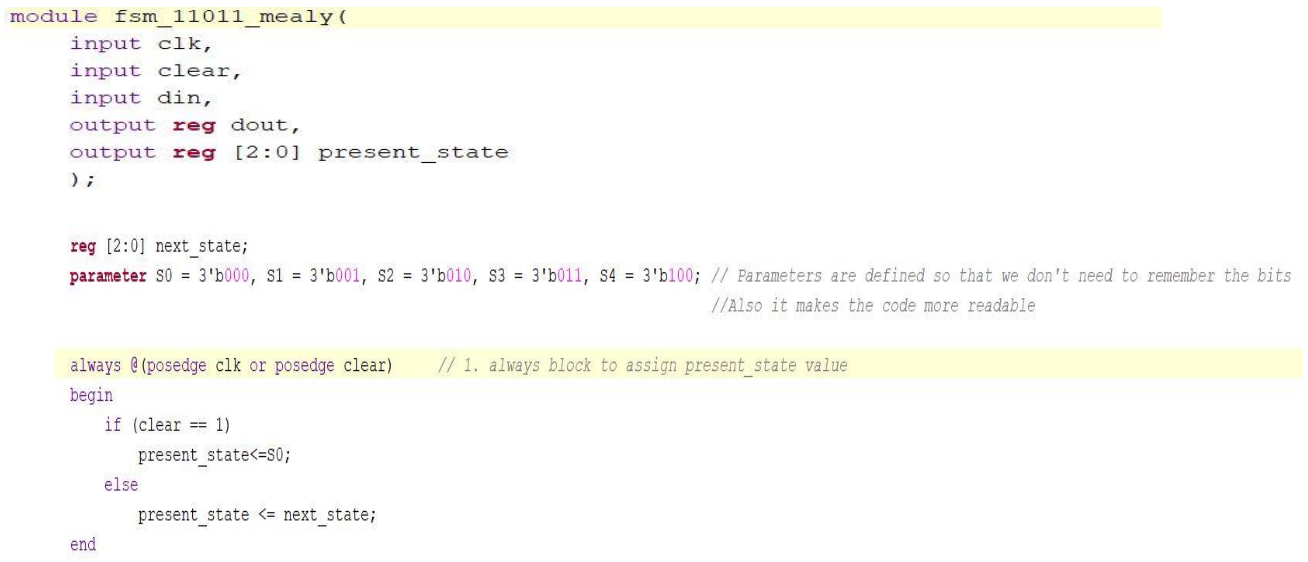
Look at the above waveform and match the results with the code. We have mimicked the debounce effect for both inp_0 and inp_1.

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

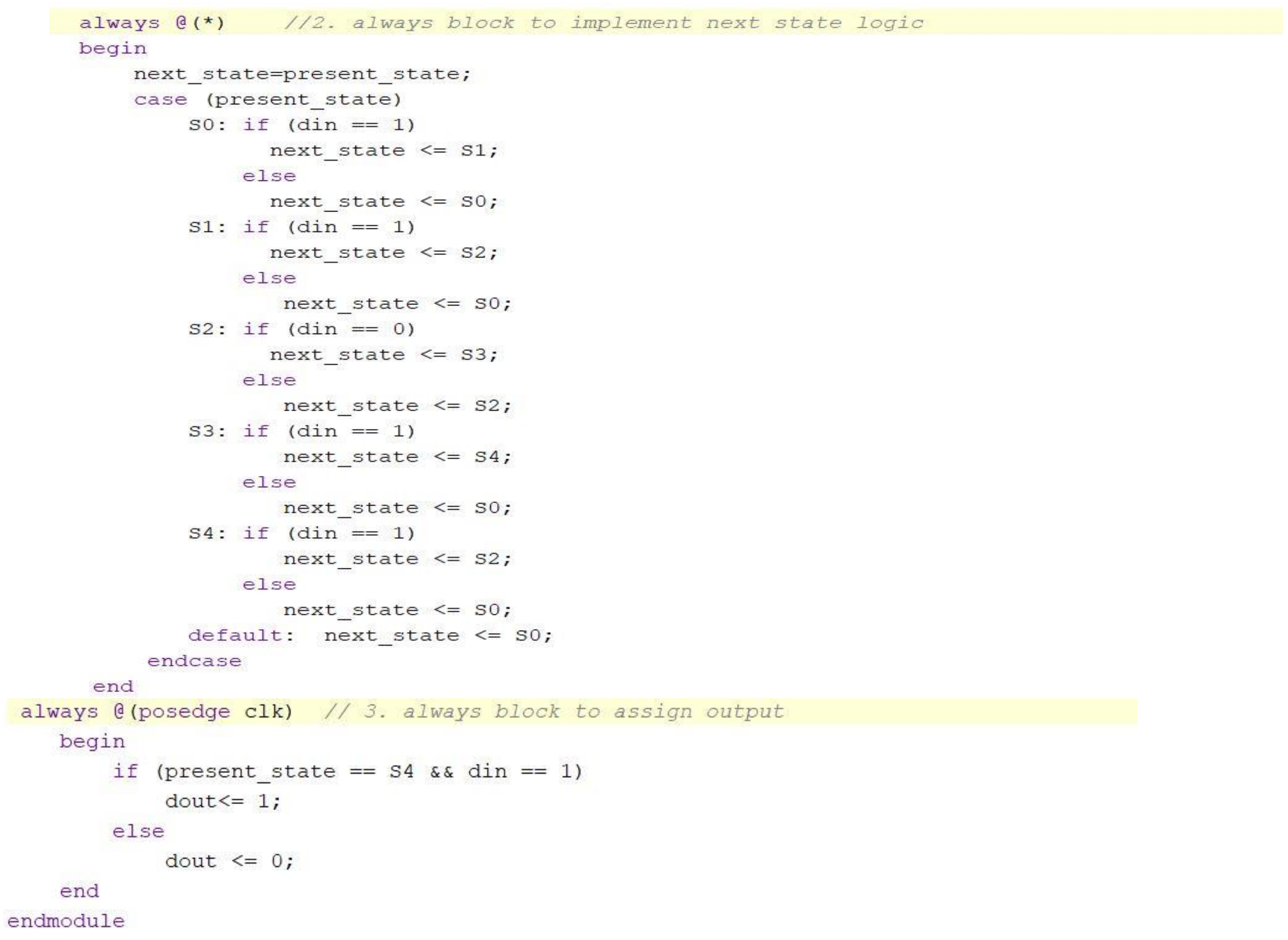
An FSM implementation in Verilog must be done in the way described in the following text. An FSM module will contain three always block.

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.

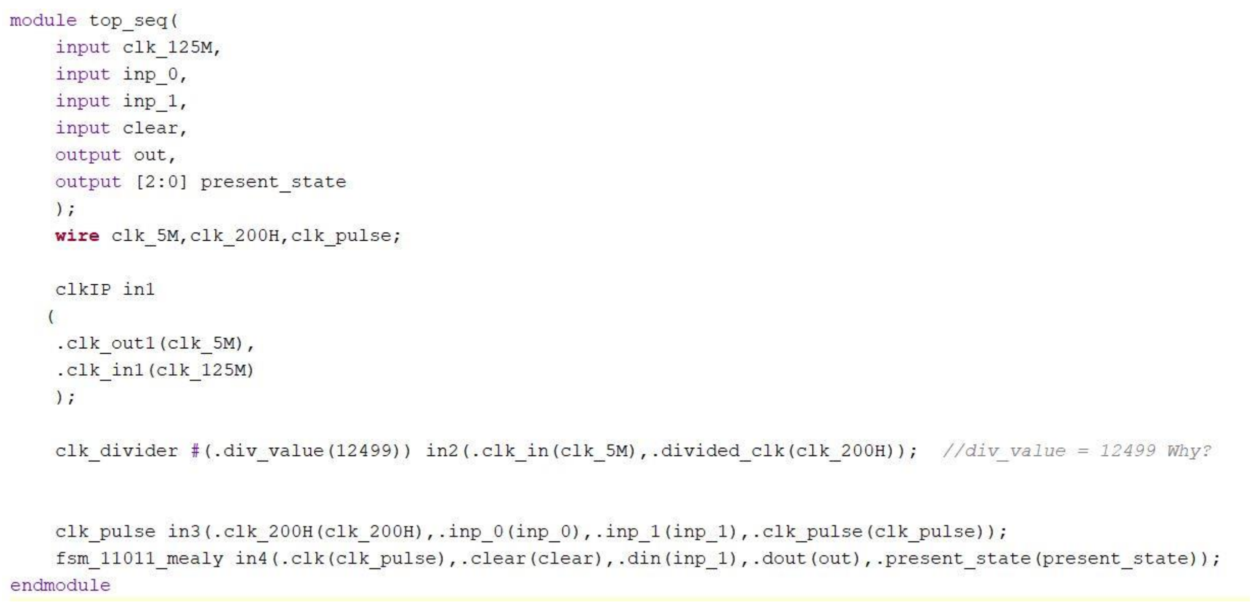


```
module fsm_11011_mealy(
    input clk,
    input clear,
    input din,
    output reg dout,
    output reg [2:0] present_state
    );

    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 clk or posedge clear)     // 1. always block to assign present_state value
    begin
        if (clear == 1)
            present_state<=S0;
        else
            present_state <= next_state;
    end
```



```
      always @(*)     //2. always block to implement next state logic
      begin
          next_state=present_state;
          case (present_state)
              S0: if (din == 1)
                    next_state <= S1;
                  else
                    next_state <= S0;
              S1: if (din == 1)
                    next_state <= S2;
                  else
                     next_state <= S0;
              S2: if (din == 0)
                    next_state <= S3;
                  else
                     next_state <= S2;
              S3: if (din == 1)
                     next_state <= S4;
                  else
                     next_state <= S0;
              S4: if (din == 1)
                     next_state <= S2;
                  else
                     next_state <= S0;
              default:  next_state <= S0;
           endcase
       end
  always @(posedge clk)  // 3. always block to assign output
     begin
         if (present_state == S4 && din == 1)
             dout<= 1;
         else
             dout <= 0;
     end
  endmodule
```

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



```
module top_seq(
    input clk_125M,
    input inp_0,
    input inp_1,
    input clear,
    output out,
    output [2:0] present_state
    );
    wire clk_5M,clk_200H,clk_pulse;

    clkIP in1
   (
    .clk_out1(clk_5M),
    .clk_in1(clk_125M)
    );

    clk_divider #(.div_value(12499)) in2(.clk_in(clk_5M),.divided_clk(clk_200H));  //div_value = 12499 Why?


    clk_pulse in3(.clk_200H(clk_200H),.inp_0(inp_0),.inp_1(inp_1),.clk_pulse(clk_pulse));
    fsm_11011_mealy in4(.clk(clk_pulse),.clear(clear),.din(inp_1),.dout(out),.present_state(present_state));
endmodule
```

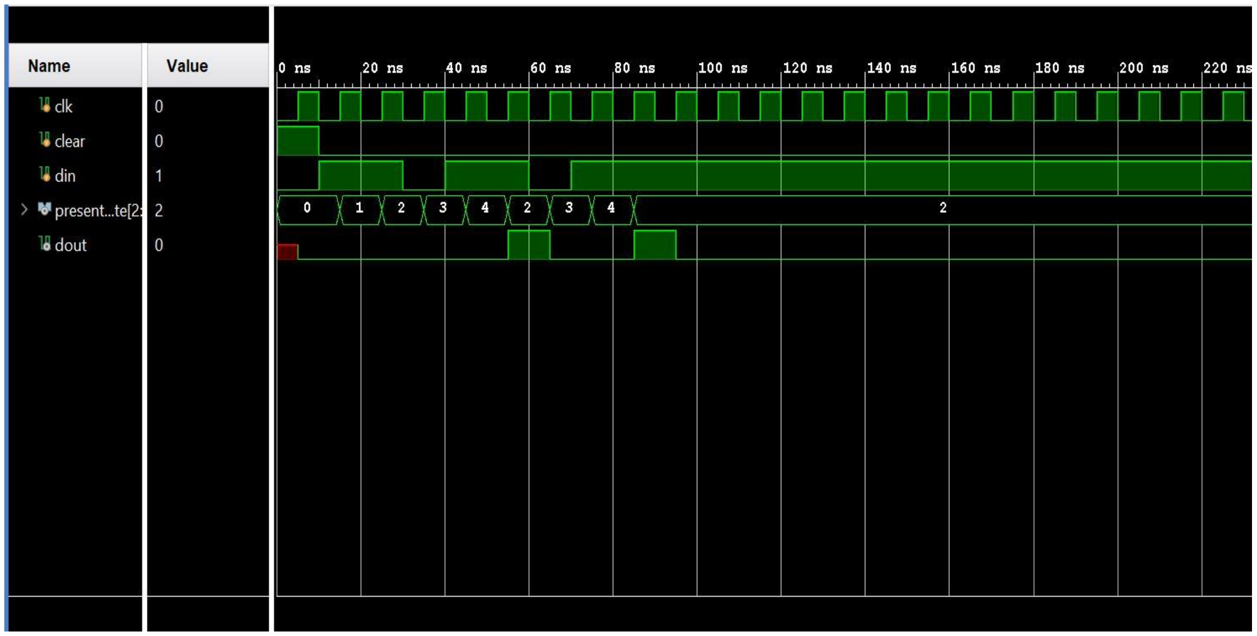
Step 4: Test the functionality of the Sequence Detector using testbench.

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

```
module seq_tb(

    );
    reg clk,clear,din;
    wire [2:0] present_state;
    wire dout;

    fsm_11011_mealy tb1(.clk(clk),.clear(clear),.din(din),.dout(dout),.present_state(present_state));

    initial
        begin
            clk<=1'b0;
            clear<=1'b1;
            din<=1'b0;
        end

     initial
        begin
        #10 clear=1'b0;

        @(negedge clk) din <= 1;  // Changing the input data at negative edge of the clock
        @(negedge clk) din <= 1;
        @(negedge clk) din <= 0;
        @(negedge clk) din <= 1;
        @(negedge clk) din <= 1;    // Pattern is completed
        @(negedge clk) din <= 0;
        @(negedge clk) din <= 1;
        @(negedge clk) din <= 1;   // Pattern is completed again
        end

        always #5 clk = ~clk;
endmodule
```

Note: We have instantiated the fsm_11011_mealy module here and not the top_seq module. The reason being we only need to check the functionality of the Sequence detector, and we can give the input sequence without any push-buttons, so no need to pass it through the debounce circuitry.

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

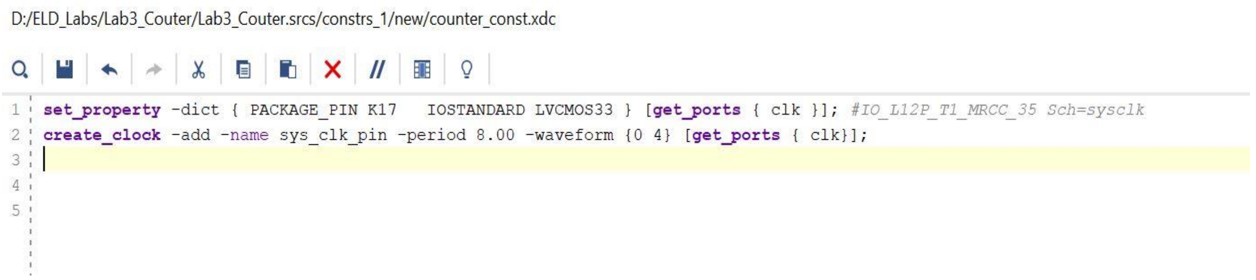


Step 5: Add VIO IP to your design to test it on hardware.

1. Create a vio_wrapper in the project in a similar way as we did in Lab 2 and Lab 3. The VIO IP will contain two input probes to monitor out, and present_state give and 2 output probes to provide input to clear, inp_0 and inp_1. The vio_wrapper module will look like, as shown below:

```
module vio_wrapper(
    input clk
    );
    wire inp_0,inp_1,clear,out;
    wire [2:0] present_state;


vio_0 vin1 (
  .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(inp_0),  // output wire [0 : 0] probe_out0
  .probe_out1(inp_1),  // output wire [0 : 0] probe_out1
  .probe_out2(clear)  // output wire [0 : 0] probe_out2
);

top_seq vin2(.clk_125M(clk),.clear(clear),.inp_0(inp_0),.inp_1(inp_1),.out(out),.present_state(present_state));
endmodule
```

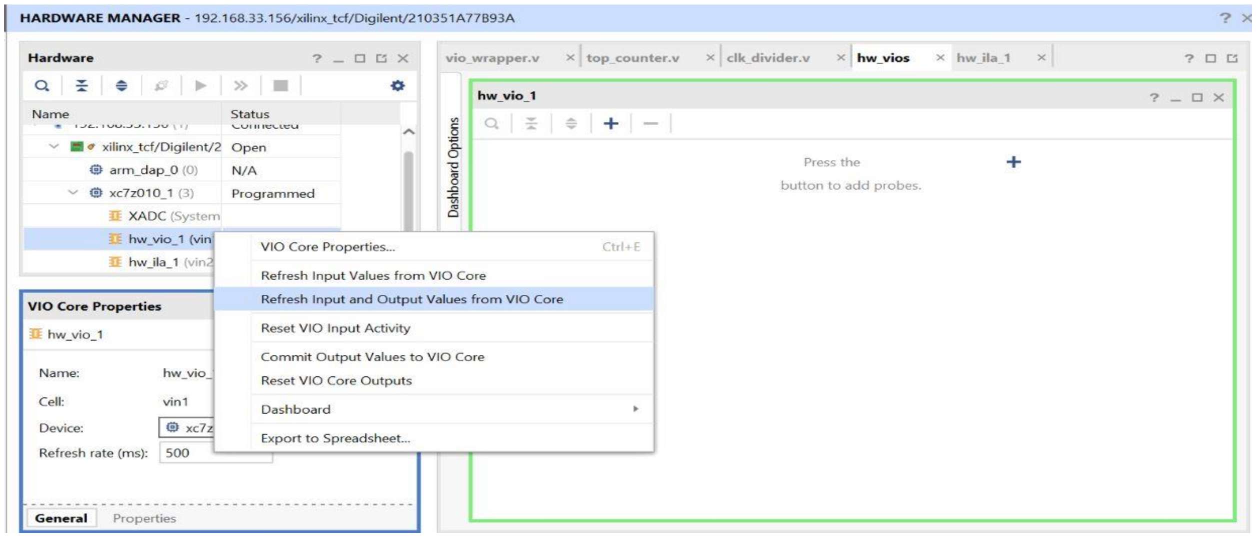
2. Add the constraints file.



D:/ELD_Labs/Lab3_Couter/Lab3_Couter.srcs/constrs_1/new/counter_const.xdc

```
set_property -dict { PACKAGE_PIN K17   IOSTANDARD LVCMOS33 } [get_ports { clk }]; #IO_L12P_T1_MRCC_35 Sch=sysclk
create_clock -add -name sys_clk_pin -period 8.00 -waveform {0 4} [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.



HARDWARE MANAGER - 192.168.33.156/xilinx_tcf/Digilent/210351A77B93A

Name	Status
192.168.33.156 (1)	Connected
xilinx_tcf/Digilent/2	Open
arm_dap_0 (0)	N/A
xc7z010_1 (3)	Programmed
XADC (System	
hw_vio_1 (vin	
hw_ila_1 (vin2	

VIO Core Properties

hw_vio_1

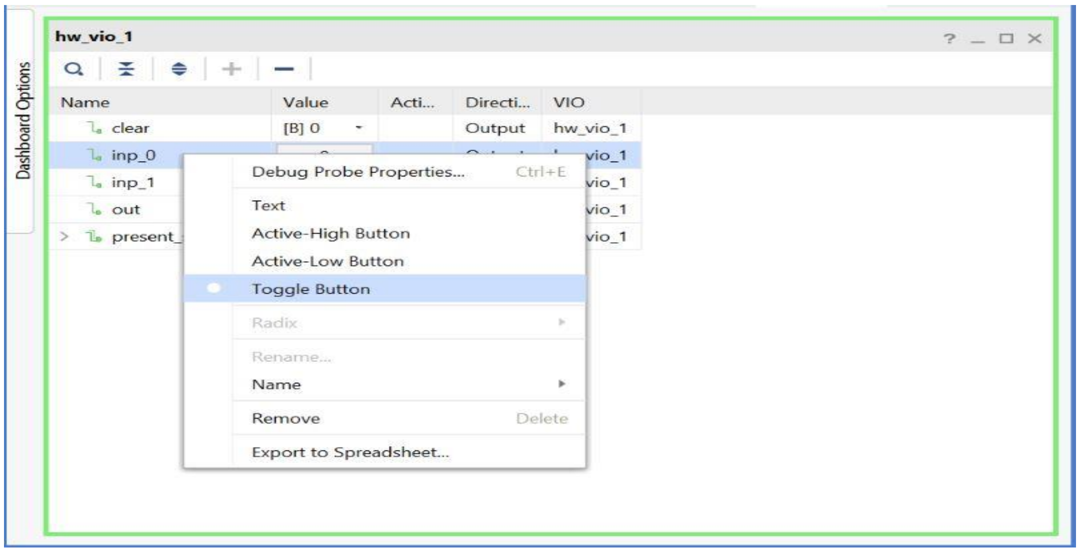
Name: hw_vio_

Cell: vin1

Device: xc7z

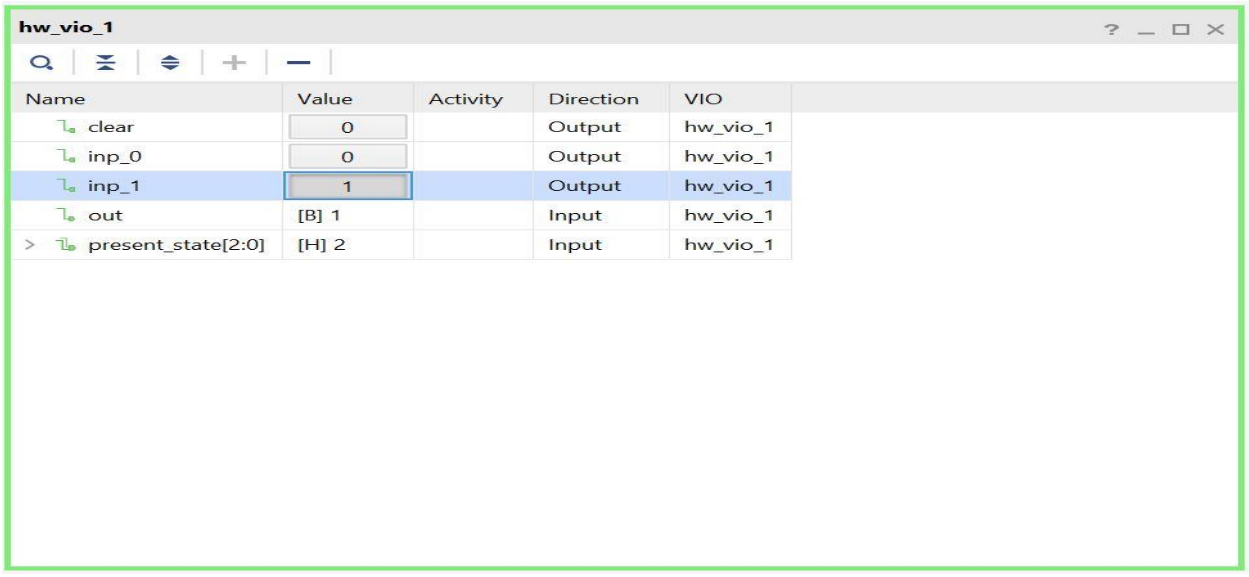
Refresh rate (ms): 500

6. Add the VIO probes and change the mode of input ports to the toggle button.



Name	Value	Acti...	Directi...	VIO
clear	[B] 0		Output	hw_vio_1
inp_0				vio_1
inp_1				vio_1
out				vio_1
present_				vio_1

7. 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.



Name	Value	Activity	Direction	VIO
clear	0		Output	hw_vio_1
inp_0	0		Output	hw_vio_1
inp_1	1		Output	hw_vio_1
out	[B] 1		Input	hw_vio_1
present_state[2:0]	[H] 2		Input	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 for the designs.