FPGA Homework 2 Report

Parsa Haghighatgoo
Student ID : 40030644

December 27, 2025

1 Question 1

The task is to write Verilog code for the circuit shown in Figure 1. The circuit contains two combinational logic blocks feeding two D flip-flops that share a common clock:

- The upper D flip-flop receives $D_1 = a \wedge b$.
- The lower D flip-flop receives $D_2 = b \vee c$.
- Both flip-flops sample their inputs on the rising edge of `clk`.

1. Write the Verilog code for the circuit below.

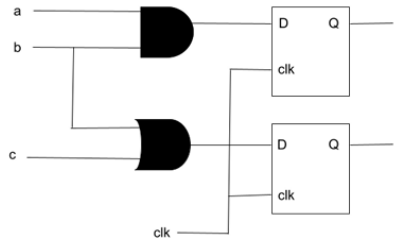


Figure 1

Figure 1: Given circuit: $q1 \leftarrow (a \wedge b)$ and $q2 \leftarrow (b \vee c)$ at each rising edge of `clk`.

2 Design Approach

The circuit is modeled using:

- Two internal wires for the combinational outputs:

$$d1 = a \wedge b, \quad d2 = b \vee c$$

- A sequential `always @(posedge clk)` block to represent the two D flip-flops.

On every rising edge of `clk`, the flip-flops capture the combinational values and update `q1` and `q2`.

3 Verilog Implementation (RTL)

Listing 1 shows the Verilog module that directly matches the logic in Figure 1.

```
module q1 (
    input  wire a,
    input  wire b,
    input  wire c,
    input  wire clk,
    output reg  q1,
    output reg  q2
);

wire d1 = a & b;
wire d2 = b | c;

always @(posedge clk) begin
    q1 <= d1;
    q2 <= d2;
end

endmodule
```

Listing 1: RTL Verilog module for Question 1.

4 Testbench

A testbench was created to verify sequential behavior. It generates a clock with a 10 ns period and applies different input combinations of (a, b, c) over time. The outputs are observed in simulation to confirm that they only update on clock edges.

```
`timescale 1ns/1ps

module q1_tb;

reg a, b, c;
reg clk;
wire q1, q2;

q1 uut (
    .a(a),
    .b(b),
    .c(c),
    .clk(clk),
    .q1(q1),
    .q2(q2)
```

```
);

// Clock generation (10 ns period)
always #5 clk = ~clk;

initial begin
    // Initialize
    clk = 0;
    a = 0; b = 0; c = 0;

    // Apply test vectors
    #10 a=0; b=0; c=0;
    #10 a=0; b=0; c=1;
    #10 a=0; b=1; c=0;
    #10 a=0; b=1; c=1;
    #10 a=1; b=0; c=0;
    #10 a=1; b=0; c=1;
    #10 a=1; b=1; c=0;
    #10 a=1; b=1; c=1;

    #20 $stop;
end

endmodule
```

Listing 2: Testbench for Question 1 (Vivado simulation).

5 Simulation Results and Output Analysis

Figure 2 shows the simulation waveform for `a`, `b`, `c`, `clk`, `q1`, and `q2`.

5.1 Expected Behavior

Because the design uses D flip-flops:

- `q1` should update only on each rising edge of `clk` to the value $(a\&b)$.
- `q2` should update only on each rising edge of `clk` to the value $(b|c)$.

Therefore, changes on inputs `a`, `b`, and `c` do not immediately change `q1` and `q2`; the outputs change only after the next rising clock edge.

5.2 Observed Behavior

In the waveform:

- `q1` transitions to 1 only when both `a` and `b` are 1 at the sampling edge.
- `q2` becomes 1 when either `b` or `c` is 1 at the sampling edge.
- Both `q1` and `q2` remain stable between rising edges, indicating correct flip-flop behavior.

Any initial unknown (`X`) values are normal at time 0 since no reset signal is defined in the circuit.

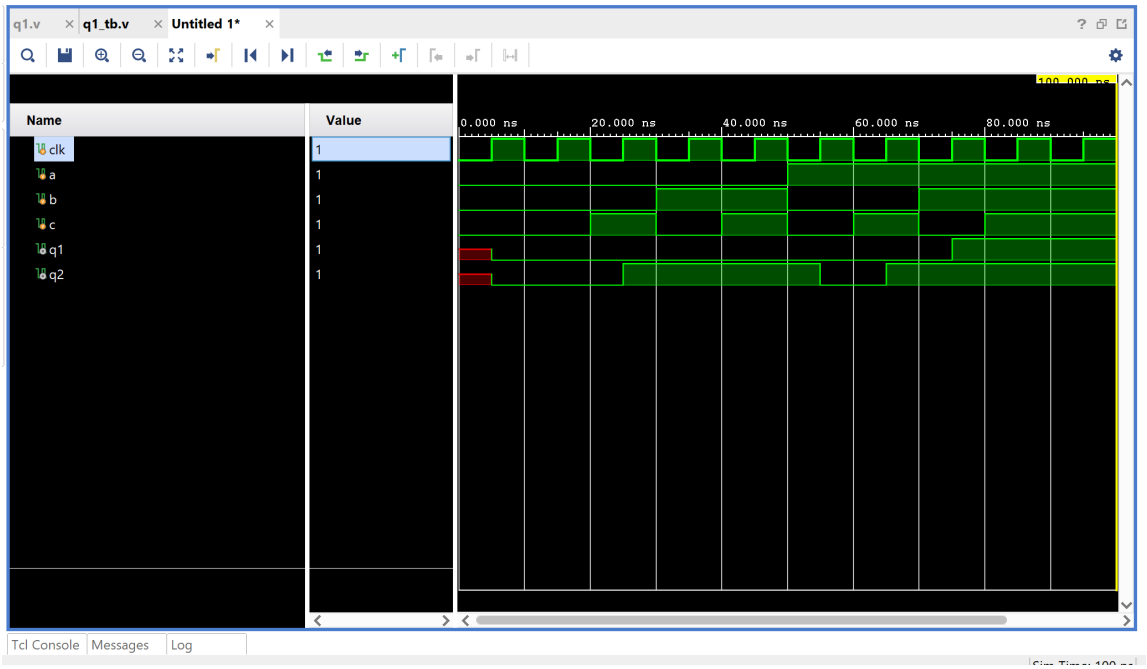


Figure 2: Simulation waveform: `q1` updates to $(a\&b)$ and `q2` updates to $(b|c)$ on each rising edge of `clk`.

6 Synthesis / Resource Utilization

The synthesized design is minimal and uses only a small number of FPGA resources because it consists of:

- Two flip-flops (for `q1` and `q2`)
- One LUT for AND logic and one LUT for OR logic (often mapped efficiently)

Figure 23 shows the Vivado utilization report for this design.

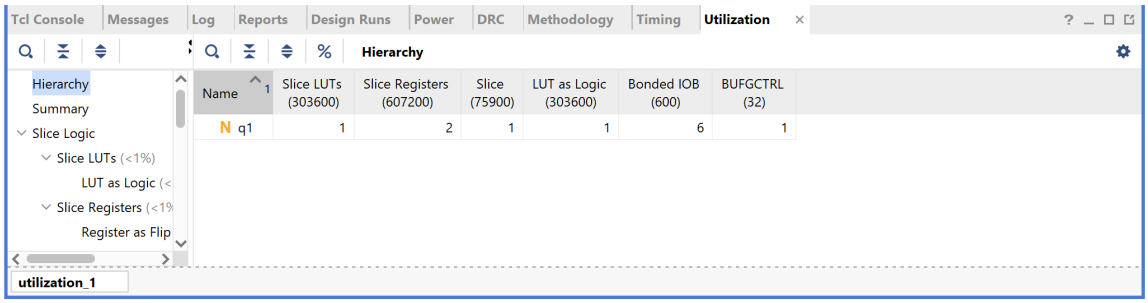


Figure 3: Vivado utilization summary for the synthesized design.

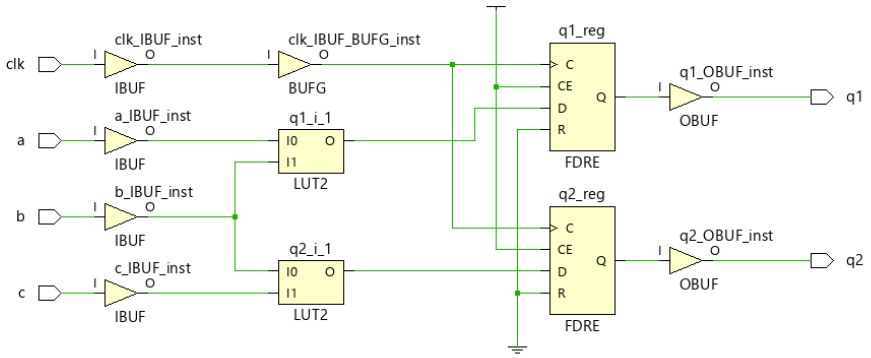


Figure 4: Vivado Schematic design.

7 Conclusion

This report implemented the given sequential circuit in Verilog by:

- Creating combinational signals $d1 = a\&b$ and $d2 = b|c$
- Capturing them on `posedge clk` using two D flip-flops

Simulation results confirm that `q1` and `q2` update only on clock edges and match the expected Boolean functions of the inputs. The utilization report verifies a compact hardware implementation.

8 Question 2: Full Adder and Ripple-Carry Adder

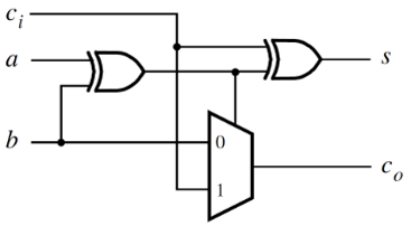
8.1 Problem Statement

Figure 5 illustrates a *full adder* circuit with inputs a, b, and c_i (carry-in), and outputs s (sum) and c_o (carry-out). The full adder computes the two-bit binary result:

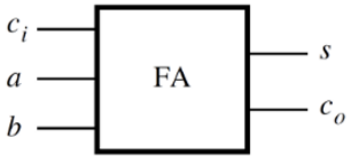
$$\{c_o, s\} = a + b + c_i$$

where the addition operator represents arithmetic addition, not logical OR.

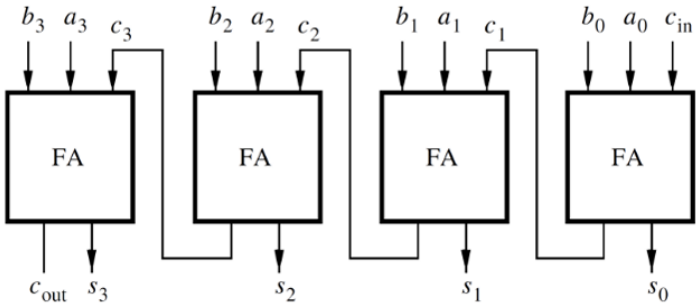
The figure also demonstrates how multiple full adder blocks can be cascaded to form a *four-bit ripple-carry adder*, where the carry output of each stage feeds the carry input of the next stage.



(a) Full adder circuit



(b) Full adder symbol



(c) Four-bit ripple-carry adder circuit

Figure 2: A ripple-carry adder circuit.

Figure 5: (a) Full adder circuit, (b) full adder symbol, and (c) four-bit ripple-carry adder.

—

8.2 Full Adder Logic

A full adder produces:

- A **sum** bit, which is the XOR of all inputs
- A **carry-out** bit, which is asserted when at least two inputs are high

The Boolean equations are:

$$s = a \oplus b \oplus c_i$$

$$c_o = (a \wedge b) \vee (a \wedge c_i) \vee (b \wedge c_i)$$

These equations match the truth table and schematic shown in the problem statement.

—

8.3 Verilog Implementation

The full adder was implemented as a purely combinational module using continuous assignments.

```
module full_adder (
    input  wire a,
    input  wire b,
```

```
    input  wire ci,
    output wire s,
    output wire co
);

assign s  = a ^ b ^ ci;
assign co = (a & b) | (a & ci) | (b & ci);

endmodule
```

Listing 3: Verilog implementation of the full adder.

—

8.4 Testbench

To verify correctness, a testbench was written to apply all eight possible input combinations of (a, b, c_i). Because the full adder is combinational, no clock signal is required.

```
`timescale 1ns/1ps

module full_adder_tb;

  reg a, b, ci;
  wire s, co;

  // Instantiate the full adder
  full_adder dut (
    .a(a),
    .b(b),
    .ci(ci),
    .s(s),
    .co(co)
  );

  initial begin
    a = 0; b = 0; ci = 0;  #10;   // 0 + 0 + 0 = 0
    a = 0; b = 0; ci = 1;  #10;   // 0 + 0 + 1 = 1
    a = 0; b = 1; ci = 0;  #10;   // 0 + 1 + 0 = 1
    a = 0; b = 1; ci = 1;  #10;   // 0 + 1 + 1 = 2
    a = 1; b = 0; ci = 0;  #10;   // 1 + 0 + 0 = 1
    a = 1; b = 0; ci = 1;  #10;   // 1 + 0 + 1 = 2
    a = 1; b = 1; ci = 0;  #10;   // 1 + 1 + 0 = 2
    a = 1; b = 1; ci = 1;  #10;   // 1 + 1 + 1 = 3

    $stop;
  end

endmodule
```

Listing 4: Testbench for the full adder.

—

8.5 Simulation Results and Analysis

Figure 6 shows the simulation waveform of the full adder.

- The `s` output correctly represents the least significant bit of the sum.
- The `co` output is asserted whenever the arithmetic sum exceeds 1.
- All output transitions occur immediately after input changes, as expected for combinational logic.

The observed results match the full adder truth table exactly.

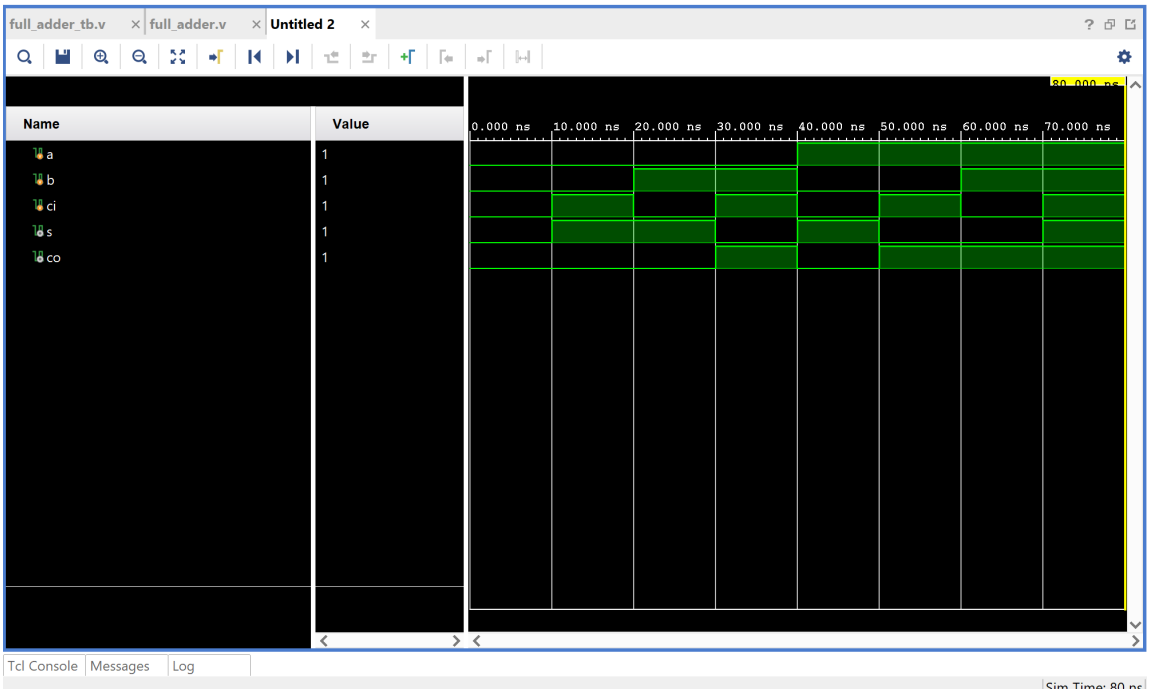


Figure 6: Simulation waveform of the full adder showing correct sum and carry behavior.

8.6 Conclusion

In this question, a full adder was successfully implemented and verified using Verilog. The design correctly computes the two-bit sum $\{c_o, s\}$ for all input combinations of a, b, and c_i. Simulation results confirm that the implementation matches the theoretical equations and the provided circuit diagrams. This full adder module can be directly used as a building block for larger arithmetic circuits such as ripple-carry adders.

8.7 Synthesis Results (Resource Utilization)

After synthesis and implementation in Vivado, the full adder maps to a very small amount of FPGA hardware. Figure 7 shows that the design uses only **one LUT** for the logic, since modern FPGA LUTs can implement multi-input Boolean functions efficiently. Because the full adder is purely combinational, it requires **no flip-flops**.

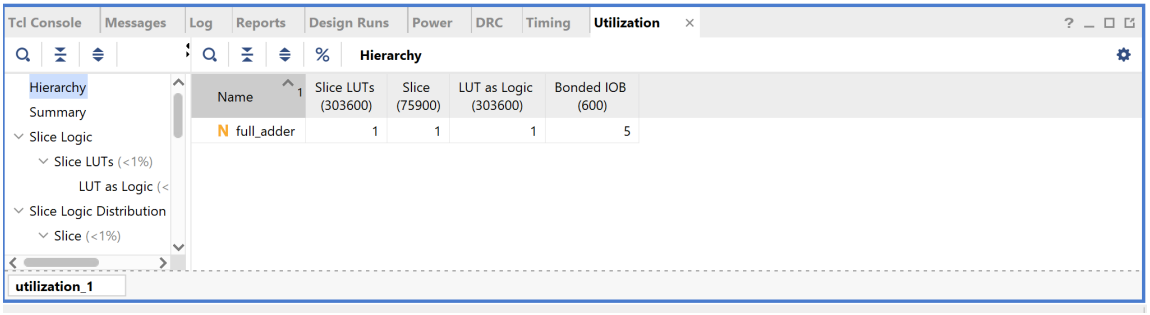


Figure 7: Vivado utilization report for the full adder design. The design uses minimal resources (one LUT as logic).

Discussion:

- The full adder outputs (s and c_o) are combinational functions of three inputs (a, b, c_i).
- FPGA tools can pack these functions into LUT resources efficiently; therefore, the utilization is very small.
- The I/O count corresponds to the ports of the module (3 inputs and 2 outputs).

8.8 Implemented Design Schematic (Technology Mapping)

Vivado's implemented schematic is shown in Figure 8. The inputs are connected through **input buffers (IBUF)** and the outputs are driven through **output buffers (OBUF)**, which is standard FPGA I/O structure. The internal logic is mapped into LUT primitives labeled **LUT3**, meaning each LUT implements a Boolean function of three inputs.

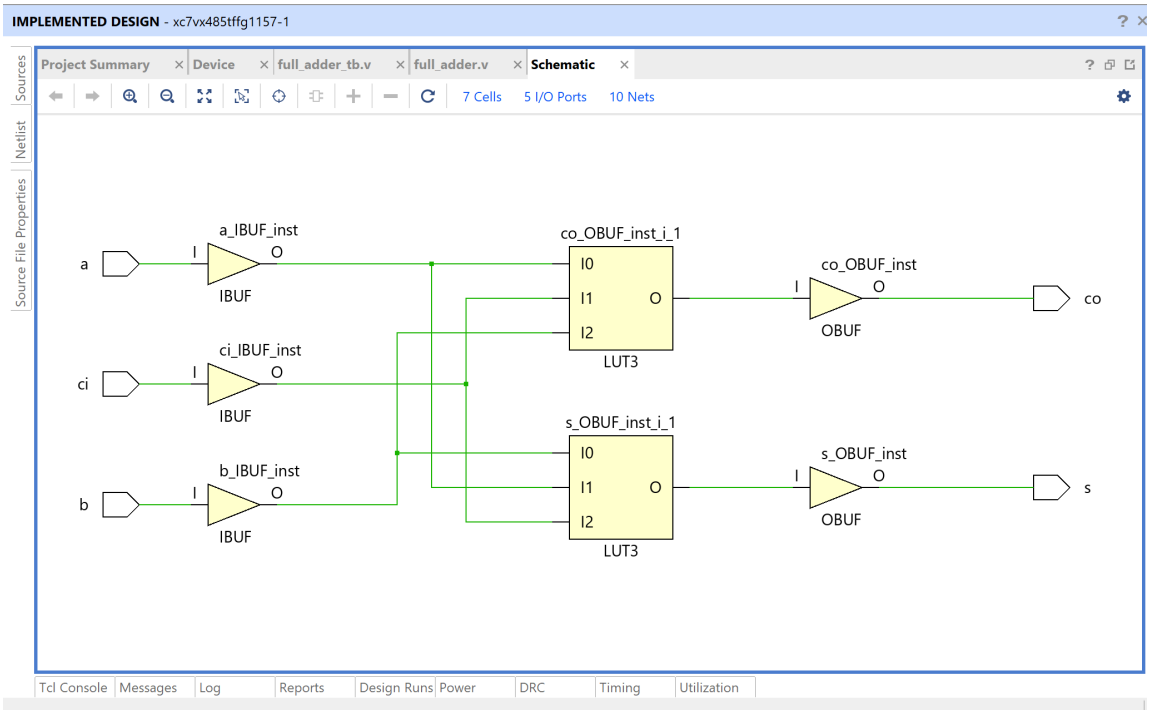


Figure 8: Implemented schematic of the full adder in Vivado. Each output is implemented using a 3-input LUT (LUT3), with IBUF/OBUF at the FPGA pins.

Discussion:

- The **sum** output $s = a \oplus b \oplus c_i$ is realized by a 3-input LUT (LUT3).
- The **carry** output $c_o = (ab) + (ac_i) + (bc_i)$ is also realized by a LUT3.
- IBUF and OBUF blocks indicate the FPGA's physical input/output buffering around the LUT logic.
- This mapping confirms that the Verilog code is synthesized exactly as intended for the full adder.

8.9 Four-Bit Ripple-Carry Adder Implementation

Using the full adder module developed in the previous section, a four-bit ripple-carry adder was constructed as shown in Figure **??**. This circuit adds two 4-bit binary numbers $a[3:0]$ and $b[3:0]$ along with an input carry c_{in}, producing a 4-bit sum $s[3:0]$ and a final carry-out c_{out}.

The carry signal propagates sequentially from the least significant bit (LSB) to the most significant bit (MSB), which is why this design is referred to as a *ripple-carry adder*.

8.10 Design Description

The design consists of four full adder instances connected in series:

- The first full adder computes the sum of the LSBs using c_{in}
- Each subsequent full adder receives the carry-out from the previous stage
- The final carry-out is produced by the MSB full adder

Mathematically, the circuit computes:

$$\{c_{out}, s[3:0]\} = a[3:0] + b[3:0] + c_{in}$$

8.11 Verilog Implementation

The Verilog module for the four-bit ripple-carry adder is shown in Listing 5.

```
module ripple_carry_adder4 (
    input  wire [3:0] a,
    input  wire [3:0] b,
    input  wire       cin,
    output wire [3:0] s,
    output wire       cout
);

wire c1, c2, c3;

full_adder fa0 (.a(a[0]), .b(b[0]), .ci(cin), .s(s[0]), .co(c1));
full_adder fa1 (.a(a[1]), .b(b[1]), .ci(c1),  .s(s[1]), .co(c2));
full_adder fa2 (.a(a[2]), .b(b[2]), .ci(c2),  .s(s[2]), .co(c3));
```

```
full_adder fa3 (.a(a[3]), .b(b[3]), .ci(c3),  .s(s[3]), .co(cout)
   );

endmodule
```

Listing 5: Verilog implementation of the 4-bit ripple-carry adder.

8.12 Testbench

A testbench was written to verify the operation of the ripple-carry adder using representative test cases, including overflow scenarios.

```
`timescale 1ns/1ps

module ripple_carry_adder4_tb;

reg  [3:0] a, b;
reg        cin;
wire [3:0] s;
wire       cout;

ripple_carry_adder4 dut (
    .a(a), .b(b), .cin(cin),
    .s(s), .cout(cout)
);

initial begin
    // test 1: 3 + 5 = 8
    a = 4'd3;  b = 4'd5;  cin = 1'b0;
    #10;

    // test 2: 9 + 6 = 15
    a = 4'd9;  b = 4'd6;  cin = 1'b0;
    #10;

    // test 3: 15 + 1 = 16 -> s=0, cout=1
    a = 4'd15; b = 4'd1;  cin = 1'b0;
    #10;

    // test 4: 7 + 8 + cin(1) = 16 -> s=0, cout=1
    a = 4'd7;  b = 4'd8;  cin = 1'b1;
    #10;

    $stop;
end

endmodule
```

Listing 6: Testbench for the 4-bit ripple-carry adder.

8.13 Simulation Results and Output Analysis

Figure 9 shows the simulation waveform for the ripple-carry adder.

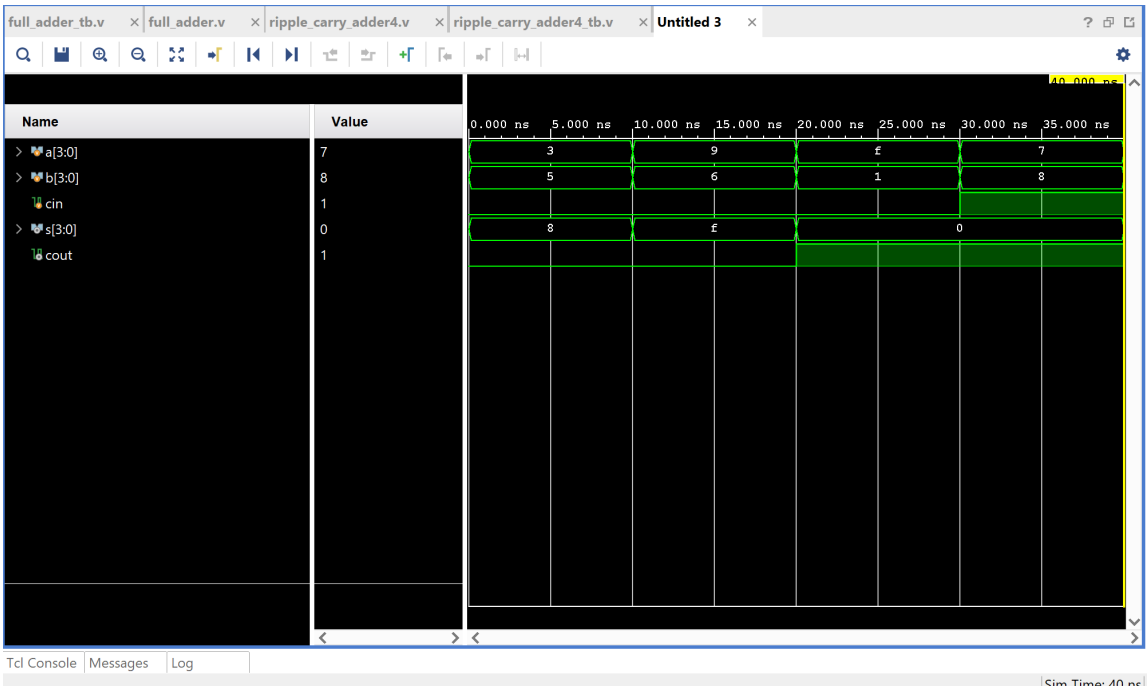


Figure 9: Simulation waveform of the 4-bit ripple-carry adder.

Observed Results:

- For $a = 3$, $b = 5$, $c_{in} = 0$: $s = 8$, $c_{out} = 0$
- For $a = 9$, $b = 6$, $c_{in} = 0$: $s = 15$, $c_{out} = 0$
- For $a = 15$, $b = 1$, $c_{in} = 0$: $s = 0$, $c_{out} = 1$ (overflow)
- For $a = 7$, $b = 8$, $c_{in} = 1$: $s = 0$, $c_{out} = 1$ (overflow)

These results confirm correct ripple-carry behavior. The carry propagates through each full adder stage until the final sum and carry-out are produced.

8.14 Discussion

The ripple-carry adder correctly performs multi-bit binary addition using modular full adder blocks. Although simple and easy to implement, ripple-carry adders suffer from carry propagation delay, as each bit must wait for the previous carry to resolve. Despite this limitation, the ripple-carry adder is widely used due to its simplicity and clear structure.

8.15 Conclusion

The four-bit ripple-carry adder was successfully implemented using four cascaded full adders. Simulation results demonstrate correct functionality for normal addition and overflow cases. This design completes the implementation of the full adder–based arithmetic circuit described in Question 2.

9 Question 3(a): JK Flip-Flop Design

9.1 Problem Statement

The objective of this question is to design a *JK flip-flop* module. A JK flip-flop is a sequential storage element with inputs J, K, a clock signal, and outputs Q and $\overline{Q}$. Depending on the values of J and K, the flip-flop can hold its state, set, reset, or toggle its output.

9.2 JK Flip-Flop Operation

The behavior of a JK flip-flop on the rising edge of the clock is summarized as follows:

J	K	Next State Q^+
0	0	Hold (Q)
0	1	Reset (0)
1	0	Set (1)
1	1	Toggle ($\overline{Q}$)

An asynchronous active-high reset is included to initialize the output to a known state.

9.3 Verilog Implementation

The JK flip-flop is implemented using a sequential `always` block triggered by the rising edge of the clock or the reset signal. The complement output $\overline{Q}$ is generated using a continuous assignment.

```
module jk_ff (
    input  wire clk,
    input  wire rst,      // async reset (active-high)
    input  wire j,
    input  wire k,
    output reg  q,
    output wire qbar
);

assign qbar = ~q;

always @(posedge clk or posedge rst) begin
    if (rst) begin
        q <= 1'b0;
    end else begin
        case ({j,k})
            2'b00: q <= q;       // hold
            2'b01: q <= 1'b0;    // reset
            2'b10: q <= 1'b1;    // set
            2'b11: q <= ~q;      // toggle
        endcase
    end
end
```

```
endmodule
```

Listing 7: Verilog implementation of the JK flip-flop.

9.4 Testbench

A self-checking testbench was written to verify the correct operation of the JK flip-flop. The testbench applies all major JK input combinations and checks the output against the expected behavior after each rising clock edge.

```
`timescale 1ns/1ps

module jk_ff_tb;

  reg clk, rst;
  reg j, k;
  wire q, qbar;

  jk_ff dut (
    .clk(clk), .rst(rst),
    .j(j), .k(k),
    .q(q), .qbar(qbar)
  );

  // 10 ns clock
  always #5 clk = ~clk;

  task step_and_check(input reg exp_q, input [1:0] jk);
    begin
      {j,k} = jk;
      @(posedge clk);
      #1;
      if (q !== exp_q) begin
        $display("FAIL time=%0t JK=%b exp_q=%b got_q=%b",
                 $time, jk, exp_q, q);
        $stop;
      end else begin
        $display("PASS time=%0t JK=%b q=%b qbar=%b",
                 $time, jk, q, qbar);
      end
    end
  endtask

  initial begin
    clk = 0; rst = 1; j = 0; k = 0;

    // asynchronous reset
    #2 rst = 1;
    #8 rst = 0;

```

```
    step_and_check(1'b0, 2'b00); // hold
    step_and_check(1'b1, 2'b10); // set
    step_and_check(1'b1, 2'b00); // hold
    step_and_check(1'b0, 2'b01); // reset
    step_and_check(1'b1, 2'b11); // toggle
    step_and_check(1'b0, 2'b11); // toggle

    $display("JK FF TEST PASSED");
    $stop;
  end

endmodule
```

Listing 8: Testbench for the JK flip-flop.

9.5 Simulation Results and Output Analysis

Figure 10 shows the simulation waveform of the JK flip-flop.

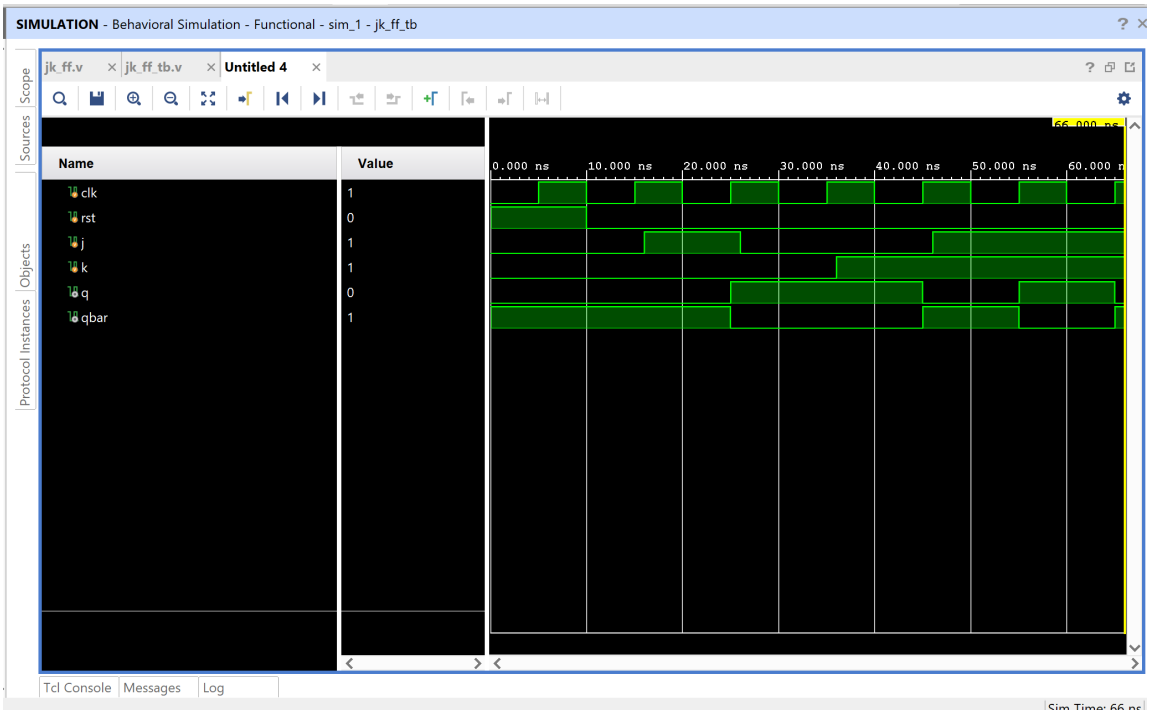


Figure 10: Simulation waveform of the JK flip-flop showing reset, hold, set, reset, and toggle operations.

Observed Behavior:

- When `rst` is asserted, Q is immediately cleared to 0.
- With $J = 0$ and $K = 0$, the output holds its previous value.
- With $J = 1$ and $K = 0$, the output is set to 1.
- With $J = 0$ and $K = 1$, the output is reset to 0.

- With $J = 1$ and $K = 1$, the output toggles on each rising edge of the clock.

The waveform confirms that the flip-flop responds only on clock edges (except for reset), which is the correct behavior for a synchronous JK flip-flop with asynchronous reset.

9.6 Conclusion

A JK flip-flop with asynchronous active-high reset was successfully designed and verified. The Verilog implementation correctly follows the JK truth table, and simulation results confirm proper hold, set, reset, and toggle behavior. This module can be used as a fundamental building block in counters and other sequential digital systems.

10 Question 3(b): 4-bit Up-Counter Using JK Flip-Flops

10.1 Problem Statement

Using the JK flip-flop designed in Question 3(a) and the circuit diagram provided in Figure 11, design a 4-bit up-counter. The counter output should increment by one on every rising edge of the clock and wrap around after reaching 15 (i.e., modulo-16 behavior). A reset input is used to initialize the counter output to zero.

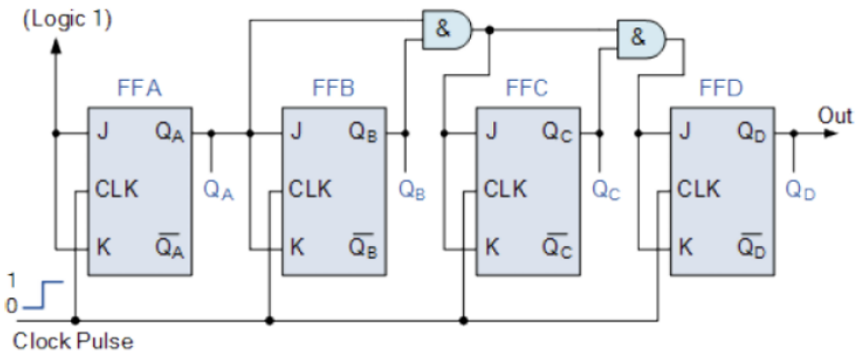


Figure 3: Counter With JK Flip-Flop

Figure 11: Counter with JK flip-flops (given diagram).

10.2 Design Approach

A JK flip-flop toggles its output when $J = K = 1$. Therefore, by setting $J = K = T$, a JK flip-flop behaves like a T flip-flop:

$$T = 1 \Rightarrow \text{toggle}, \qquad T = 0 \Rightarrow \text{hold}$$

To construct a synchronous 4-bit up-counter:

- Bit 0 toggles every clock: $T_0 = 1$
- Bit 1 toggles when bit 0 is 1: $T_1 = Q_A$
- Bit 2 toggles when bits 1:0 are 11: $T_2 = Q_A Q_B$
- Bit 3 toggles when bits 2:0 are 111: $T_3 = Q_A Q_B Q_C$

This matches the AND-gate enable structure shown in the diagram.

10.3 Verilog Implementation

The 4-bit up-counter is built by instantiating four JK flip-flops and driving each one with the proper toggle-enable signal.

```
module upcounter4_jk (
    input  wire clk,
    input  wire rst,
    output wire [3:0] out
);
wire qa, qb, qc, qd;

wire t0 = 1'b1;
wire t1 = qa;
wire t2 = qa & qb;
wire t3 = qa & qb & qc;

jk_ff FFA (.clk(clk), .rst(rst), .j(t0), .k(t0), .q(qa), .qbar())
   ;
jk_ff FFB (.clk(clk), .rst(rst), .j(t1), .k(t1), .q(qb), .qbar())
   ;
jk_ff FFC (.clk(clk), .rst(rst), .j(t2), .k(t2), .q(qc), .qbar())
   ;
jk_ff FFD (.clk(clk), .rst(rst), .j(t3), .k(t3), .q(qd), .qbar())
   ;

assign out = {qd, qc, qb, qa};

endmodule
```

Listing 9: Verilog implementation of 4-bit JK up-counter.

10.4 Testbench

A clock is generated with a 10 ns period. The reset is asserted initially and then released, allowing the counter to run.

```
`timescale 1ns/1ps

module upcounter4_jk_tb;

  reg clk;
  reg rst;
```

```
  wire [3:0] out;

  upcounter4_jk dut (
    .clk(clk),
    .rst(rst),
    .out(out)
  );

  // Clock generation: 10 ns period
  always #5 clk = ~clk;

  initial begin
    clk = 0;
    rst = 1;

    // Apply reset
    #15;
    rst = 0;

    // Let the counter run
    #200;

    $stop;
  end

endmodule
```

Listing 10: Testbench for the 4-bit JK up-counter.

10.5 Simulation Results and Output Analysis

Figure 12 shows the simulation waveform. After reset is deasserted, the output increments on each rising edge of the clock and cycles through hexadecimal values:

$$0, 1, 2, \ldots, 9, a, b, c, d, e, f, 0, 1, \ldots$$

This confirms correct modulo-16 up-counter behavior.

Figure 12: Simulation waveform of the 4-bit JK up-counter.

10.6 Implemented Schematic View

Vivado's schematic view confirms the design is composed of four JK flip-flop instances with the correct interconnections, and the 4-bit output is routed through output buffers.

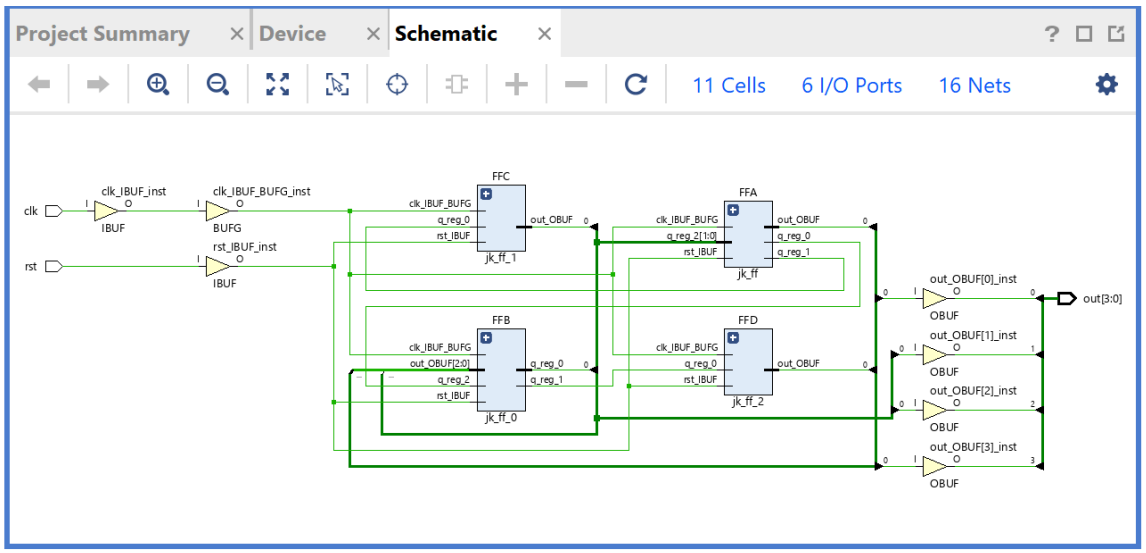


Figure 13: Implemented schematic of the 4-bit JK up-counter in Vivado.

10.7 Resource Utilization

The utilization report in Figure 14 shows the counter uses a small number of FPGA resources. As expected, the design requires four registers (one per bit) and a few LUTs to implement the AND enable logic.

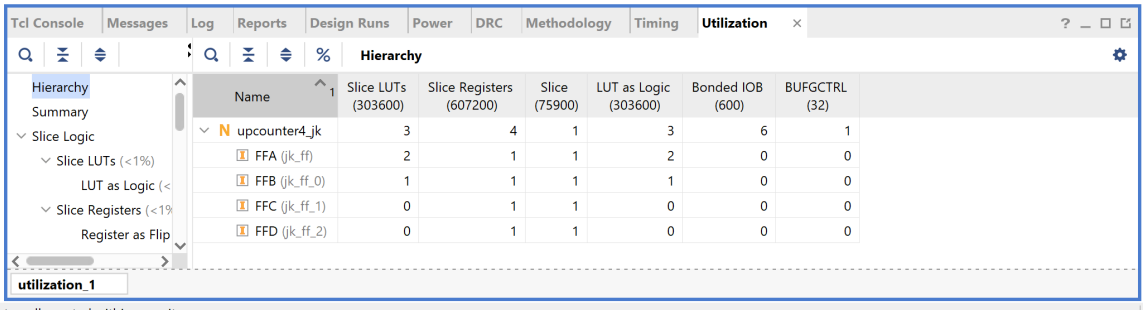


Figure 14: Vivado utilization report for the 4-bit JK up-counter.

10.8 Timing Summary

Figure 15 shows the timing summary. The report indicates that timing constraints are met (no failing endpoints), confirming the counter can operate correctly at the specified clock constraint in the project.

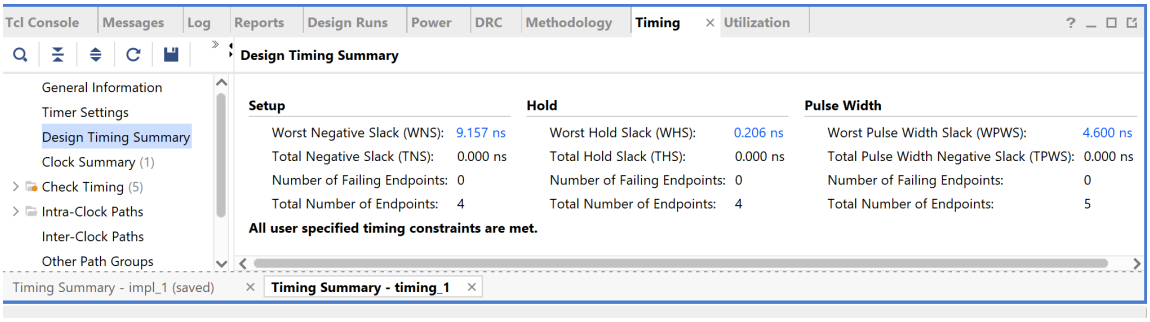


Figure 15: Vivado timing summary for the 4-bit JK up-counter.

10.9 Conclusion

A 4-bit synchronous up-counter was successfully implemented using four JK flip-flops. By driving each JK flip-flop with the appropriate toggle-enable condition, the design increments by one on each rising edge and wraps after 15. Simulation, schematic mapping, utilization, and timing results confirm correct operation.

11 Question 3(c): RTL Implementation of a 4-bit Up-Counter

11.1 Problem Statement

Design a 4-bit up-counter directly using an `always` block. The counter must increment on every rising edge of the clock and reset asynchronously to zero when the reset signal is asserted.

11.2 Design Approach

Instead of explicitly instantiating JK flip-flops, this design uses a register-based RTL description. The counter is implemented using a sequential `always` block that triggers

on the rising edge of the clock or the rising edge of the reset signal.

When reset is asserted, the output is cleared to zero. Otherwise, the counter increments by one on each clock cycle. Since the output is 4 bits wide, the counter naturally wraps around after reaching 15, implementing modulo-16 behavior.

11.3 Verilog Implementation

The RTL-based counter is shown in Listing 11.

```
module upcounter4_rtl (
    input  wire clk,
    input  wire rst,          // async reset (active-high)
    output reg  [3:0] out
);

always @(posedge clk or posedge rst) begin
    if (rst)
        out <= 4'b0000;
    else
        out <= out + 1'b1;
end

endmodule
```

Listing 11: RTL Verilog implementation of 4-bit up-counter.

11.4 Testbench

The testbench generates a 10 ns clock and applies an initial reset before allowing the counter to run freely.

```
`timescale 1ns/1ps

module upcounter4_rtl_tb;

  reg clk;
  reg rst;
  wire [3:0] out;

  upcounter4_rtl dut (
    .clk(clk),
    .rst(rst),
    .out(out)
  );

  // Clock generation: 10 ns period
  always #5 clk = ~clk;

  initial begin
    clk = 0;
    rst = 1;

```

```
22      // Apply reset
23      #15;
24      rst = 0;
25
26      // Let counter run freely
27      #200;
28
29      $stop;
30    end
31
32  endmodule
```

Listing 12: Testbench for RTL up-counter.

11.5 Simulation Results

Figure 16 shows the simulation waveform. After reset is deasserted, the output increments sequentially:

$$0, 1, 2, \ldots, 9, a, b, c, d, e, f, 0, 1, \ldots$$

This confirms correct modulo-16 up-counter behavior.

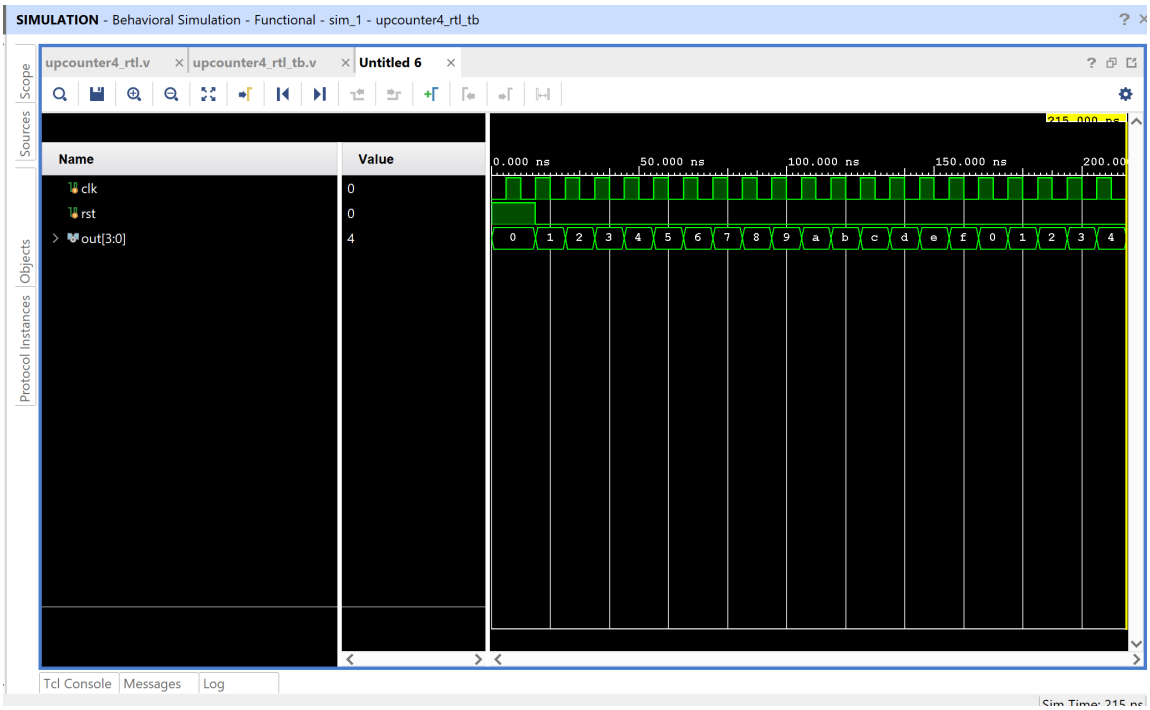


Figure 16: Simulation waveform of RTL 4-bit up-counter.

11.6 Implemented Schematic

The synthesized schematic in Figure 17 shows four flip-flops with minimal combinational logic, reflecting the simplicity of the RTL description.

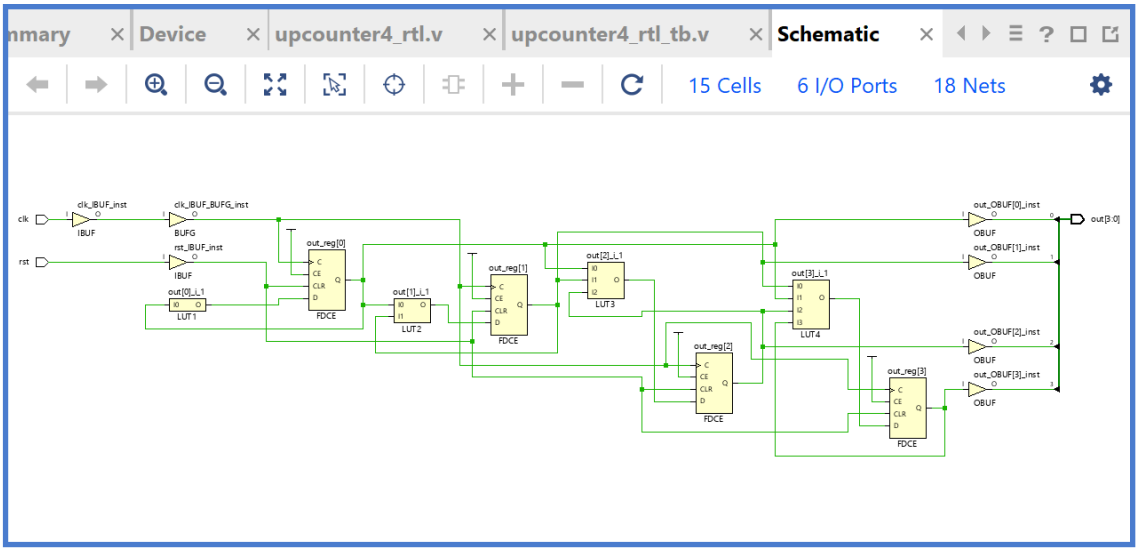


Figure 17: Implemented schematic of RTL up-counter in Vivado.

11.7 Resource Utilization

Figure 18 presents the utilization report. Compared to the JK-based counter, this RTL implementation uses fewer LUTs while still requiring four registers.

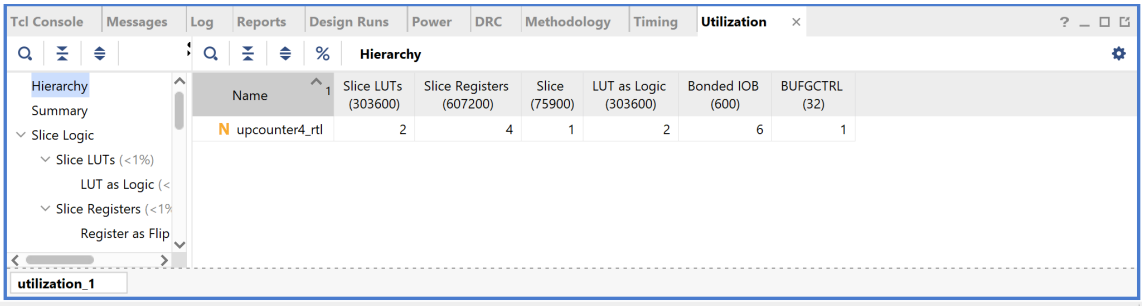


Figure 18: Vivado utilization report for RTL up-counter.

11.8 Timing Analysis

The timing summary in Figure 19 confirms that all setup, hold, and pulse width constraints are met, with no failing paths reported.

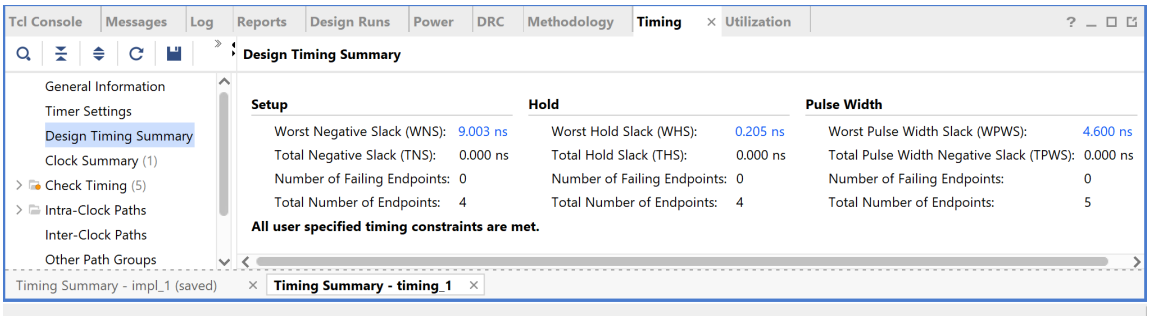


Figure 19: Timing summary for RTL up-counter.

11.9 Comparison with JK-Based Counter

Compared to the JK flip-flop implementation in Question 3(b), the RTL counter:

- Uses fewer LUTs
- Is simpler and more concise
- Is easier to scale to wider counters

Both designs, however, produce identical functional behavior.

11.10 Conclusion

A 4-bit up-counter was successfully implemented using a direct RTL approach with an `always` block. Simulation, schematic, utilization, and timing results verify correct operation and demonstrate the efficiency of RTL-based counter design.

12 Question 3(d): Design Comparison

12.1 Problem Statement

Compare the designs of the 4-bit up-counters implemented in Question 3(b) (JK flip-flop based counter) and Question 3(c) (RTL counter using an `always` block) in terms of:

- RTL schematic structure
- Maximum clock speed
- Resource utilization

The comparison is performed using synthesis and implementation results generated by Xilinx Vivado for the target FPGA device.

12.2 RTL Schematic Comparison

The RTL schematic of the JK-based counter (Question 3(b)) shows four explicitly instantiated JK flip-flops connected through AND-gate logic that generates the toggle enable signals for each counter bit. This design closely follows the theoretical JK counter structure typically presented in digital design textbooks.

In contrast, the RTL schematic of the always-block counter (Question 3(c)) consists of four D-type registers and a small incrementer circuit that computes the next state `out` + 1. The RTL description allows the synthesis tool to infer the optimal hardware implementation automatically.

Overall, the JK-based design is more structural and explicit, while the RTL-based design is higher-level and more abstract.

12.3 Resource Utilization Comparison

Figures from the Vivado utilization reports indicate the following resource usage:

Table 1: Resource utilization comparison of the two counter designs.

Design	Slice LUTs	Registers	BUFG	IOBs
JK Counter (Q3b)	3	4	1	6
RTL Counter (Q3c)	2	4	1	6

Both designs require four registers, corresponding to the four counter bits. However, the RTL counter uses fewer LUTs than the JK-based design. This reduction is due to Vivado's ability to optimize the incrementer logic more efficiently than the explicit AND-gate toggle network used in the JK design.

12.4 Maximum Clock Speed Comparison

Both designs were constrained with a clock period of 10 ns (100 MHz), as shown in the Clock Summary report. The maximum achievable clock speed was estimated using the Worst Negative Slack (WNS) reported by Vivado.

JK-Based Counter

$$T_{\text{min}} = 10.000 \text{ ns} - 9.157 \text{ ns} = 0.843 \text{ ns}$$

$$F_{\text{max}} \approx \frac{1}{0.843 \text{ ns}} \approx 1186 \text{ MHz}$$

RTL Counter

$$T_{\text{min}} = 10.000 \text{ ns} - 9.003 \text{ ns} = 0.997 \text{ ns}$$

$$F_{\text{max}} \approx \frac{1}{0.997 \text{ ns}} \approx 1003 \text{ MHz}$$

12.5 Timing Summary Interpretation

Both counters meet all timing constraints, with no failing endpoints reported. The JK-based design exhibits a slightly higher maximum theoretical clock frequency due to a marginally simpler critical path. However, the difference is small and not significant for practical operation at moderate clock speeds such as 100 MHz.

12.6 Overall Comparison and Discussion

- The RTL counter is more resource-efficient, using fewer LUTs.
- The JK counter closely matches the conceptual flip-flop-based design.
- The JK counter achieves a slightly higher theoretical maximum clock speed.
- Both designs operate correctly well above the required clock frequency.

12.7 Conclusion

Both counter designs successfully implement a 4-bit up-counter. The JK-based design provides clear insight into flip-flop-level counter construction, while the RTL always-block implementation offers a simpler, more compact, and scalable solution. In practice, the RTL approach is preferred for FPGA design due to its reduced resource usage and ease of development, while the JK implementation remains valuable for educational purposes.

Question 4

The following finite impulse response (FIR) filter is considered:

$$y(n) = ax(n) + bx(n-1) + cx(n-2) \tag{1}$$

This report presents the design, implementation, simulation, and timing analysis for the architecture with the **lowest latency**.

13 (a) Lowest Latency FIR Filter

13.1 Proposed Architecture

Figure 20 shows the hand-drawn architecture of the proposed FIR filter. The design follows the direct-form FIR structure and minimizes latency by avoiding any pipelining in the arithmetic datapath.

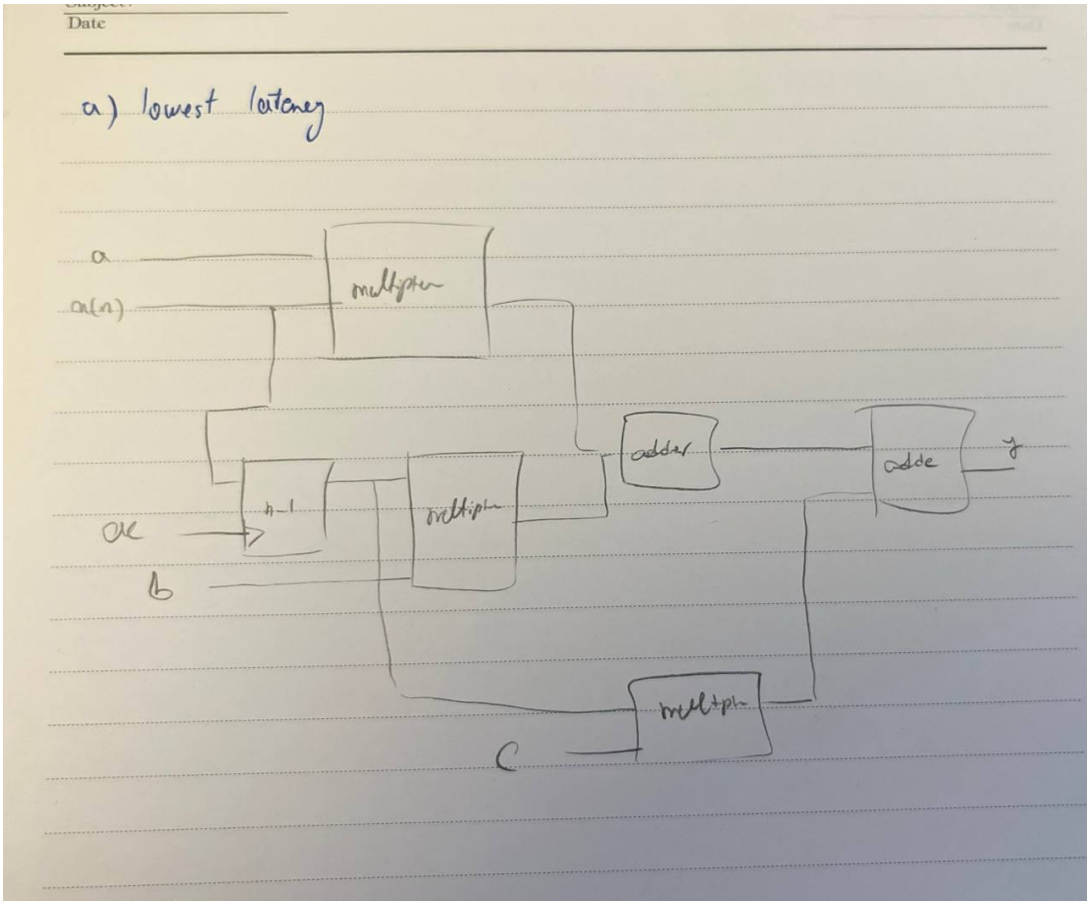


Figure 20: Hand-drawn lowest-latency FIR filter architecture

The architecture consists of:

- Two delay elements implementing $x(n-1)$ and $x(n-2)$
- Three multipliers for coefficients a, b, and c
- Two adders to accumulate the partial products

Since no pipeline registers are inserted in the multiply–add chain, the output is produced with minimum latency.

13.2 RTL Implementation

The FIR filter was implemented in Verilog using fixed-point arithmetic. The delay elements are implemented using registers, and the output is registered to ensure synchronous operation.

13.3 RTL Schematic

Figure 21 shows the RTL schematic generated by Vivado after synthesis. The diagram confirms the presence of the expected multipliers, adders, and delay registers corresponding to the direct-form FIR architecture.

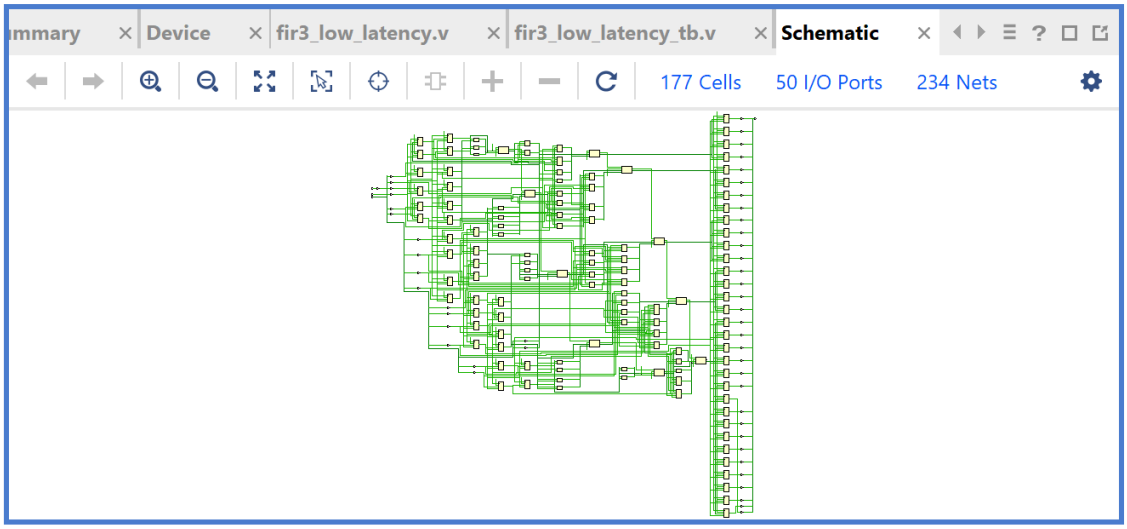


Figure 21: RTL schematic of the lowest-latency FIR filter

The absence of pipeline registers in the datapath confirms that the design is optimized for minimum latency.

13.4 Simulation Results

Behavioral simulation was performed to verify functional correctness. Figure 22 shows the simulation waveform.

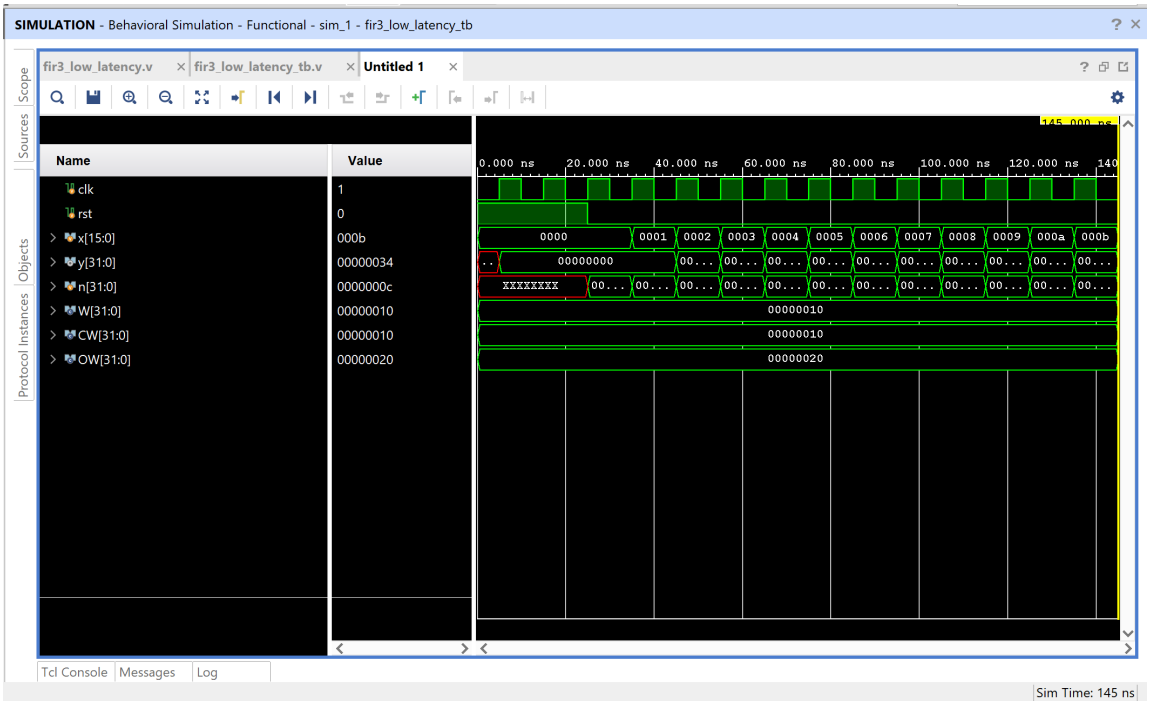


Figure 22: Behavioral simulation waveform of the FIR filter

From the waveform:

- Initial undefined values occur before reset due to uninitialized registers

- After reset deassertion, the delay line fills correctly
- The output follows the FIR equation exactly

For example, with coefficients $a = 1$, $b = 2$, and $c = 3$, the output matches:

$$y(n) = x(n) + 2x(n-1) + 3x(n-2)$$

This confirms the correctness of the implementation.

13.5 Resource Utilization

Figure 23 shows the post-synthesis resource utilization report.

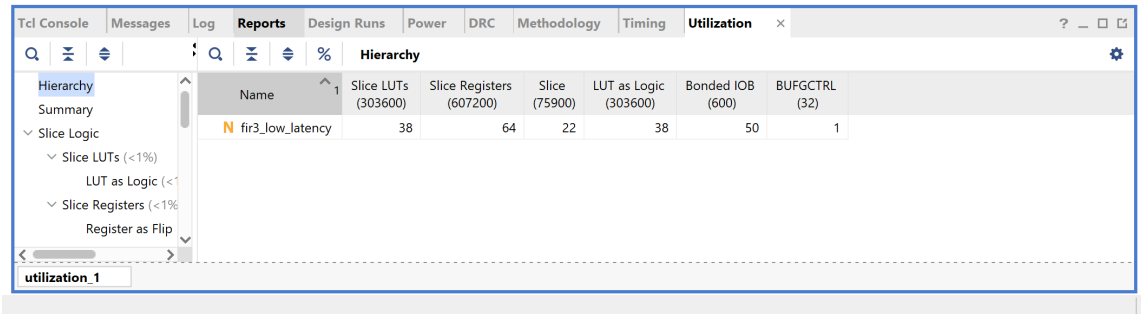


Figure 23: Post-synthesis resource utilization

The design uses:

- 38 Slice LUTs
- 64 Slice Registers
- DSP blocks inferred for multipliers
- 1 BUFG

The low utilization is a result of the compact direct-form structure and the absence of pipeline registers.

13.6 Timing Analysis Without Clock Constraint

Figure 24 shows the timing report before applying a clock constraint.

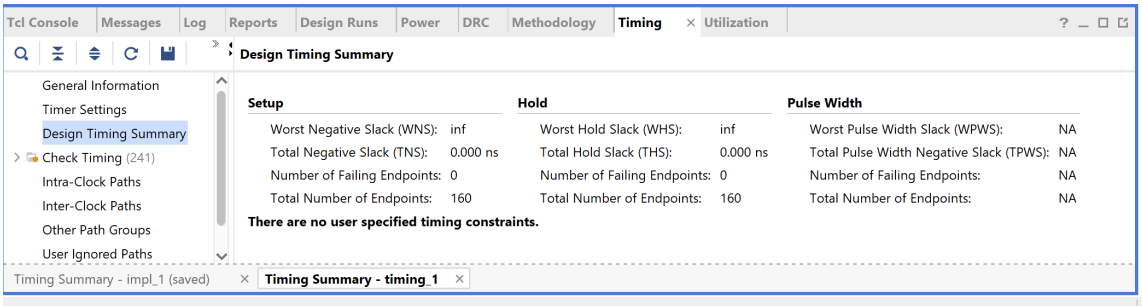


Figure 24: Timing summary without user-defined clock constraint

Since no clock constraint is defined, Vivado reports infinite slack and cannot determine the maximum operating frequency.

13.7 Timing Analysis With Clock Constraint

A clock constraint of 10 was applied. Figure 25 shows the post-implementation timing results.

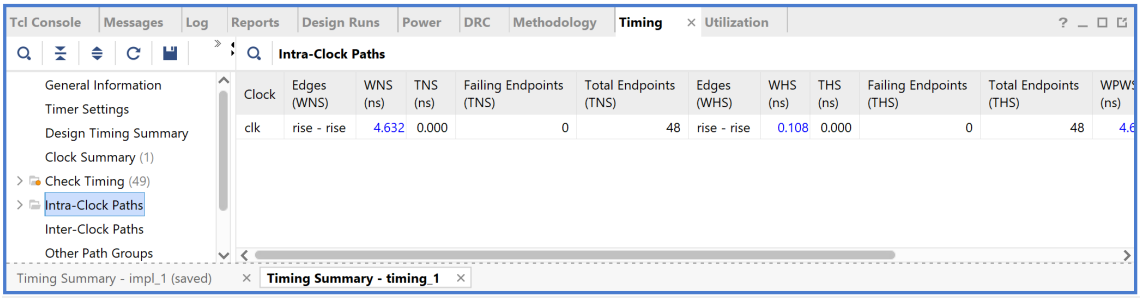


Figure 25: Timing summary with 10 ns clock constraint

The reported worst negative slack (WNS) is:

$$\text{WNS} = +4.632 \text{ ns}$$

The minimum achievable clock period is:

$$T_{\text{min}} = 10 - 4.632 = 5.368 \text{ ns}$$

Thus, the maximum operating frequency is:

$$F_{\text{max}} \approx 186 \text{ MHz}$$

13.8 Critical Path Discussion

The critical path of the design spans:

- A register holding a delayed input sample
- One multiplier
- Two cascaded adders
- The output register

The lack of pipelining results in a long combinational path, which limits the maximum achievable clock frequency despite the low latency.

13.9 Conclusion

The lowest-latency FIR filter design achieves minimal sample delay and low hardware utilization. However, the long combinational critical path limits the maximum operating frequency. This highlights the trade-off between latency and timing performance in FIR filter architectures.

14 (b) Highest Throughput FIR Filter

14.1 Proposed Architecture

Figure 26 shows the hand-drawn architecture of the highest-throughput FIR filter. In this design, pipeline registers are inserted between arithmetic blocks in order to reduce the combinational delay and maximize the achievable clock frequency.

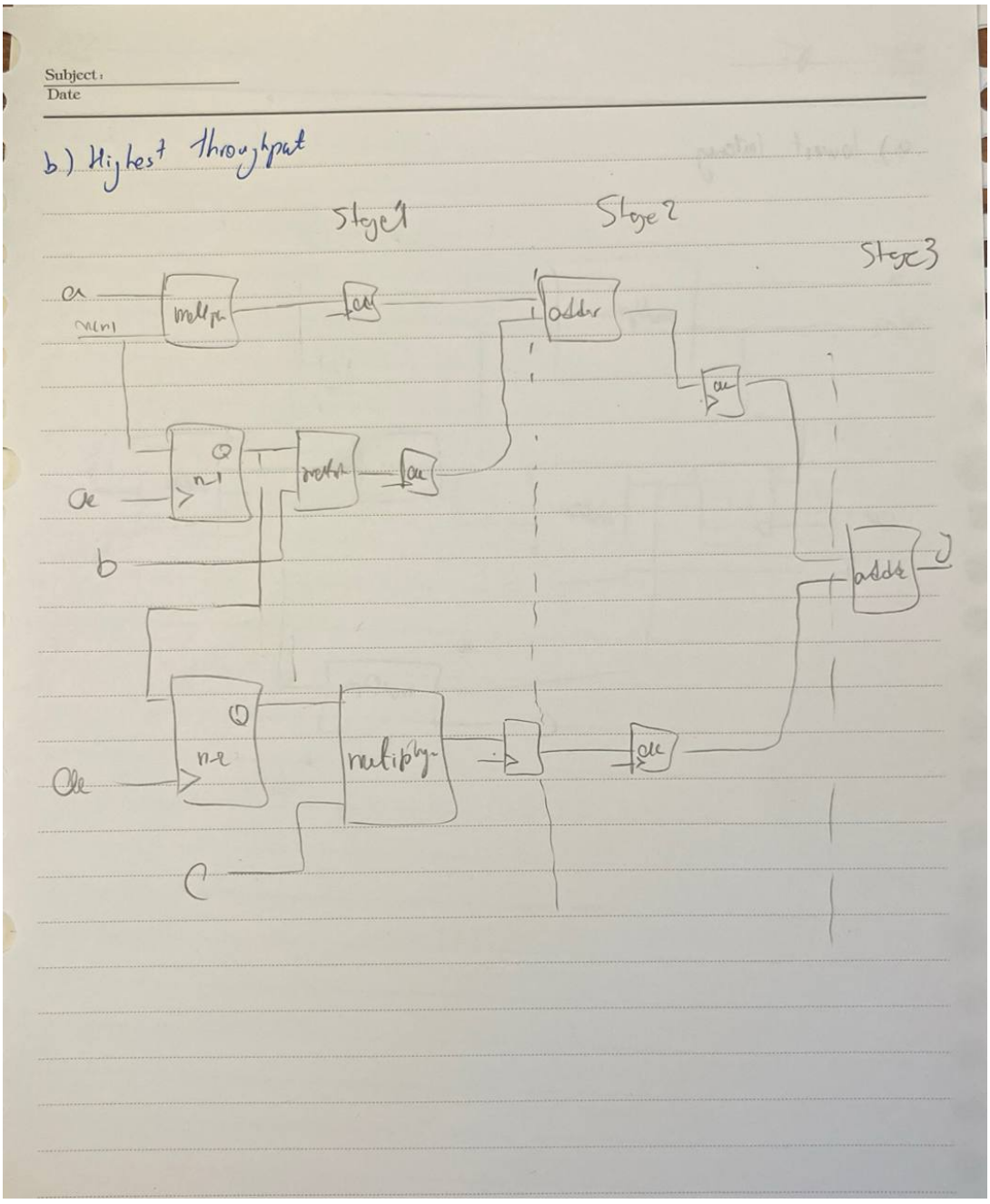


Figure 26: Hand-drawn pipelined FIR filter architecture for highest throughput

The architecture is divided into multiple pipeline stages:

- Stage 1: Multiplication of input samples with coefficients
- Stage 2: First addition and pipeline alignment

- Stage 3: Final accumulation and output register

By inserting registers between stages, the critical path is significantly shortened, allowing the circuit to operate at a much higher clock frequency.

14.2 RTL Implementation

The FIR filter was implemented in Verilog using a fully pipelined datapath. Pipeline registers are placed after each multiplier and adder to maximize throughput. Although this increases the latency in terms of clock cycles, the design is capable of processing one input sample per clock cycle at a higher operating frequency.

14.3 RTL Schematic

Figure 28 shows the RTL schematic generated by Vivado after synthesis. Compared to the lowest-latency design, a larger number of registers is visible, confirming the insertion of pipeline stages in the datapath.

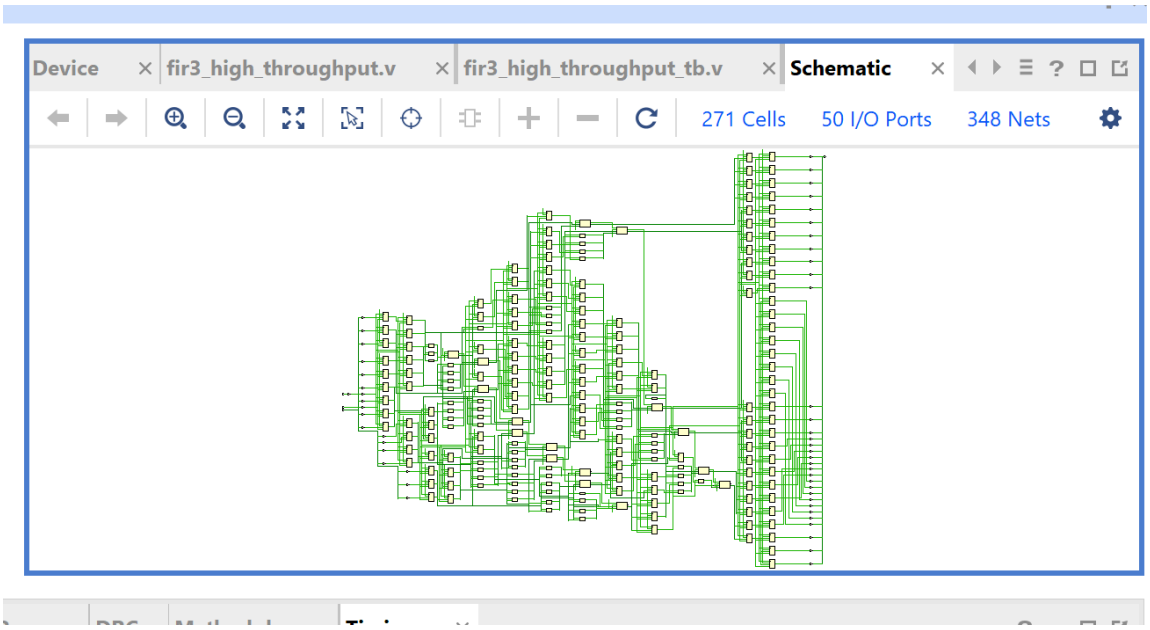


Figure 27: RTL schematic of the highest-throughput FIR filter

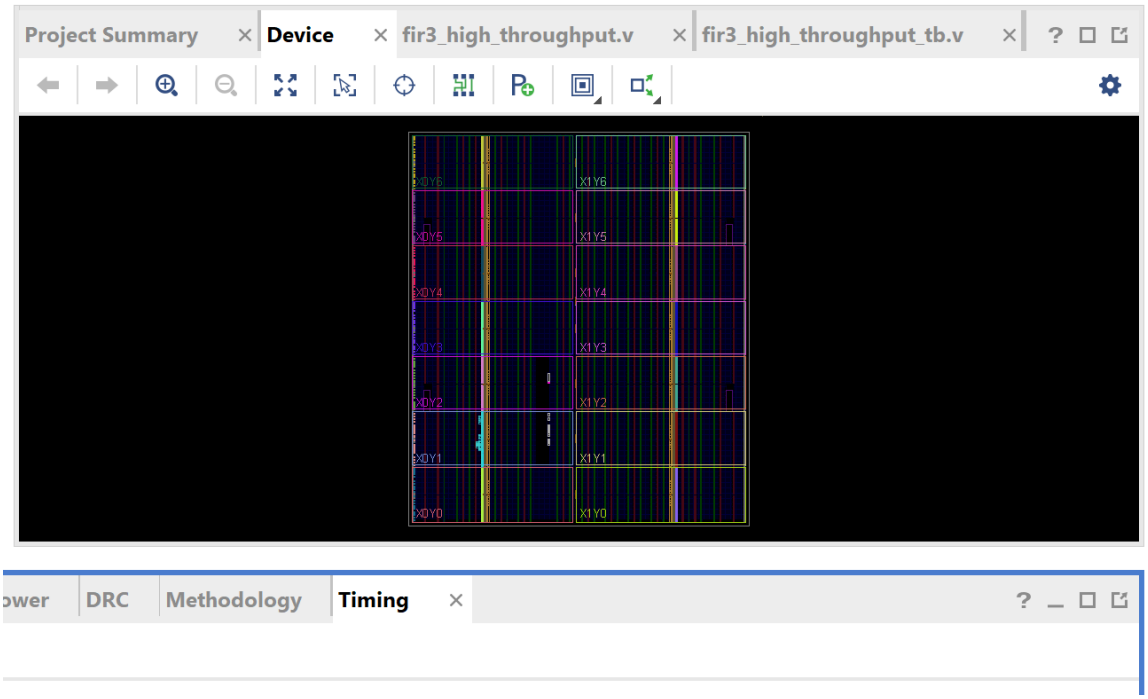


Figure 28: RTL schematic of the highest-throughput FIR filter

The schematic clearly illustrates the pipelined structure, which is responsible for reducing the combinational delay of the critical path.

14.4 Simulation Results

Behavioral simulation was performed to verify functional correctness. Figure 29 shows the simulation waveform.

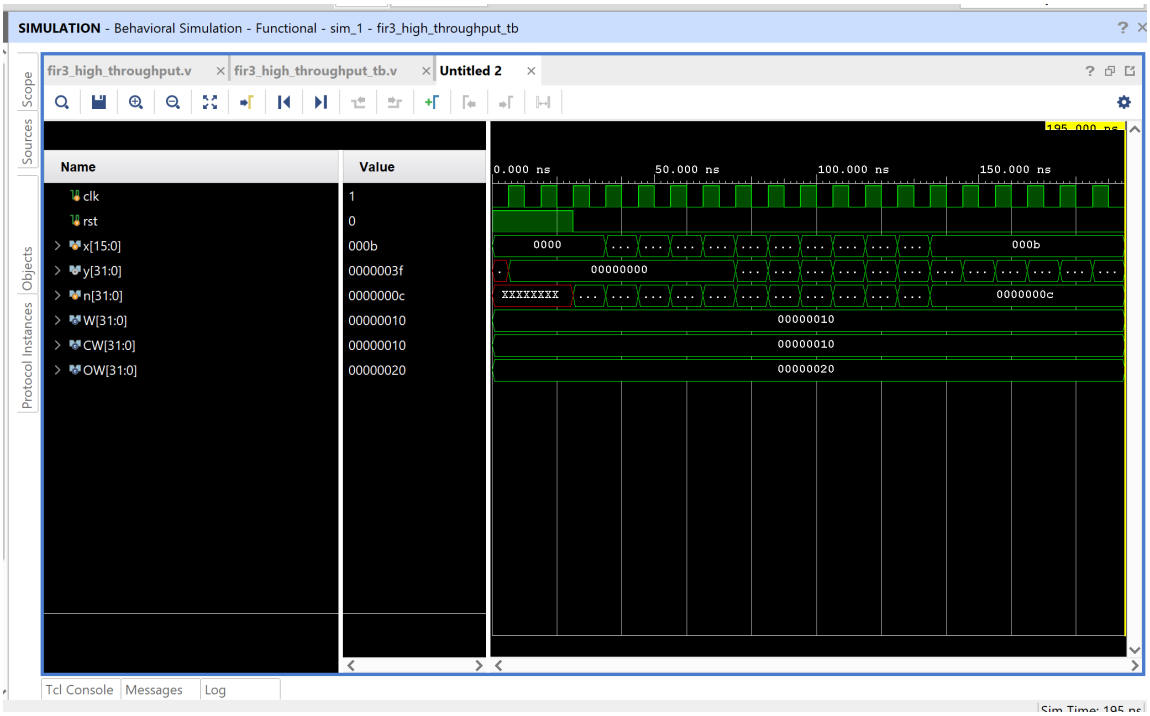


Figure 29: Behavioral simulation waveform of the pipelined FIR filter

From the waveform:

- Initial undefined values are observed before reset due to pipeline registers
- After reset deassertion, the pipeline gradually fills
- Once filled, the output produces one valid sample per clock cycle

Although the output is delayed by several clock cycles compared to the lowest-latency design, the functional behavior follows the FIR equation exactly, confirming correctness.

14.5 Resource Utilization

Figure 30 shows the post-synthesis resource utilization.

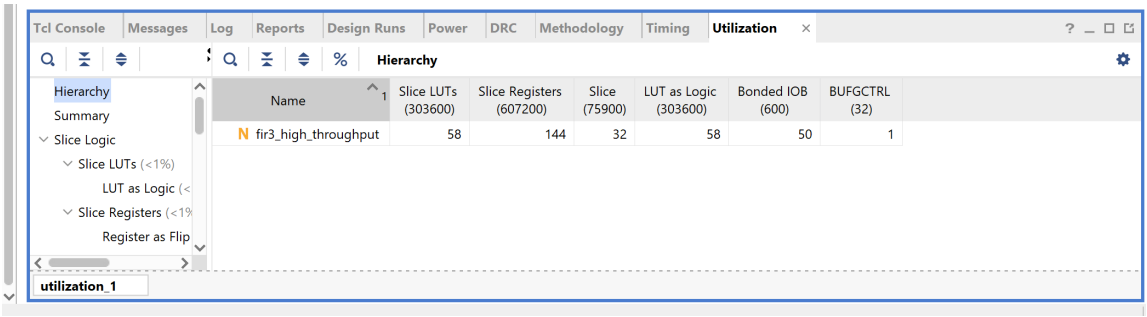


Figure 30: Resource utilization for the highest-throughput FIR filter

The design utilizes:

- 58 Slice LUTs
- 144 Slice Registers
- DSP blocks inferred for multipliers
- 1 BUFG

Compared to part (a), the number of registers is significantly higher due to pipeline insertion. This increase in resource usage is the trade-off required to achieve higher throughput.

14.6 Timing Analysis

A clock constraint of 10 was applied. Figure 32 shows the post-implementation timing summary.

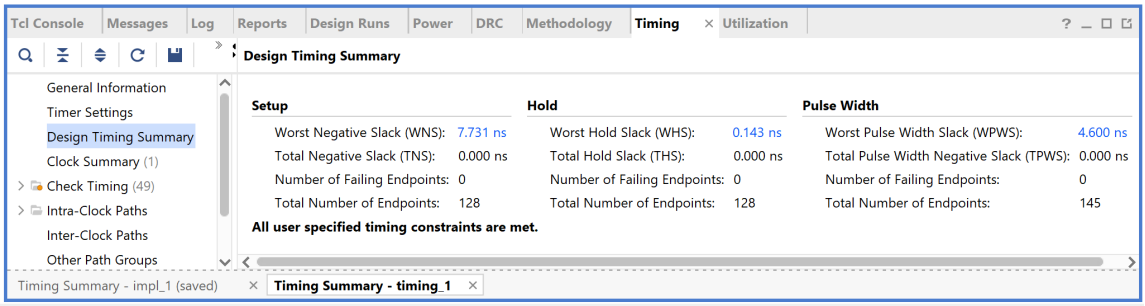


Figure 31: Post-implementation timing summary for highest-throughput design

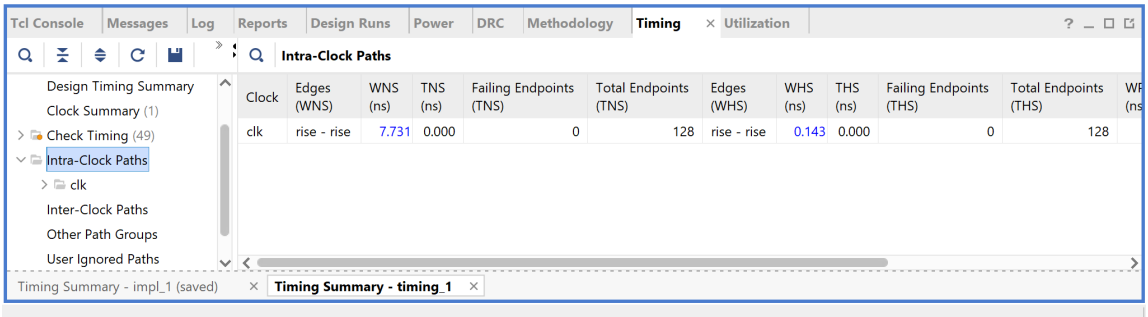


Figure 32: Post-implementation timing summary for highest-throughput design

The reported worst negative slack is:

$$\text{WNS} = +7.731 \text{ ns}$$

The minimum achievable clock period is therefore:

$$T_{\text{min}} = 10 - 7.731 = 2.269 \text{ ns}$$

The corresponding maximum operating frequency is:

$$F_{\text{max}} \approx 441 \text{ MHz}$$

This represents a substantial improvement compared to the lowest-latency design.

14.7 Critical Path Discussion

Due to pipelining, the critical path is reduced to:

- A register-to-register path containing at most one multiplier or one adder

This short critical path enables a much higher clock frequency, which directly translates to increased throughput.

14.8 Discussion

The highest-throughput FIR design successfully maximizes performance by introducing pipeline stages in the datapath. While the latency in clock cycles increases, the design can operate at a significantly higher clock frequency, making it suitable for high-speed signal processing applications.

The comparison between parts (a) and (b) highlights the fundamental trade-off between latency and throughput in FIR filter architectures.

15 (c) Lowest Area FIR Filter

15.1 Proposed Architecture

Figure 33 shows the hand-drawn architecture of the lowest-area FIR filter. In this design, hardware resources are minimized by sharing a single multiplier and a single adder across multiple clock cycles.

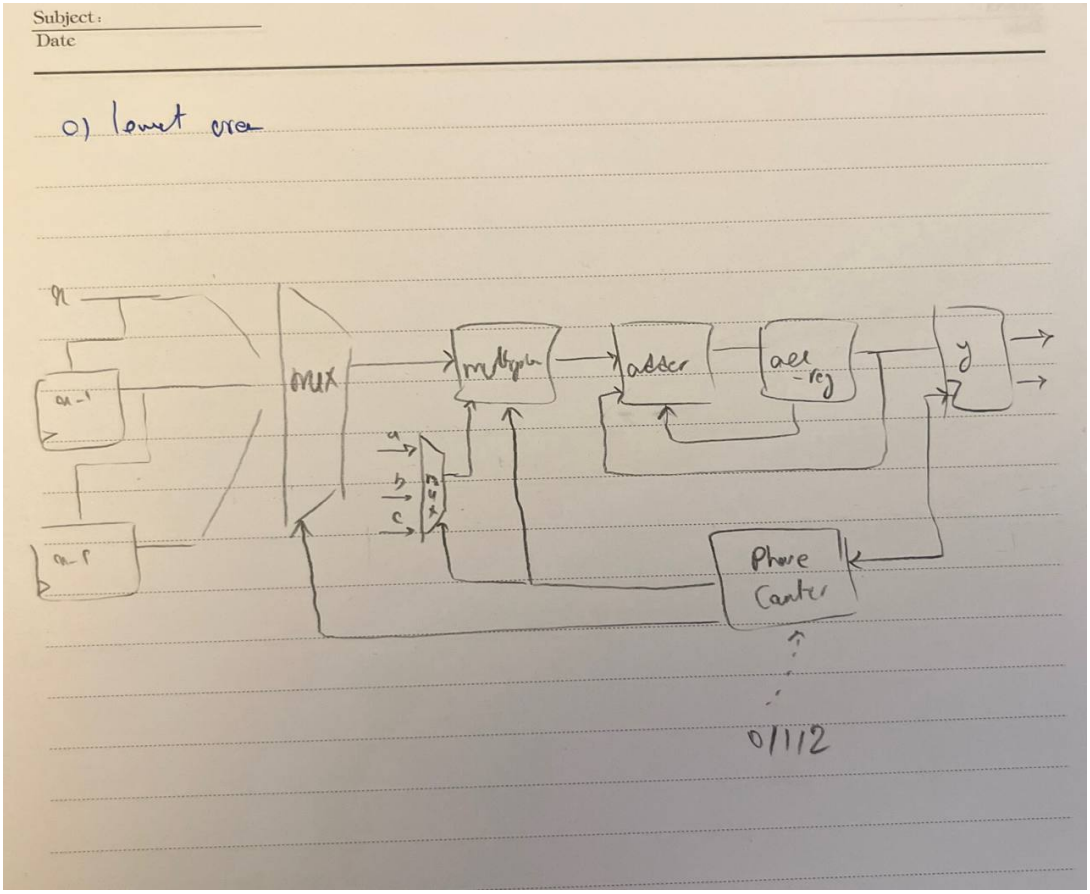


Figure 33: Hand-drawn resource-shared FIR filter architecture

A phase counter controls multiplexers that sequentially select the input samples $x(n)$, $x(n-1)$, and $x(n-2)$, along with their corresponding coefficients a, b, and c. The partial products are accumulated over three clock cycles, producing one output sample every three cycles.

15.2 RTL Implementation

The FIR filter was implemented using time-multiplexing and an accumulator-based architecture. Only one multiplier and one adder are instantiated, significantly reducing the overall hardware area. A small control unit (phase counter) orchestrates the computation across multiple cycles.

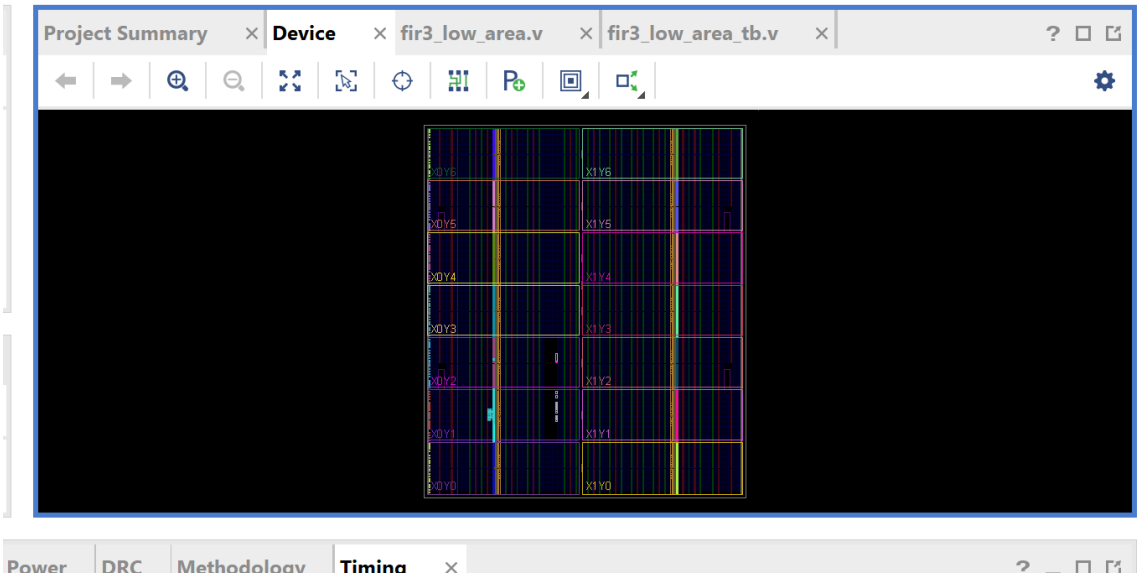


Figure 34: RTL Implementation of the lowest-area FIR filter

15.3 Simulation Results

Behavioral simulation was performed to verify correctness. Figure 35 shows the simulation waveform.

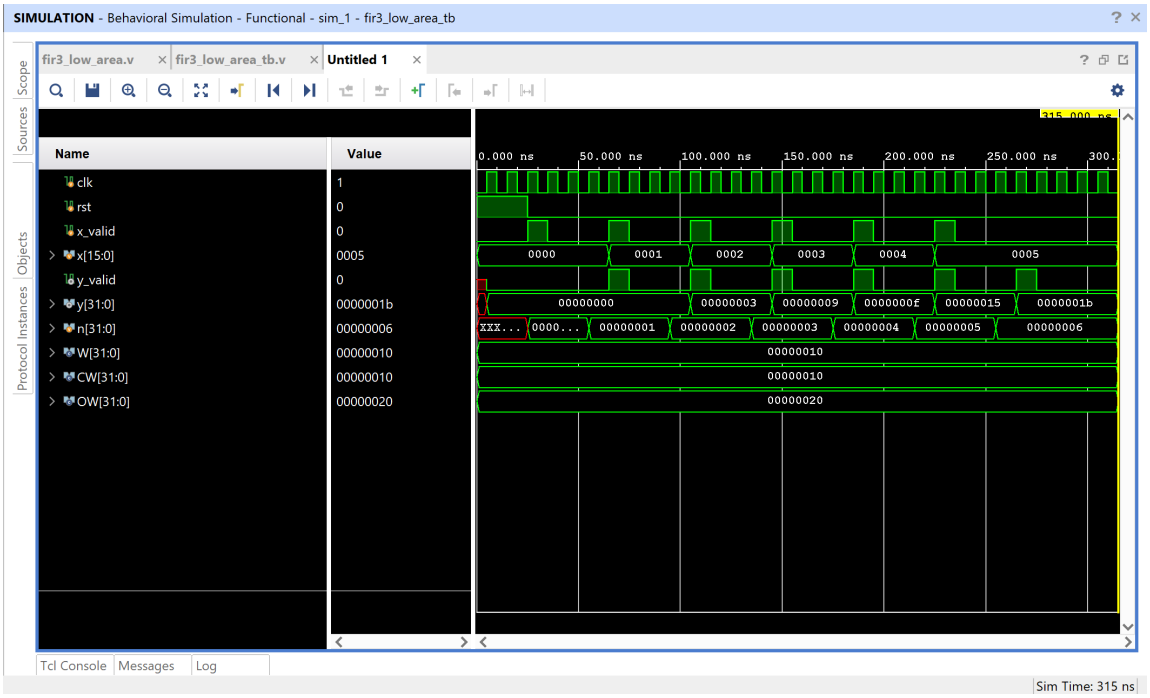


Figure 35: Behavioral simulation waveform of the lowest-area FIR filter

The waveform shows that:

- New inputs are accepted only when `x_valid` is asserted
- The output `y` is updated only when `y_valid` is asserted
- One valid output is produced every three clock cycles

This confirms correct functionality with reduced throughput due to resource sharing.

15.4 Resource Utilization

Figure 36 presents the post-synthesis resource utilization.

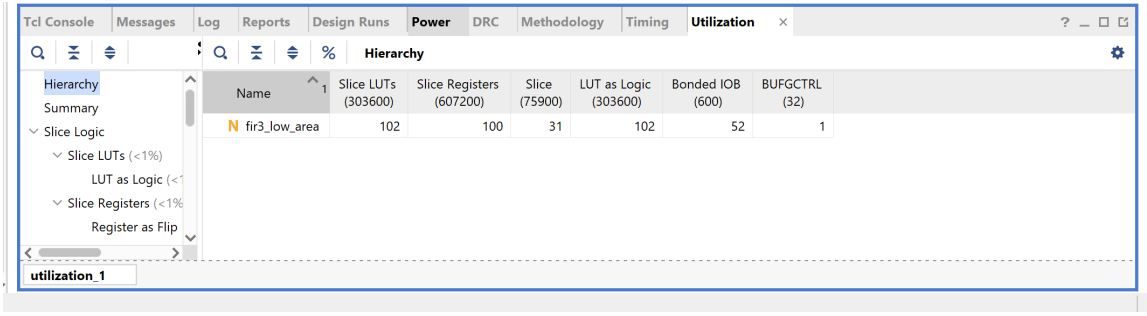


Figure 36: Resource utilization for the lowest-area FIR filter

The design uses:

- 102 Slice LUTs
- 100 Slice Registers
- 31 Slices
- A single shared multiplier

Compared to designs (a) and (b), this architecture minimizes arithmetic resources by reusing hardware across multiple clock cycles.

15.5 Timing Analysis

Timing analysis was performed with a 10 clock constraint. Figure 38 shows the post-implementation timing summary.

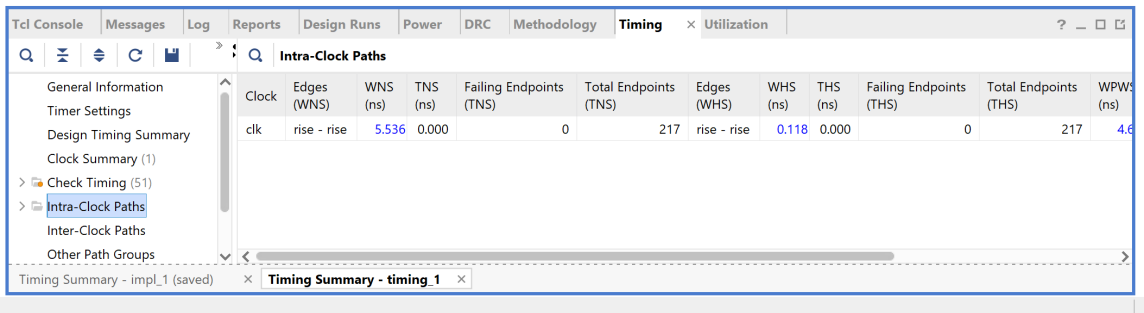


Figure 37: Timing summary for the lowest-area FIR filter

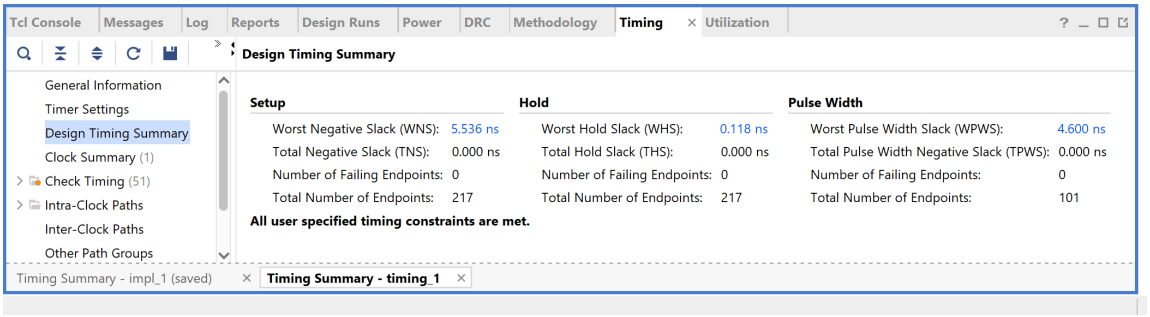


Figure 38: Timing summary for the lowest-area FIR filter

The reported worst negative slack is:

$$\text{WNS} = +5.536 \text{ ns}$$

The minimum clock period is:

$$T_{\text{min}} = 10 - 5.536 = 4.464 \text{ ns}$$

Resulting in a maximum operating frequency of:

$$F_{\text{max}} \approx 224 \text{ MHz}$$

15.6 Critical Path Discussion

The critical path consists of a register-to-register path containing a multiplexer, a single multiplier, and an adder. Due to the reduced combinational complexity per cycle, timing constraints are easily met despite the shared architecture.

15.7 Discussion

The lowest-area FIR filter design achieves minimal hardware usage by employing resource sharing and time-multiplexing. While throughput is reduced compared to the pipelined design, the architecture is highly area-efficient and suitable for resource-constrained applications.

This design illustrates the trade-off between hardware area and throughput in FIR filter implementations.