Lab 5

Tasks to be done in this Lab:

1. Design and implement a non-overlapping sequence detector using Moore FSM and verify its functionality using testbench.
2. Design an 8-bit adder with and without pipelining and observe the effect on the maximum operating frequency of the design.

ERRATA

A couple of points needs to be noted for the future labs and must be corrected in the clock divider module used in previous labs:

1. _next variable should not be initialized with 0. It is not necessary to initialize the output of the comb ckt. Only _reg variables should be initialized.
2. _next variable should not be used in the if statement comparison. We should use only the _reg variable in the if statement as it is stable and changes on at the clock-edge.

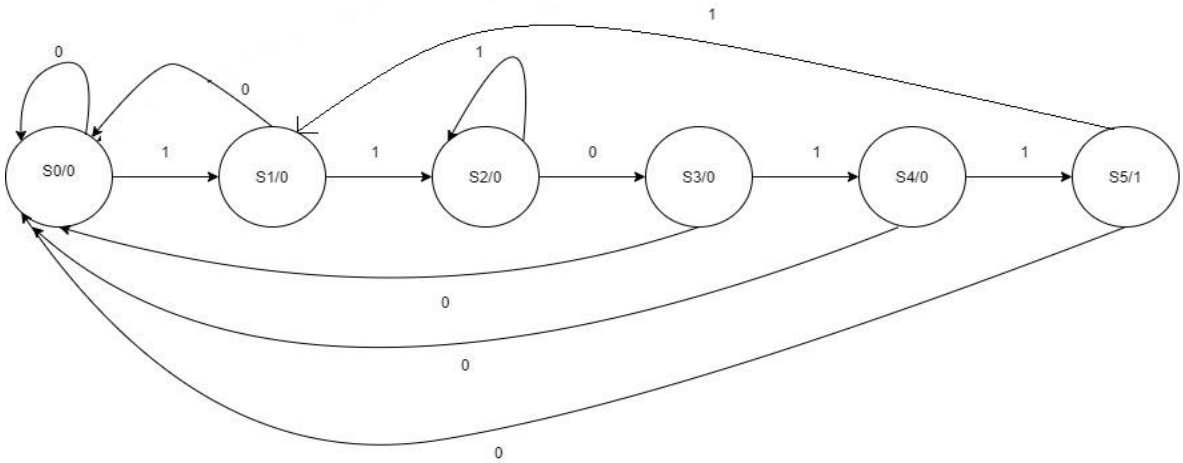
Part -1

This part is the continuation of the FSM designing using Verilog done in Lab 4. The design details remain the same (please refer to Lab 4 handout for the design details). In this Lab, we will design a non-overlapping sequence detector (11011) using Moore type FSM.

All the modules designed in Lab 4 can be re-used here except for the FSM module. As the state diagram for the non-overlapping sequence detector will be different, and Moore's FSM implementation is slightly different (for the output assignment), we will be designing a new module for the FSM.

FSM Implementation

The state diagram for the Moore type sequence detector(11011) without overlap is shown below:



The modified code for the FSM module is given below:

```
module fsm_11011_moore(
    input clk,
    input clear,
    input din,
    output  reg dout=0,
    output reg [2:0] present_state=0
    );
    reg [2:0] next_state;
    parameter S0 = 3'b000, S1 = 3'b001, S2 = 3'b010, S3 = 3'b011, S4 = 3'b100, S5=3'b101;

    always @(posedge clk or posedge clear)   // 1. always block for present_state assignment (Same as in Mealy FSM)
    begin
        if (clear == 1)
            present_state<=S0;
        else
            present_state <= next_state;
    end


   always @(*)   // 2. always block for next_state logic implementation (New state added in Moore FSM)
   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 <= S5;
               else
                  next_state <= S0;
           S5: if (din == 1)
                  next_state <= S1;
               else
                  next_state <= S0;
           default:  next_state <= S0;
        endcase
   end

 always @(*)  // 3. Always block for output assignment(different from Mealy FSM)
 begin
    if(present_state == S5)
        dout=1'b1;
    else
        dout=1'b0;
 end

endmodule
```

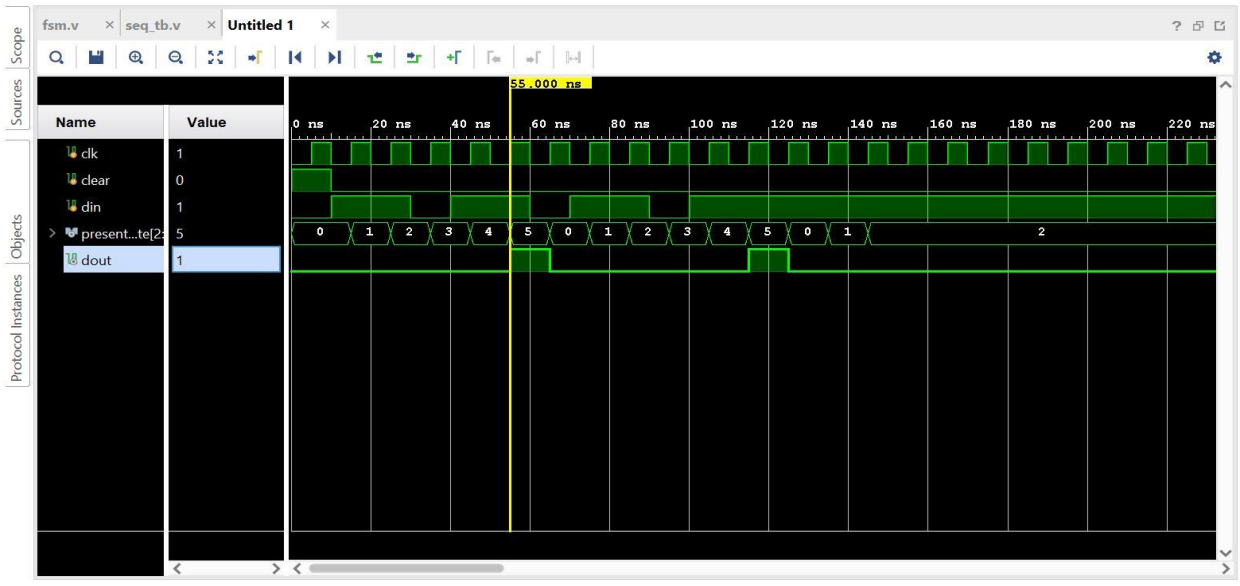
Step 4: Test the functionality of the Sequence Detector using testbench.

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

```
`timescale 1ns / 1ps
module seq_tb(

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

    fsm_11011_moore 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;
            @(negedge clk) din <= 1;
            @(negedge clk) din <= 0;
            @(negedge clk) din <= 1;        // Pattern is completed
            @(negedge clk) din <= 1;
            @(negedge clk) din <= 0;
            @(negedge clk) din <= 1;
            @(negedge clk) din <= 1;
            @(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_moore 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.

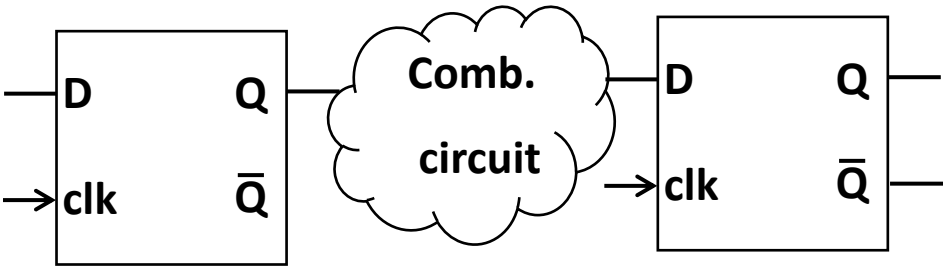


Part-2

Pre-Requisites: Go through the Pipelining Basics Video posted on YouTube

In this part, we will make two different designs of an 8-bit adder—one without pipelining and the other with pipelining and observe the effect on the maximum operating frequency and latency of the design.

Flip Flop Timing Constraints:



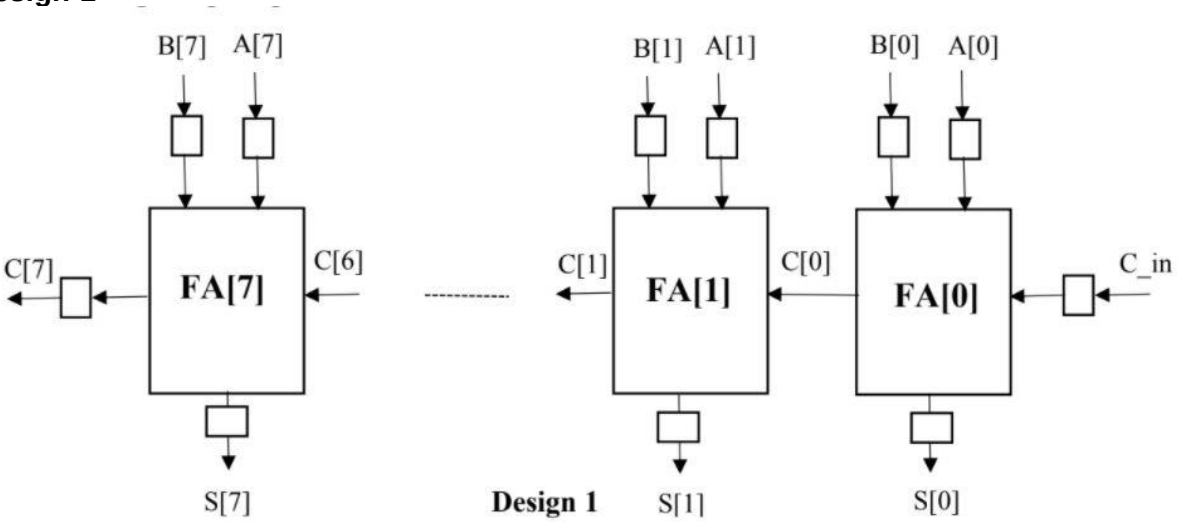
$$T_{clk} \geq T_{clk-out} + T_{pmax} + T_{setup} + T_{skew}$$

$$T_{clk-out} + T_{pmin} \geq T_{hold} + T_{skew}$$

- Setup Time→ The time period before the active clock edge during which input must not change.
- Hold Time→Time after the active clock edge (+ve edge for +ve edge triggered FF) during which the input must be stable.

Create a new project and add the full adder module we used in Lab 2. Add a new design source and create the top module of the design top_pipelining with three input ports, one clock port of single-bit, two ports of 8 bits each, and one output port for the sum of 9 bits.

Design-1



C[7] will be assigned to S[8].

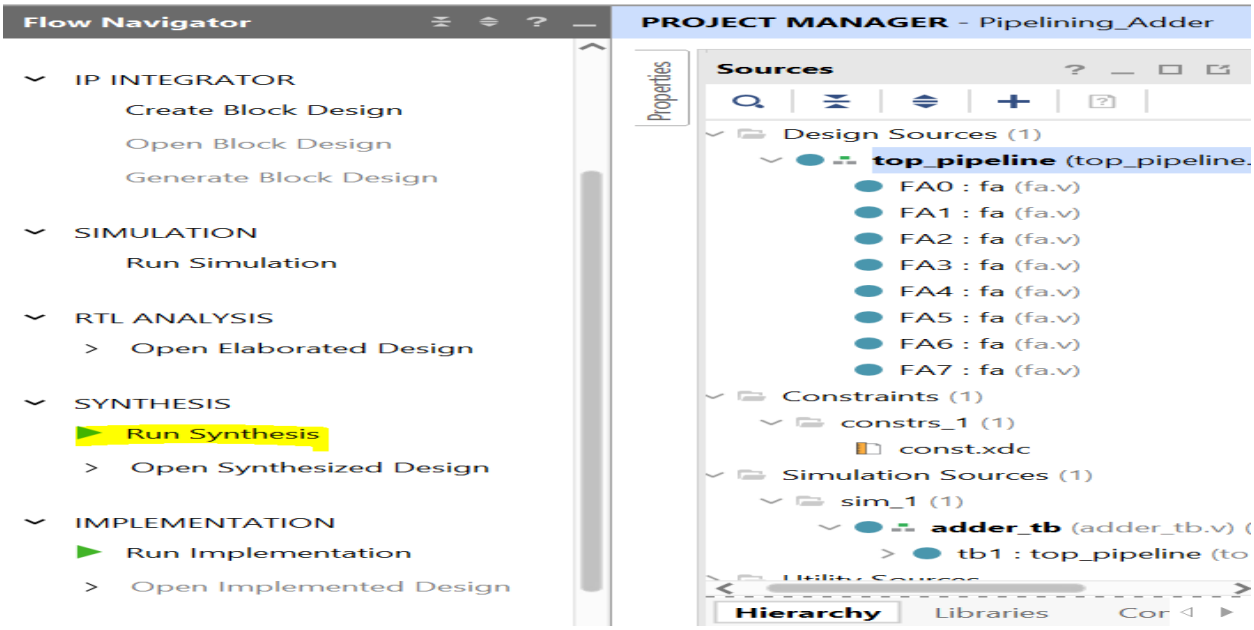
```
module top_pipeline(
    input clk,
    input [7:0] A,
    input [7:0] B,
    output reg [8:0] Sum
    );

    reg [7:0] FF_A,FF_B;
    wire [8:0] S;
    wire [6:0] C;

    always@(posedge clk)
    begin
        FF_A <= A;
        FF_B <= B;
        Sum  <= S;
    end

    fa FA0(.A(FF_A[0]), .B(FF_B[0]), .C(1'b0), .Sum(S[0]), .Carry(C[0]));
    fa FA1(.A(FF_A[1]), .B(FF_B[1]), .C(C[0]), .Sum(S[1]), .Carry(C[1]));
    fa FA2(.A(FF_A[2]), .B(FF_B[2]), .C(C[1]), .Sum(S[2]), .Carry(C[2]));
    fa FA3(.A(FF_A[3]), .B(FF_B[3]), .C(C[2]), .Sum(S[3]), .Carry(C[3]));
    fa FA4(.A(FF_A[4]), .B(FF_B[4]), .C(C[3]), .Sum(S[4]), .Carry(C[4]));
    fa FA5(.A(FF_A[5]), .B(FF_B[5]), .C(C[4]), .Sum(S[5]), .Carry(C[5]));
    fa FA6(.A(FF_A[6]), .B(FF_B[6]), .C(C[5]), .Sum(S[6]), .Carry(C[6]));
    fa FA7(.A(FF_A[7]), .B(FF_B[7]), .C(C[6]), .Sum(S[7]), .Carry(S[8]));

endmodule
```

The next step is to add constraints. For that, add a constraint file and add the following constraint:

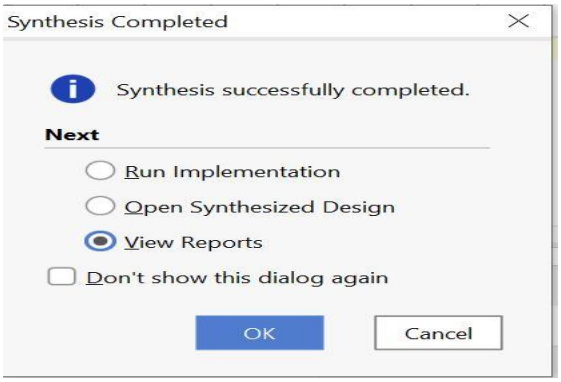
```
1  create_clock -period 2.5 [get_ports clk]
```

Note you can change the clock period by making changes in the constraint.

Synthesize the design using the run synthesis option shown below:

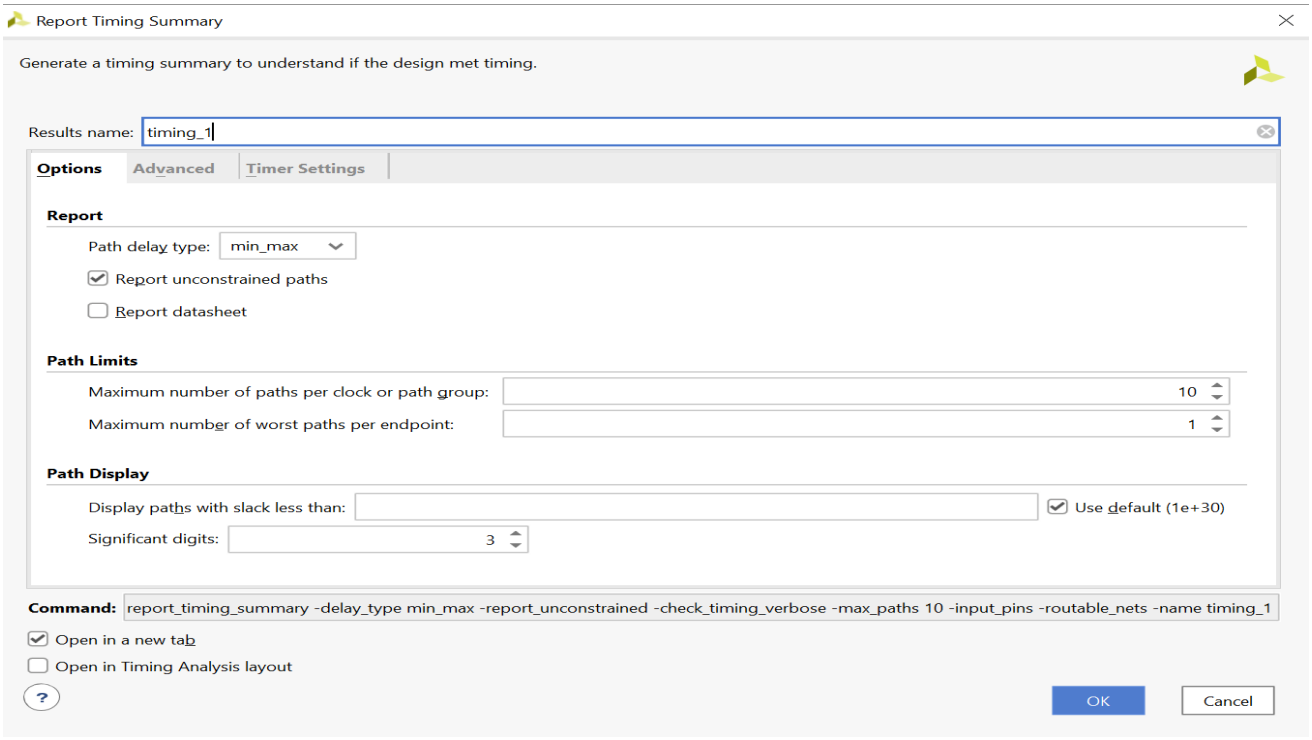


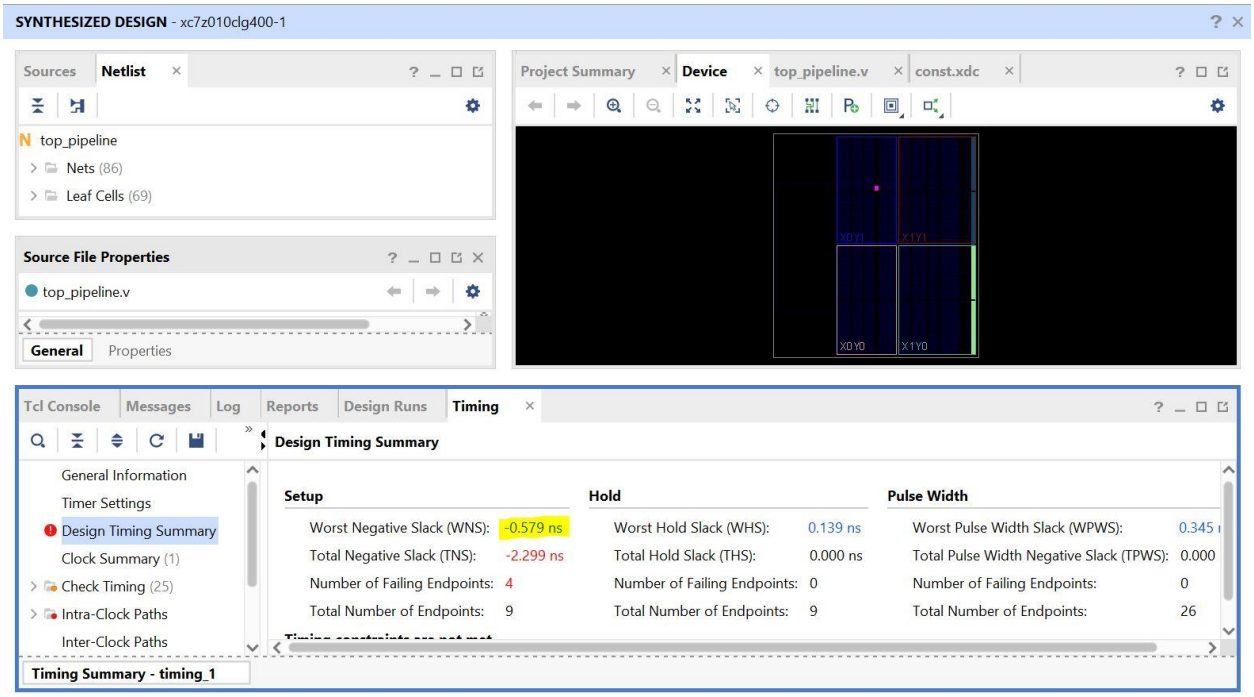
Once the synthesis is complete, you will get the following pop-up. Select view reports and click OK.



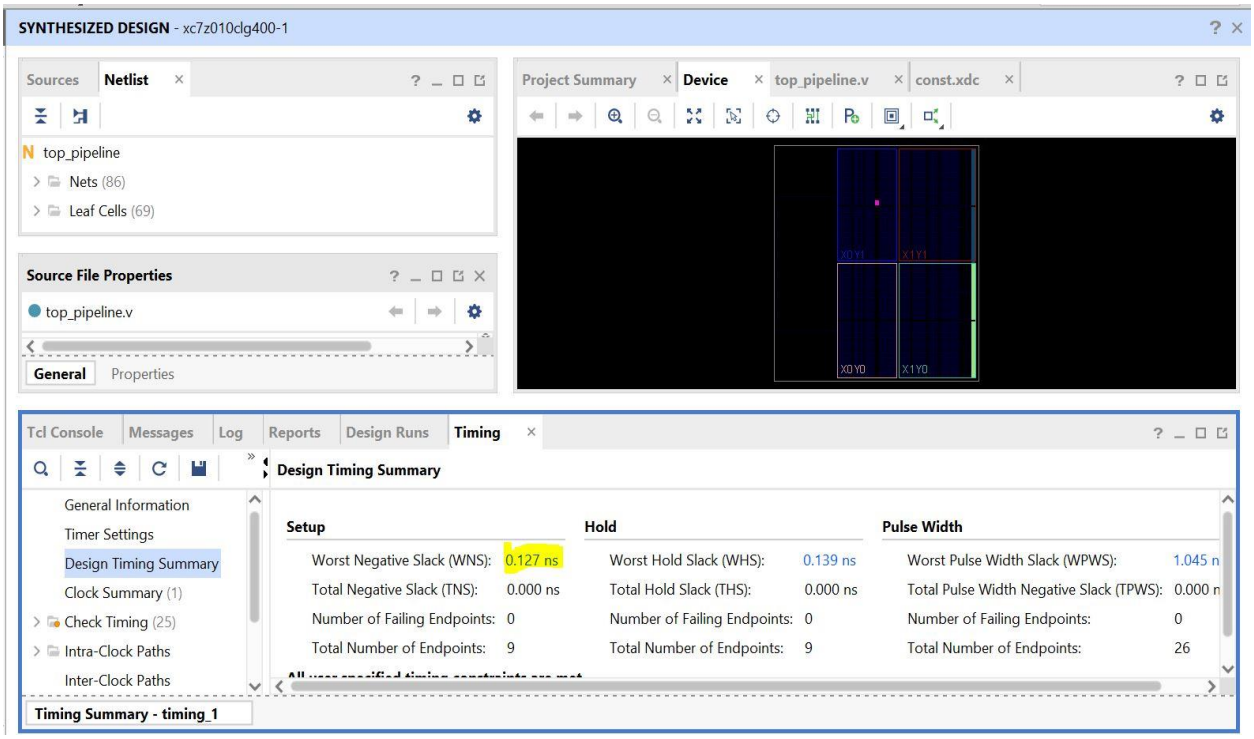
Next, click on open synthesized design and click on Report Timing Summary.

Click OK on the next pop-up window.





As you can observe in the Timing Summary given above, we are getting a negative slack for the Setup Checks. That means the clock period of 2.5ns is not sufficient for the design. The clock period needs to be increased by at least 0.579 ns. Change the clock period in constraints to 3.2ns and recheck the timing reports.

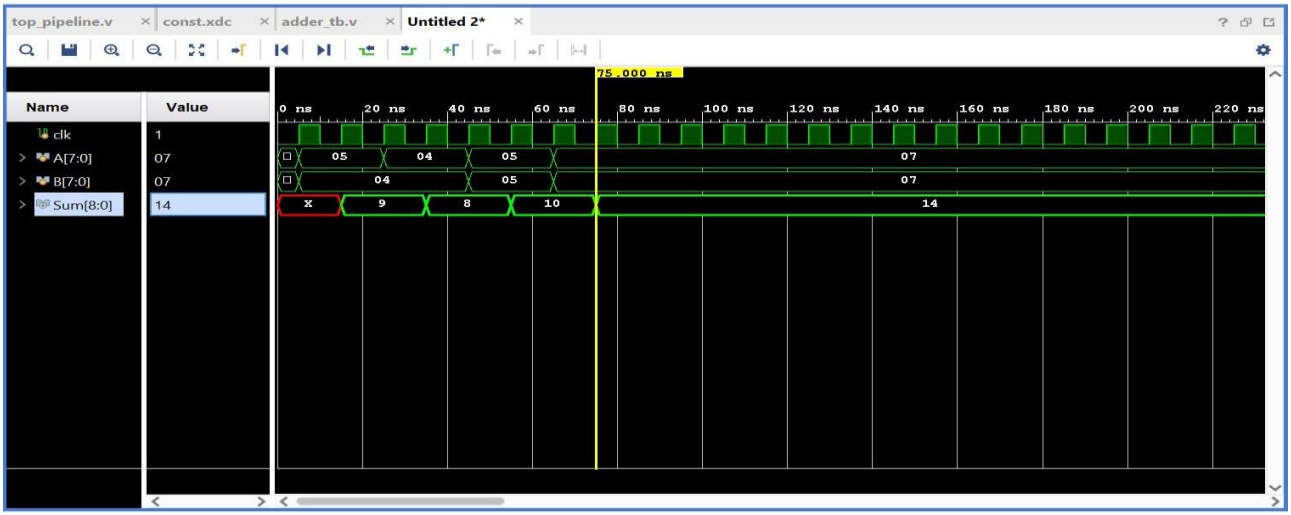


As you can see that both setup and hold slack are positive. That means timings requirements are met for the design with a setup slack of 0.127 ns.

I**nterpretation of positive and negative Slack:** Timing slack is a quantitative measure of whether we are able to meet the timing requirements of the design. At this point, it is necessary to define two more quantities, namely Arrival Time and Required Time. Arrival time is the time at which the processed data will we arriving at the D pin of capturing flip flop when the worst cases of flip-flop delay, data path delay, and setup time are considered. The required time is the clock period (subtract the skew if it is there to take the worst case). Ideally, the arrival time should be less than the required time to ensure that the correct data is captured at the capturing flip-flop. The difference of required time and arrival time is known as Slack. If Slack is positive, then we can ensure that the correct data will be captured or the synchronicity of the design is maintained. But if the Slack is negative, that means that the data is arriving after the arrival of the clock, and hence the incorrect data will be captured, or synchronicity is disturbed.

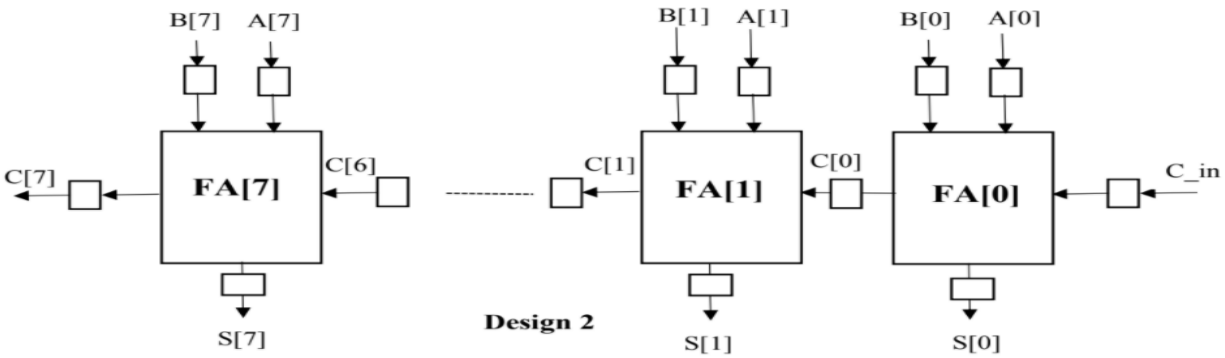
Check the functionality of the design using the testbench given below. You will observe that the output is coming after one clock cycle due to the insertion of flip flops at the input and output ports.

```
module adder_tb(

    );   // A Testbench have no port definition

    reg clk;
    reg [7:0] A,B;
    wire [8:0] Sum; //Step 1: Variable definittion

    top_pipeline tb1(.clk(clk),.A(A),.B(B),.Sum(Sum));  //Step 2: DUT Instantiation

    initial     //Step 3: Initialization of variables
        begin
            clk=1'b0;
            A=8'd0;
            B=8'd0;
        end

    initial     //Step 4 : Changing the Stimulus
        begin
            @(posedge clk) A=8'd5;B=8'd4;
            @(posedge clk) A=8'd5;B=8'd4;
            @(posedge clk) A=8'd4;B=8'd4;
            @(posedge clk) A=8'd4;B=8'd4;
            @(posedge clk) A=8'd5;B=8'd5;
            @(posedge clk) A=8'd5;B=8'd5;
            @(posedge clk) A=8'd7;B=8'd7;
        end

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



Design-2

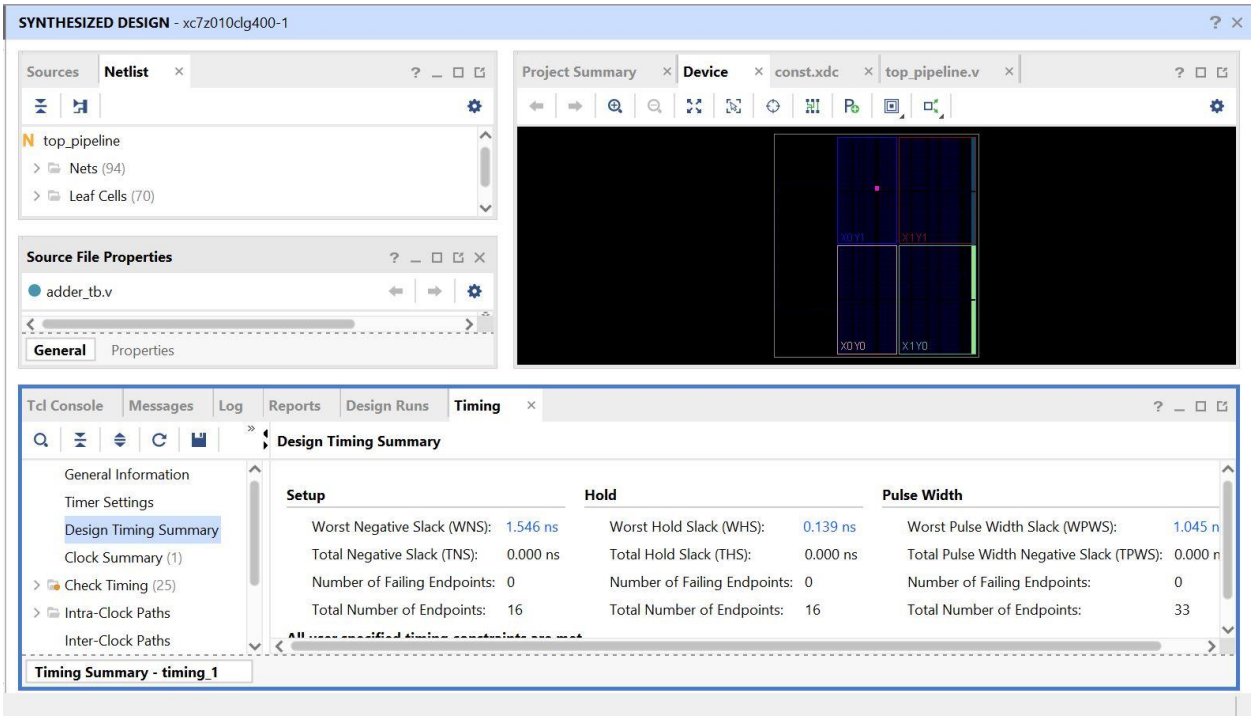
In design-2, add flip flops at the input carry of each full adder as well as shown below:



The code of design-2 is given below:

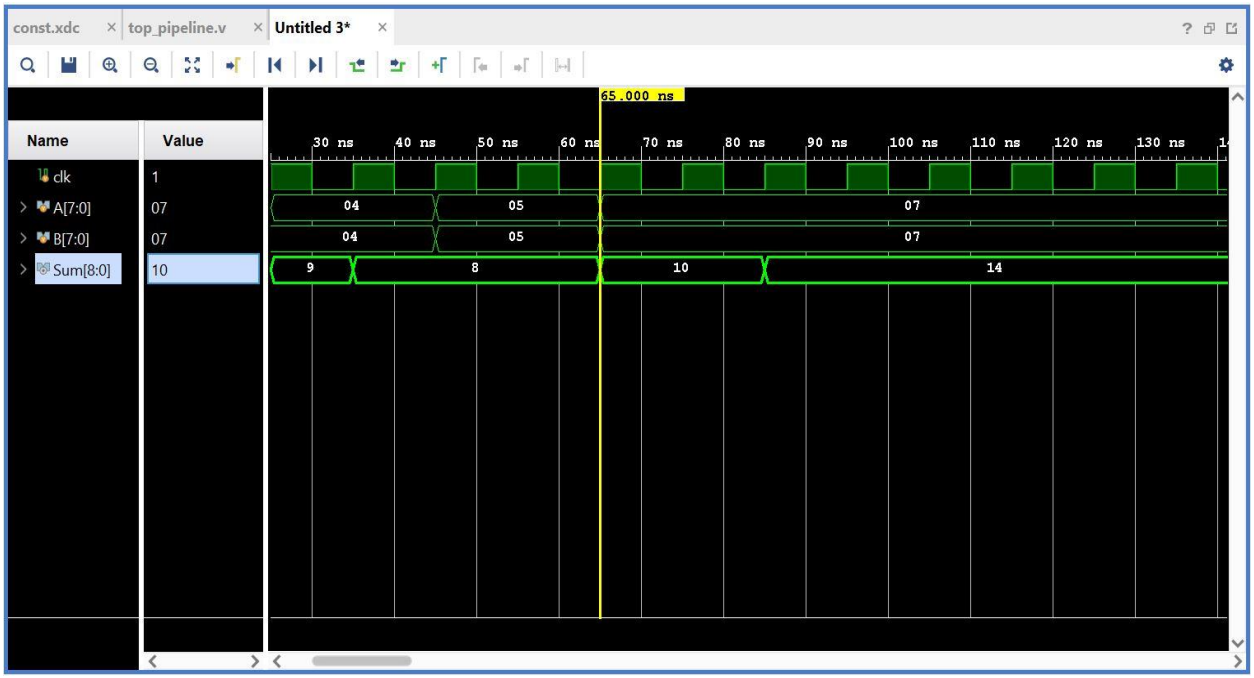
```
module top_pipeline(
    input clk,
    input [7:0] A,
    input [7:0] B,
    output reg [8:0] Sum
    );

    wire [8:0] S;
    wire [6:0] C;
    reg [7:0] FF_A,FF_B;
    reg [6:0] regC;


    always@(posedge clk)
    begin
        FF_A <= A;
        FF_B <= B;
        Sum  <= S;
        regC <= C;
    end

      fa FA0(.A(FF_A[0]), .B(FF_B[0]), .C(1'b0)    , .Sum(S[0]), .Carry(C[0]));
      fa FA1(.A(FF_A[1]), .B(FF_B[1]), .C(regC[0]), .Sum(S[1]), .Carry(C[1]));
      fa FA2(.A(FF_A[2]), .B(FF_B[2]), .C(regC[1]), .Sum(S[2]), .Carry(C[2]));
      fa FA3(.A(FF_A[3]), .B(FF_B[3]), .C(regC[2]), .Sum(S[3]), .Carry(C[3]));
      fa FA4(.A(FF_A[4]), .B(FF_B[4]), .C(regC[3]), .Sum(S[4]), .Carry(C[4]));
      fa FA5(.A(FF_A[5]), .B(FF_B[5]), .C(regC[4]), .Sum(S[5]), .Carry(C[5]));
      fa FA6(.A(FF_A[6]), .B(FF_B[6]), .C(regC[5]), .Sum(S[6]), .Carry(C[6]));
      fa FA7(.A(FF_A[7]), .B(FF_B[7]), .C(regC[6]), .Sum(S[7]), .Carry(S[8]));

endmodule
```

Synthesize this design with a 2.5 ns clock period and check the timing reports.



As we can clearly see, while design-1 had a negative slack at 2.5ns, design-2 is having a positive slack of 1.546ns. This is due to the insertion of flip-flops in the critical path; the maximum operating frequency is increased.

Now, if we look at the output of the waveform, we can see that the output is taking one additional clock cycle, and hence latency is increased.



Conclusion: Adding registers/flip-flops in the critical path reduces the critical path delay and hence the clock time period. As a result, the maximum clock frequency increases. But as a trade-off, the latency of the design increases.

Deliverables for Lab 5 Submission(Graded):

1. PDF with code, testbench, simulation screenshot for design-1.
2. Code, testbench, simulation screenshot, Timing Report screenshots for design-2 in the same PDF