# Project 1: Building Blocks

## A Comprehensive Study of Fundamental Digital Circuits

**Abhishek Sharma**

**Abstract**

This document marks the beginning of my ambitious journey to design and implement 100 RTL projects, with a primary focus on SystemVerilog. Each project is meticulously crafted to enhance my understanding of digital circuit design, starting with fundamental building blocks. Through detailed analysis, RTL coding, testbench creation, and practical applications, this comprehensive study aims to establish a strong foundation for more complex digital systems and innovations.

# Contents

# 1 Project Overview

The "Building Blocks" project consists of fundamental digital circuits including basic logic gates, arithmetic circuits, multiplexers, encoders, decoders, and flip-flops. Each sub-project is documented with its design, simulation results, schematic, advantages, disadvantages, and applications.

# 2 Logic Gates

## 2.1 Description

Logic gates are the basic building blocks of digital circuits. In this section, we implement AND, OR, NOT, NAND, NOR, XOR, and XNOR gates using traditional methods and NAND gates. Using NAND gates is beneficial in processor design as they are universal gates, simplifying the design process.

## 2.2 RTL Code

Listing 1: Logic Gates RTL Code

```
module logic_gates (
    input logic A,
    input logic B,
    output logic AND_out,
    output logic OR_out,
    output logic NOT_out,
    output logic NAND_out,
    output logic NOR_out,
    output logic XOR_out,
    output logic XNOR_out
);
    // Traditional implementation
    assign AND_out = A & B;
    assign OR_out = A  B;
    assign NOT_out = ~A;
    assign NAND_out = ~(A & B);
    assign NOR_out = ~(A  B);
    assign XOR_out = A ^ B;
    assign XNOR_out = ~(A ^ B);
endmodule
```

## 2.3 Processor-Optimized RTL Code

Listing 2: Logic Gates using NAND Gates RTL Code

```
module logic_gates_nand (
    input logic A,
    input logic B,
    output logic AND_out,
    output logic OR_out,
    output logic NOT_out,
    output logic NAND_out,
    output logic NOR_out,
    output logic XOR_out,
    output logic XNOR_out
);
    // NAND gate
```

```
    assign NAND_out = ~(A & B);

    // NOT gate using NAND
    assign NOT_out = ~(A & A);

    // AND gate using NAND
    assign AND_out = ~(~(A & B) & ~(A & B));

    // OR gate using NAND
    assign OR_out = ~(~A & ~B);

      // XOR using NAND gates
    assign n1 = ~(A & B);
    assign n2 = ~(A & n1);
    assign n3 = ~(B & n1);
    assign XOR_out = ~(n2 & n3);

    // XNOR using NAND gates
    assign XNOR_out = ~(XOR_out);
endmodule
```

## 2.4 Testbench

Listing 3: Logic Gates Testbench

```
module test_logic_gates;
    logic A, B;
    logic AND_out, OR_out, NOT_out, NAND_out, NOR_out, XOR_out,
        XNOR_out;

    logic_gates uut (
        .A(A),
        .B(B),
        .AND_out(AND_out),
        .OR_out(OR_out),
        .NOT_out(NOT_out),
        .NAND_out(NAND_out),
        .NOR_out(NOR_out),
        .XOR_out(XOR_out),
        .XNOR_out(XNOR_out)
    );

    initial begin
        // Test vectors
        A = 0; B = 0;
        #10 A = 0; B = 1;
        #10 A = 1; B = 0;
        #10 A = 1; B = 1;
        #10 $stop;
    end
endmodule
```

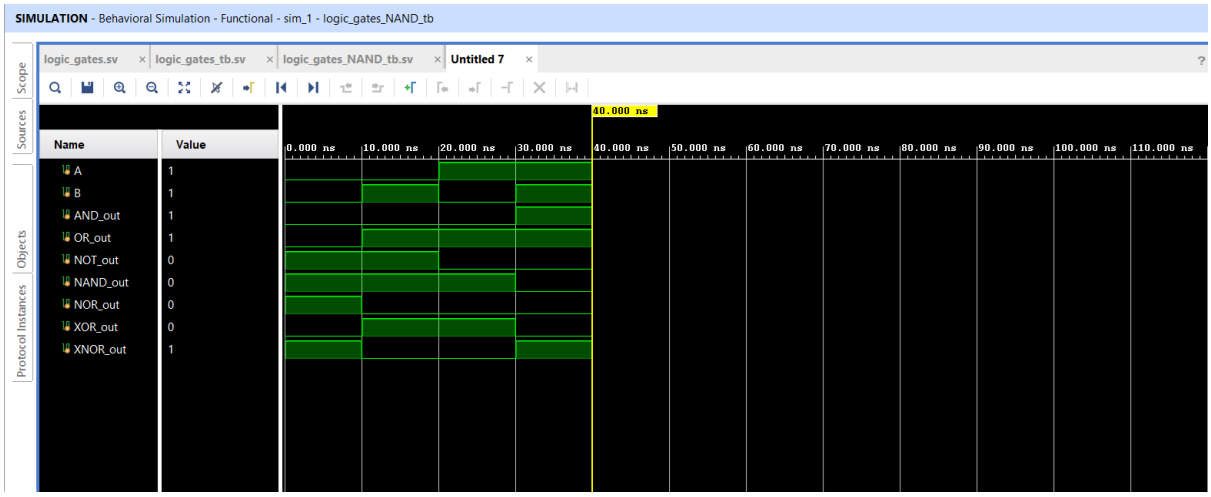


Figure 1: Simulation results of the logic gates

## 2.5 Simulation Results

## 2.6 Schematic

## 2.7 Advantages

- Simple design and easy to understand.
- Fundamental building blocks for more complex circuits.
- Using NAND gates makes the design flexible and universal for processors.

## 2.8 Disadvantages

- More gates are needed to implement complex functions.
- Increased power consumption and delay due to additional gates.

## 2.9 Applications

- Basic digital circuits and systems.
- Used in arithmetic circuits and data processing.

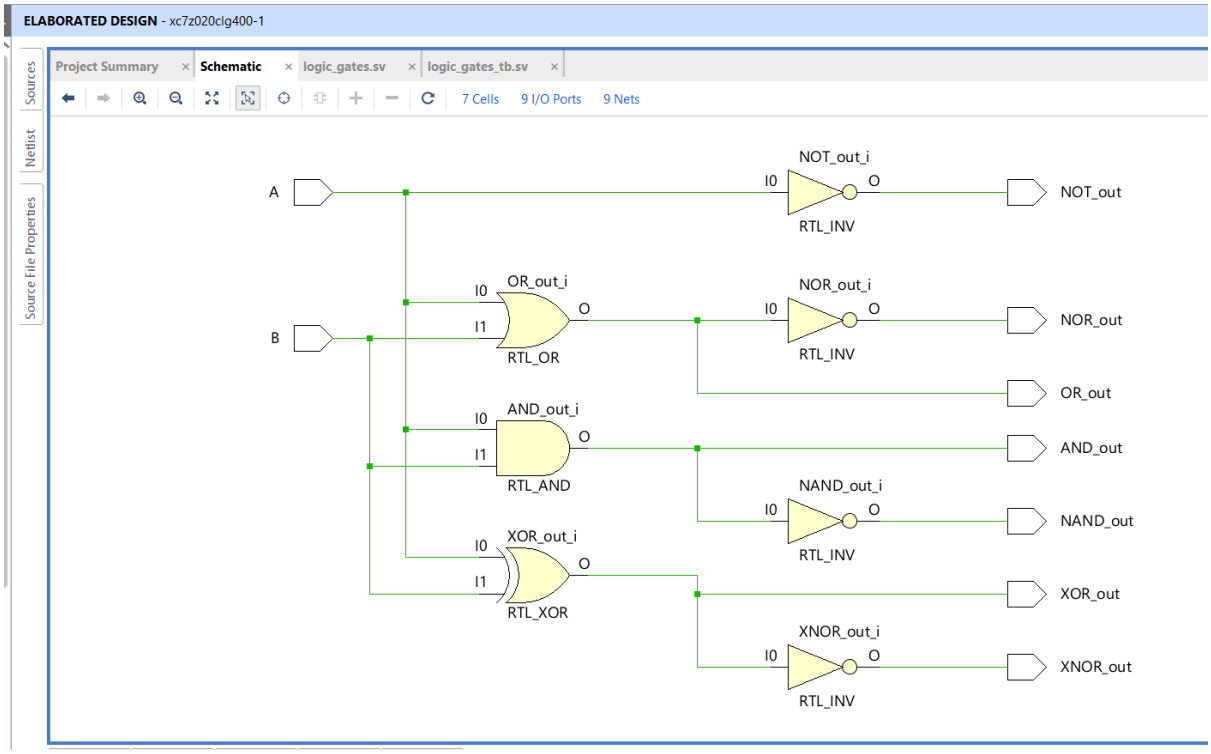


Figure 2: Schematic of the logic gates

# 3 Multiplexers

## 3.1 Description

Multiplexers are combinational circuits that select binary information from multiple input lines and direct it to a single output line.



## 3.2 RTL Code

Listing 4: 4-to-1 Multiplexer RTL Code

```
module multiplexer_4to1 (
    input logic [3:0] in,
    input logic [1:0] sel,
    output logic out
);
   assign out = (sel == 2'b00) ? in[0] :
                (sel == 2'b01) ? in[1] :
                (sel == 2'b10) ? in[2] :
                (sel == 2'b11) ? in[3] :
                1'b0; // Default case (optional, to handle invalid
                    'sel' values)
endmodule
```

## 3.3 Testbench

Listing 5: 4-to-1 Multiplexer Testbench

```
module test_multiplexer_4to1;
    logic [3:0] in;
```

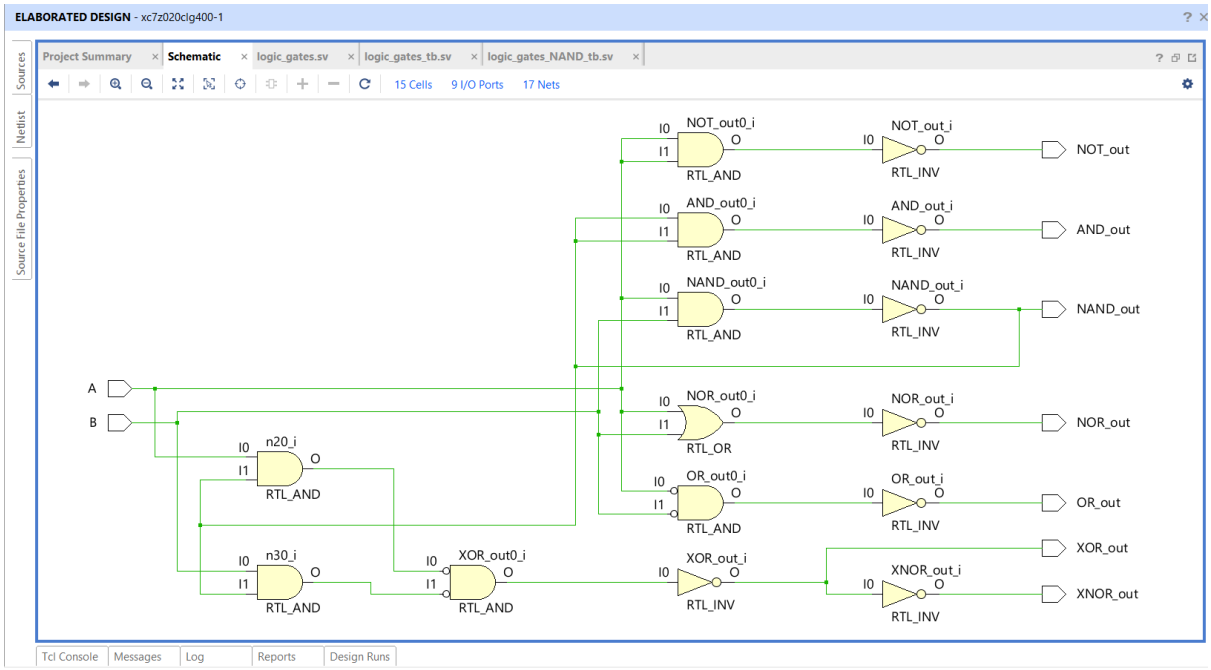


Figure 3: Schematic of the logic gates using NAND

```
    logic [1:0] sel;
    logic out;

    multiplexer_4to1 uut (
        .in(in),
        .sel(sel),
        .out(out)
    );

    initial begin
        // Test vectors
        in = 4'b1010;
        sel = 2'b00;
        #10 sel = 2'b01;
        #10 sel = 2'b10;
        #10 sel = 2'b11;
        #10 $stop;
    end
endmodule
```

## 3.4 Advantages

- Efficiently selects one of several input signals.
- Reduces the number of data lines needed.

## 3.5 Disadvantages

- Limited to the number of input lines it can handle.
- Increased complexity for larger multiplexers.

## 3.6 Applications

- Data routing.
- Signal multiplexing.
- Digital communication systems.

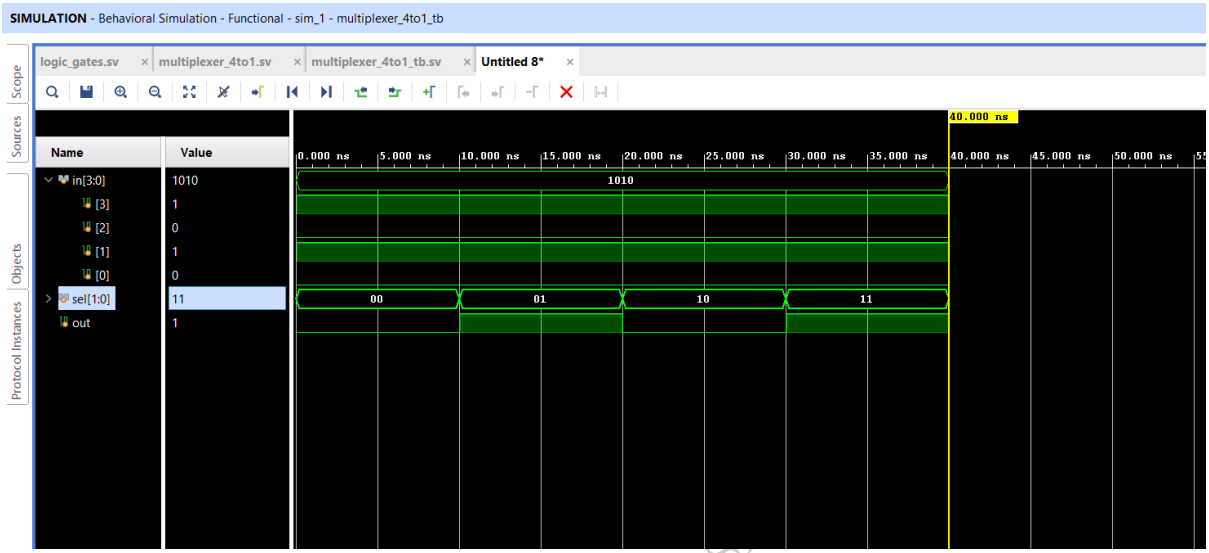


Figure 4: Simulation results of the 4-to-1 multiplexer

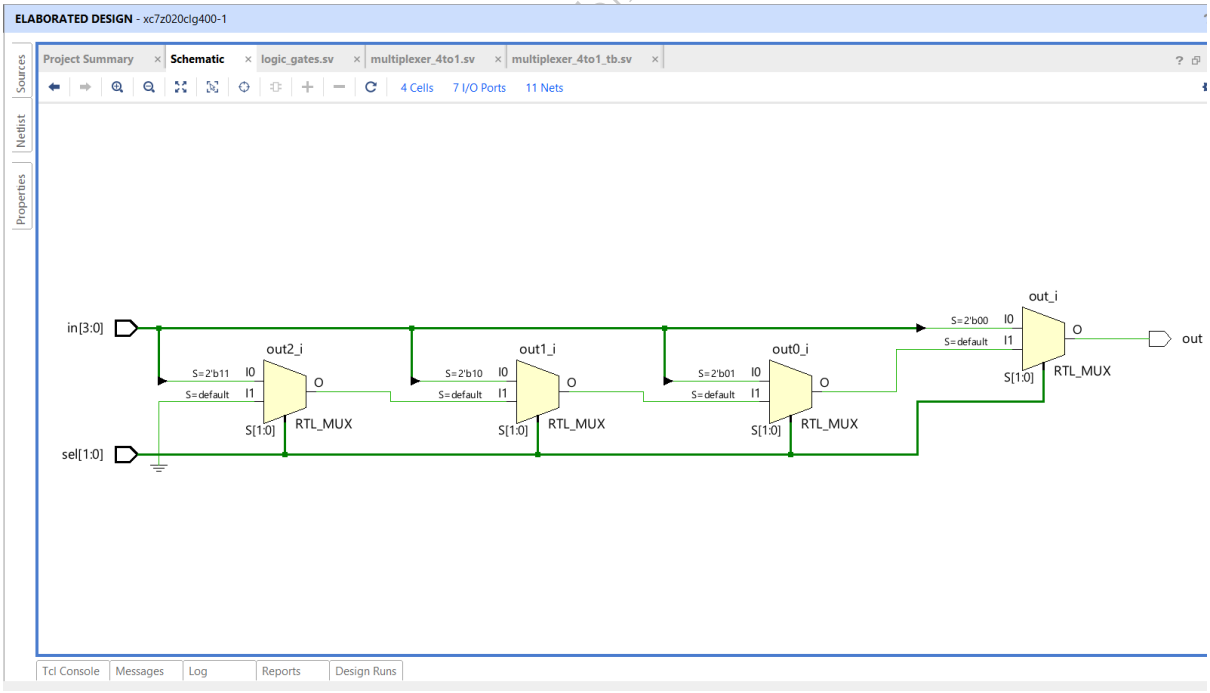


Figure 5: Schematic of the 4-to-1 multiplexer

# 4 Decoders

## 4.1 Description

Decoders are combinational circuits used for converting data between different forms. Encoders reduce the number of lines by converting them into a binary code, whereas decoders perform the reverse operation.

## 4.2 RTL Code

Listing 6: 3-to-8 Decoder RTL Code

```
module decoder_3to8 (
    input logic [2:0] in,
    output logic [7:0] out
);
    always_comb begin
        out = 8'b00000000;
        case (in)
            3'b000: out[0] = 1;
            3'b001: out[1] = 1;
            3'b010: out[2] = 1;
            3'b011: out[3] = 1;
            3'b100: out[4] = 1;
            3'b101: out[5] = 1;
            3'b110: out[6] = 1;
            3'b111: out[7] = 1;
        endcase
    end
endmodule
```



## 4.3 Testbench

Listing 7: 3-to-8 Decoder Testbench

```
module test_decoder_3to8;
    logic [2:0] in;
    logic [7:0] out;

    decoder_3to8 uut (
        .in(in),
        .out(out)
    );

    initial begin
        // Test vectors
        in = 3'b000;
        #10 in = 3'b001;
        #10 in = 3'b010;
        #10 in = 3'b011;
        #10 in = 3'b100;
        #10 in = 3'b101;
        #10 in = 3'b110;
        #10 in = 3'b111;
        #10 $stop;
    end
endmodule
```

## 4.4 Simulation Results

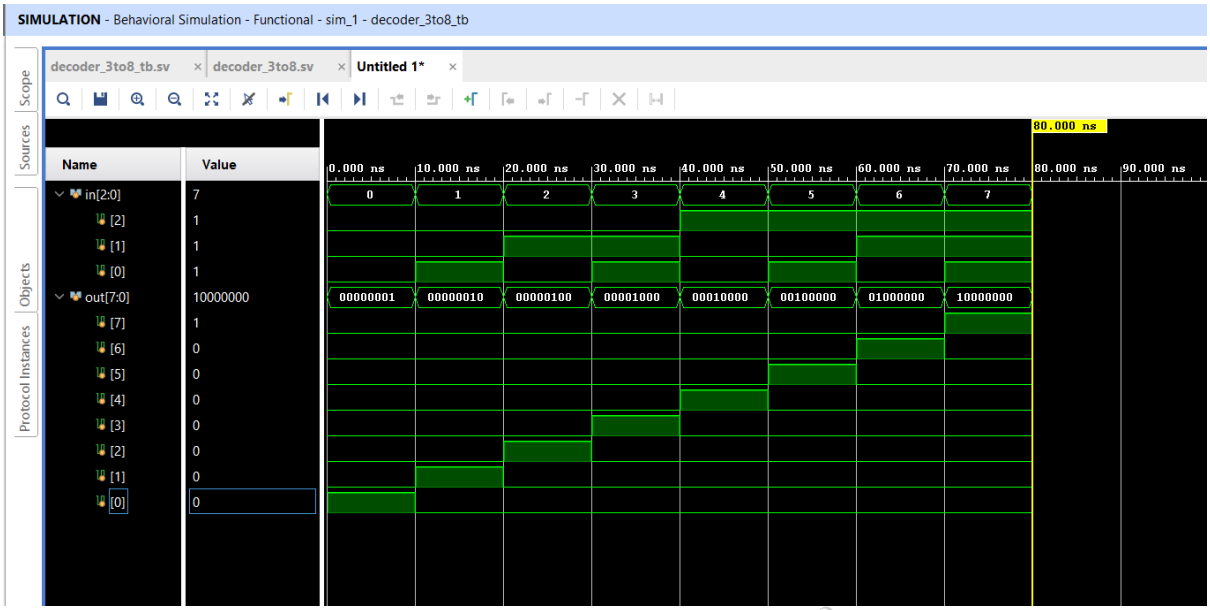


Figure 6: Simulation results of the 3-to-8 decoder

## 4.5 Schematic

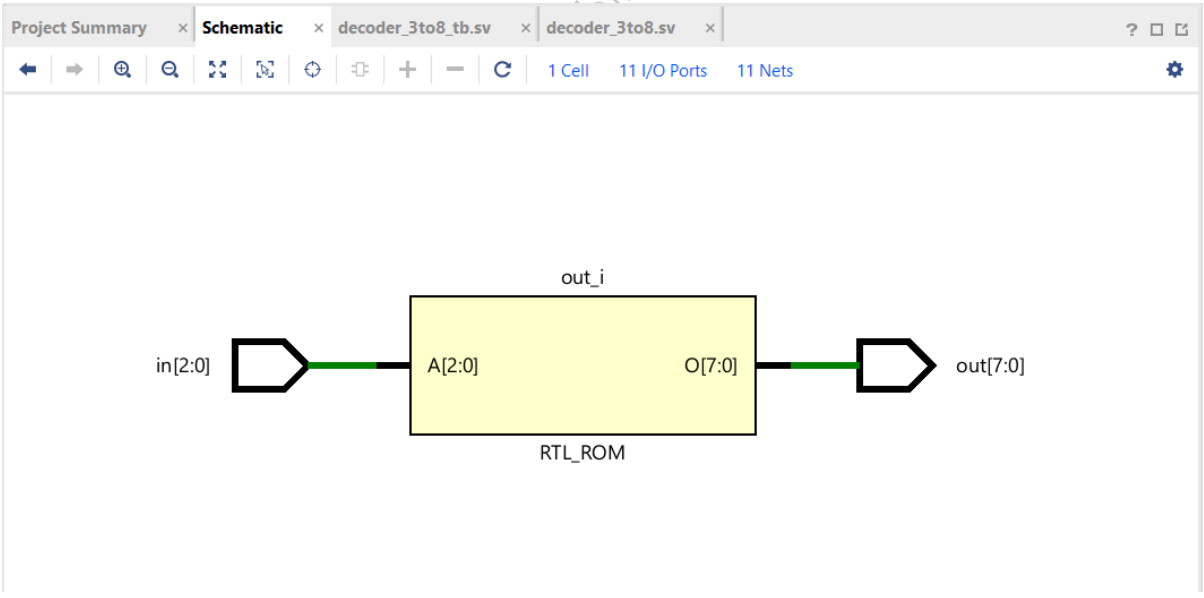


Figure 7: Schematic of the 3-to-8 decoder

## 4.6 Advantages

- Simplifies the selection of specific lines from many.
- Converts binary data to one-hot encoding.

## 4.7 Disadvantages

- Complexity increases with the number of input lines.
- Power consumption and delay can be higher for larger decoders.

## 4.8 Applications

- Memory address decoding.
- Data multiplexing and demultiplexing.
- Binary to one-hot conversion.

# 5 Encoders

## 5.1 Description

Encoders and decoders are combinational circuits used for converting data between different forms. Encoders reduce the number of lines by converting them into a binary code, whereas decoders perform the reverse operation.

## 5.2 RTL Code

Listing 8: 8-to-3 Encoder RTL Code

```
module encoder_8_to_3 (
    input logic [7:0] in, // 8 input lines
    output logic [2:0] out // 3 output lines
);
    always_comb begin
        case (in)
            8'b00000001: out = 3'b000;
            8'b00000010: out = 3'b001;
            8'b00000100: out = 3'b010;
            8'b00001000: out = 3'b011;
            8'b00010000: out = 3'b100;
            8'b00100000: out = 3'b101;
            8'b01000000: out = 3'b110;
            8'b10000000: out = 3'b111;
            default: out = 3'bxxx; // Invalid input
        endcase
    end
endmodule
```



## 5.3 Testbench

Listing 9: 8-to-3 Encoder Testbench

```
module test_encoder_8_to_3;
    logic [7:0] in;
    logic [2:0] out;

    encoder_8_to_3 uut (
        .in(in),
        .out(out)
    );

    initial begin
        // Test all possible inputs
        in = 8'b00000001; #10;
        $display("Input: %b, Output: %b", in, out);

        in = 8'b00000010; #10;
        $display("Input: %b, Output: %b", in, out);

        in = 8'b00000100; #10;
        $display("Input: %b, Output: %b", in, out);

        in = 8'b00001000; #10;
        $display("Input: %b, Output: %b", in, out);
```

```
23
24          in = 8'b00010000; #10;
25          $display("Input: %b, Output: %b", in, out);
26
27          in = 8'b00100000; #10;
28          $display("Input: %b, Output: %b", in, out);
29
30          in = 8'b01000000; #10;
31          $display("Input: %b, Output: %b", in, out);
32
33          in = 8'b10000000; #10;
34          $display("Input: %b, Output: %b", in, out);
35
36          $stop;
37      end
38  endmodule
```

## 5.4 Simulation Results


Figure 8: Simulation results of the 8-to-3 encoder

## 5.5 Schematic

## 5.6 Advantages

- Reduces the number of data lines required for communication.
- Simplifies the design of digital circuits by converting multiple input lines into a smaller number of output lines.

## 5.7 Disadvantages

- Limited to one active input at a time; if multiple inputs are active, it results in invalid output.
- Requires additional logic to handle invalid states or multiple active inputs.

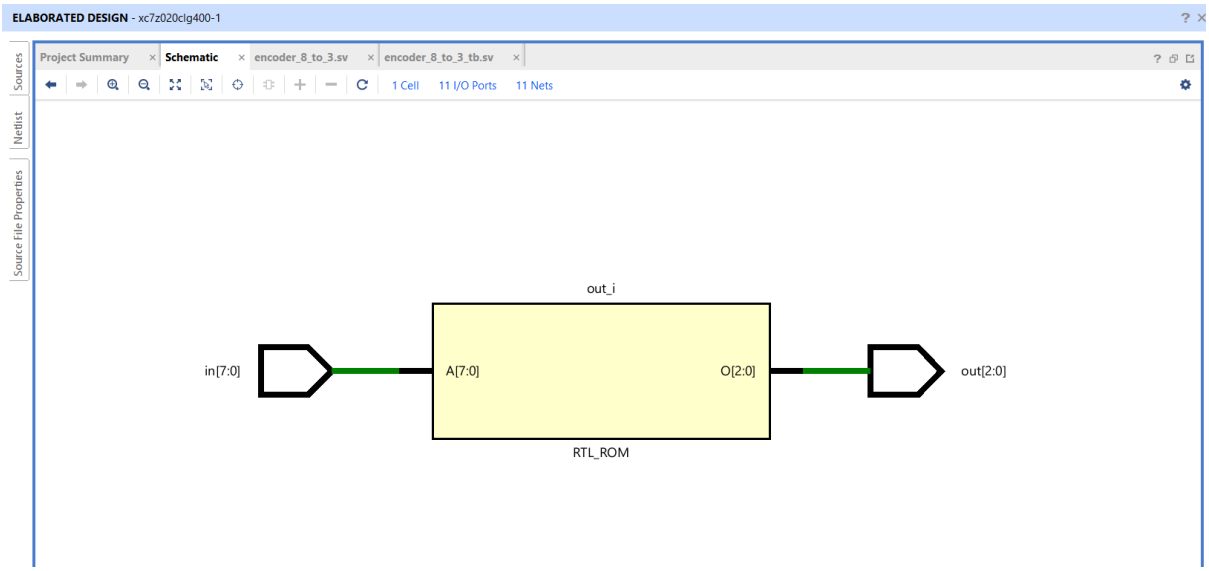


Figure 9: Schematic of the 8-to-3 encoder

## 5.8 Applications

- Data compression and transmission.
- Multiplexing data from multiple sources.
- Address decoding in memory systems.



# 6 D Flip-Flops

## 6.1 Description

Flip-flops are fundamental building blocks for sequential circuits. They are used for storing binary data and for synchronization purposes in digital systems.

## 6.2 RTL Code

Listing 10: D Flip-Flop RTL Code

```
module d_flip_flop (
    input logic D,
    input logic clk,
    output logic Q
);
    always_ff @(posedge clk) begin
        Q <= D;
    end
endmodule
```

## 6.3 Testbench

Listing 11: D Flip-Flop Testbench

```
module test_d_flip_flop;
    logic D, clk;
    logic Q;
```

```
4
5       d_flip_flop uut (
6           .D(D),
7           .clk(clk),
8           .Q(Q)
9       );
10
11      initial begin
12          // Initialize clock
13          clk = 0;
14          forever #5 clk = ~clk;
15      end
16
17      initial begin
18          // Test vectors
19          D = 0;
20          #10 D = 1;
21          #10 D = 0;
22          #10 D = 1;
23          #10 $stop;
24      end
25  endmodule
```

## 6.4 Simulation Results

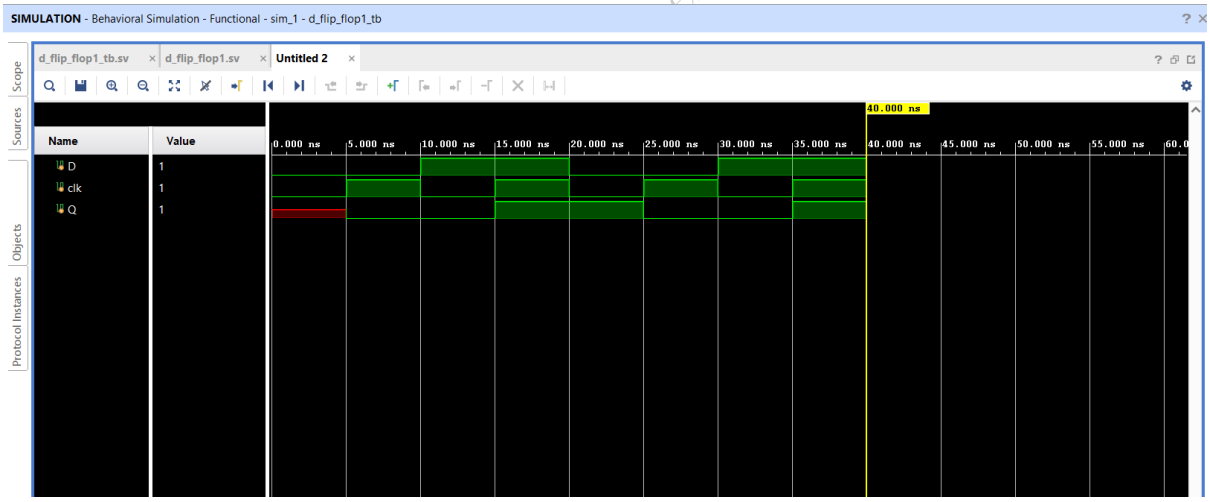


Figure 10: Simulation results of the D flip-flop

The "don't care" or "X" state in the first cycle of my simulation is because the output `Q` of my `d_flip_flop` module is not explicitly initialized to a known state before the first clock edge occurs. In digital simulations, registers (such as `Q`) start in an unknown state unless they are explicitly initialized.

To ensure that Q starts in a known state, we can modify the testbench to include an explicit reset or initialization step. Explicit Initialization: Adding Q = 0; ensures that Q is set to a known state (0 in this case) before the clock starts toggling. This prevents the initial "don't care" state.

Simulation Behavior: With this initialization, the simulation will now have Q starting from 0. Subsequent clock edges will update Q based on the value of D.

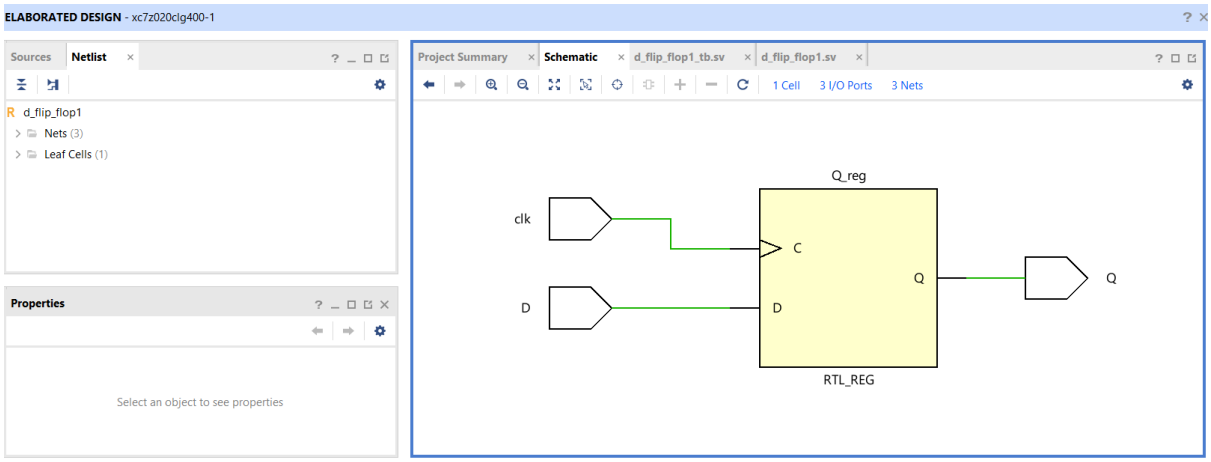


Figure 11: Schematic of the D flip-flop

## 6.5 Schematic

## 6.6 Advantages

- Simple and efficient data storage.
- Used in shift registers, counters, and memory devices.

## 6.7 Disadvantages

- Limited to storing one bit of data.
- Power consumption and delay increase with additional flip-flops.



## 6.8 Applications

- Data storage and transfer.
- Synchronization in digital systems.
- Timing and control circuits.

# 7 T Flip-Flop

### 7.0.1 Description

A T flip-flop toggles its output on each rising edge of the clock if the T input is high.

### 7.0.2 RTL Code

Listing 12: T Flip-Flop RTL Code

```
module t_flip_flop(
input logic T,
    input logic clk,
    input logic rst,  // Add reset signal
    output logic Q
    );
      always_ff @(posedge clk or posedge rst) begin
        if (rst)
            Q <= 0;  // Initialize Q to 0 on reset
```

```
        else if (T)
            Q <= ~Q;
    end

endmodule
```

#### 7.0.3 Testbench

Listing 13: T Flip-Flop Testbench

```
module t_flip_flop_tb;
    logic T, clk, rst;
    logic Q;
     t_flip_flop uut (
        .T(T),
        .clk(clk),
        .rst(rst),  // Connect reset signal
        .Q(Q)
    );

 initial begin
        // Initialize clock
        clk = 0;
        forever #5 clk = ~clk;
    end

    initial begin
        // Initialize reset and apply it initially
        rst = 1;
        #10 rst = 0;

        // Test vectors
        T = 0;
        #10 T = 1;
        #10 T = 0;
        #10 T = 1;
        #10 $stop;
    end
endmodule
```

#### 7.0.4 Simulation Results

#### 7.0.5 Schematic

#### 7.0.6 Advantages

- Simplifies the design of counters.
- Can be used in frequency division applications.

#### 7.0.7 Disadvantages

- May require additional logic to handle reset conditions.

#### 7.0.8 Applications

- Counters.
- Frequency dividers.

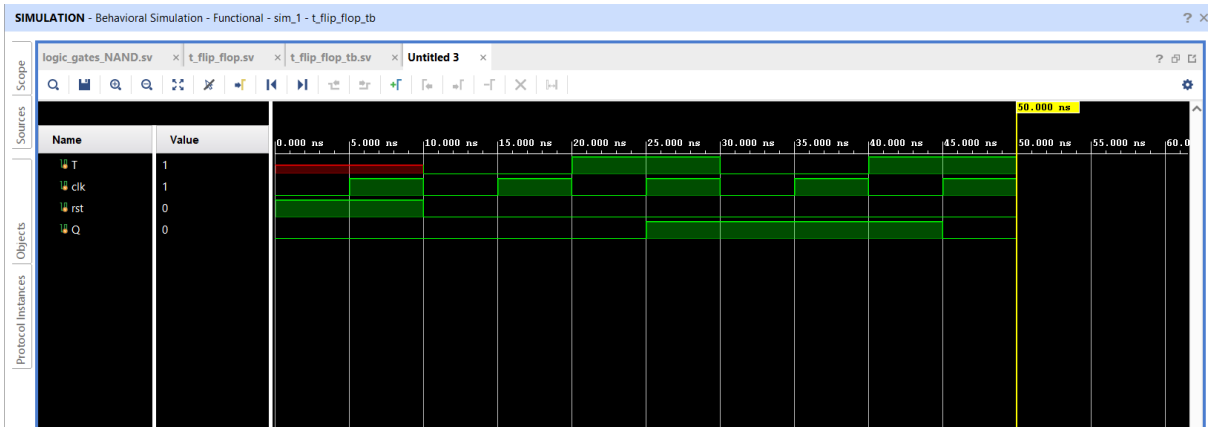


Figure 12: Simulation results of the T flip-flop

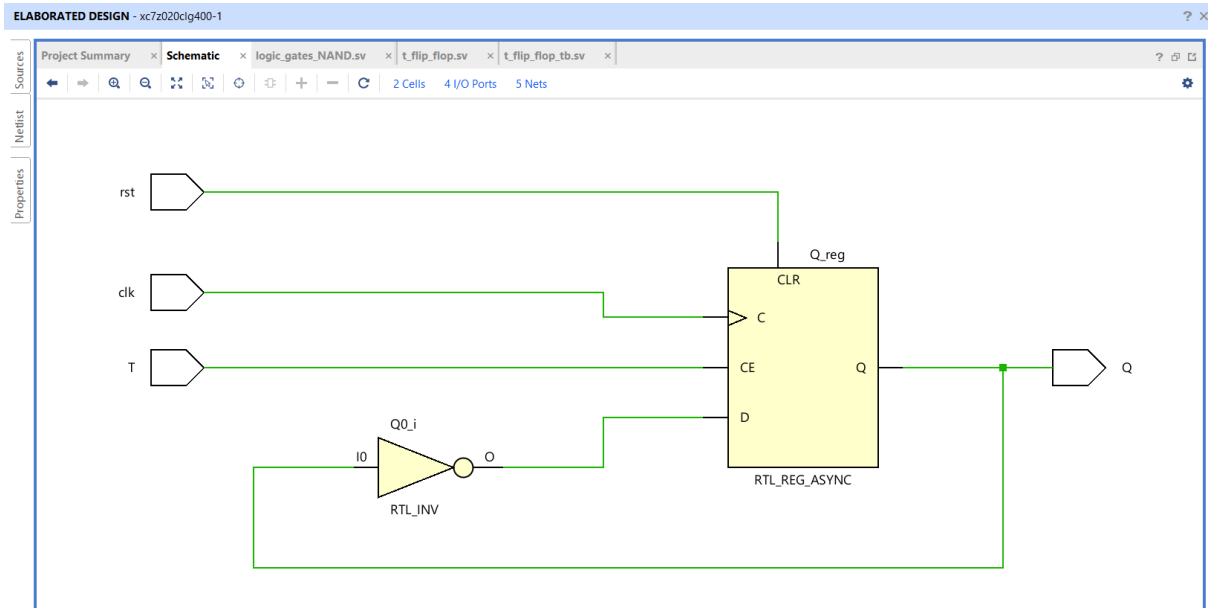


Figure 13: Schematic of the T flip-flop

# 8 JK Flip-Flop

### 8.0.1 Description

A JK flip-flop operates based on the combination of J and K inputs, allowing set, reset, and toggle operations.

### 8.0.2 RTL Code

Listing 14: JK Flip-Flop RTL Code

```
module jk_flip_flop (
    input logic J,
    input logic K,
    input logic clk,
    output logic Q
);
    always_ff @(posedge clk) begin
        case ({J, K})
            2'b00: Q <= Q;          // No change
```

```
            2'b01: Q <= 0;          // Reset
            2'b10: Q <= 1;          // Set
            2'b11: Q <= ~Q;         // Toggle
        endcase
    end
endmodule
```

### 8.0.3 Testbench

Listing 15: JK Flip-Flop Testbench

```
module test_jk_flip_flop;
    logic J, K, clk;
    logic Q;

    jk_flip_flop uut (
        .J(J),
        .K(K),
        .clk(clk),
        .Q(Q)
    );

    initial begin
        // Initialize clock
        clk = 0;
        forever #5 clk = ~clk;
    end

    initial begin
        // Test vectors
        J = 0; K = 0;
        #10 J = 1; K = 0;
        #10 J = 0; K = 1;
        #10 J = 1; K = 1;
        #10 $stop;
    end
endmodule
```

### 8.0.4 Simulation Results

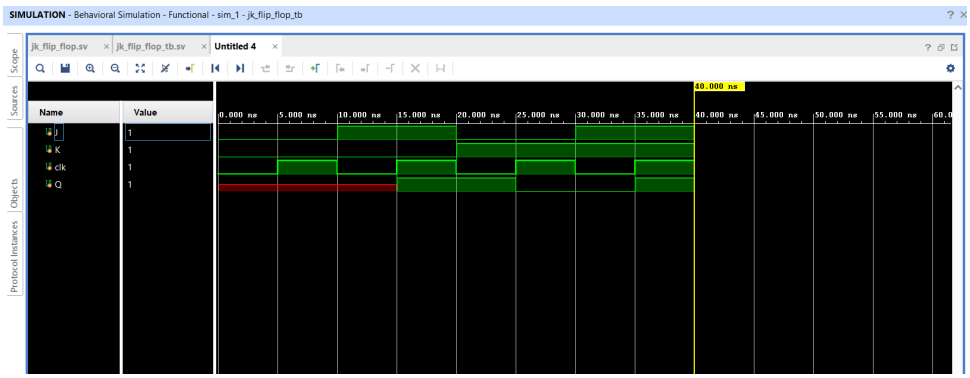


Figure 14: Simulation results of the JK flip-flop

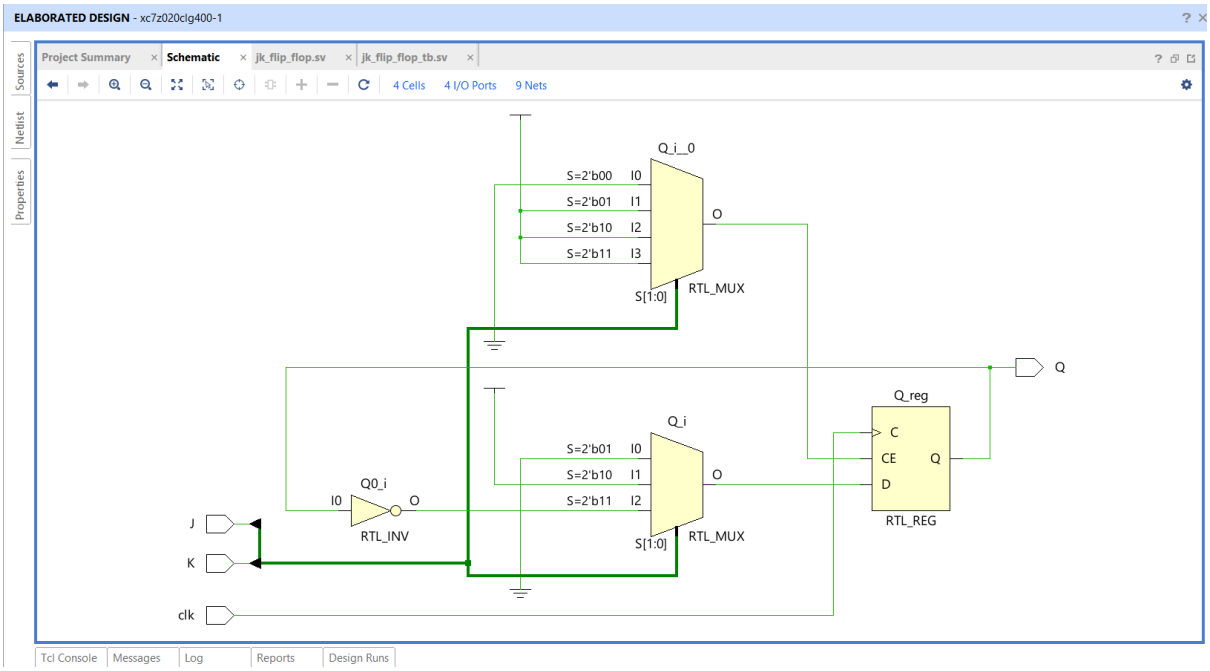


Figure 15: Schematic of the JK flip-flop

#### 8.0.5 Schematic

#### 8.0.6 Advantages

- Versatile and can perform set, reset, and toggle operations.
- Can be used in a wide range of applications due to its flexibility.



#### 8.0.7 Disadvantages

- More complex than D and T flip-flops.
- May require additional circuitry to prevent invalid states.

#### 8.0.8 Applications

- Counters.
- Shift registers.
- Control circuits.

# 9 SR Flip-Flop

#### 9.0.1 Description

An SR flip-flop sets or resets its output based on the set (S) and reset (R) inputs.

#### 9.0.2 RTL Code

Listing 16: SR Flip-Flop RTL Code

```
module sr_flip_flop (
    input logic S,
    input logic R,
    input logic clk,
```

```
    output logic Q
);
    always_ff @(posedge clk) begin
        if (S && ~R)
            Q <= 1;                 // Set
        else if (~S && R)
            Q <= 0;                 // Reset
        else if (S && R)
            Q <= 1'bx;              // Invalid state
    end
endmodule
```

#### 9.0.3 Testbench

Listing 17: SR Flip-Flop Testbench

```
module test_sr_flip_flop;
    logic S, R, clk;
    logic Q;

    sr_flip_flop uut (
        .S(S),
        .R(R),
        .clk(clk),
        .Q(Q)
    );

    initial begin
        // Initialize clock
        clk = 0;
        forever #5 clk = ~clk;
    end

    initial begin
        // Test vectors
        S = 0; R = 0;
        #10 S = 1; R = 0;
        #10 S = 0; R = 1;
        #10 S = 1; R = 1;
        #10 $stop;
    end
endmodule
```

#### 9.0.4 Simulation Results

The "don't care" or "X" state for `Q` when `S = 0` and `R = 0`, as well as when `S = 1` and `R = 1`, is because these conditions are not well-defined for a basic SR flip-flop. In a typical SR flip-flop:

- When `S = 0` and `R = 0`, `Q` retains its previous state.
- When `S = 1` and `R = 0`, `Q` is set to 1.
- When `S = 0` and `R = 1`, `Q` is reset to 0.
- When `S = 1` and `R = 1`, `Q` is set to an invalid state, which we can define as a specific state if needed (e.g., setting it to `'x`).

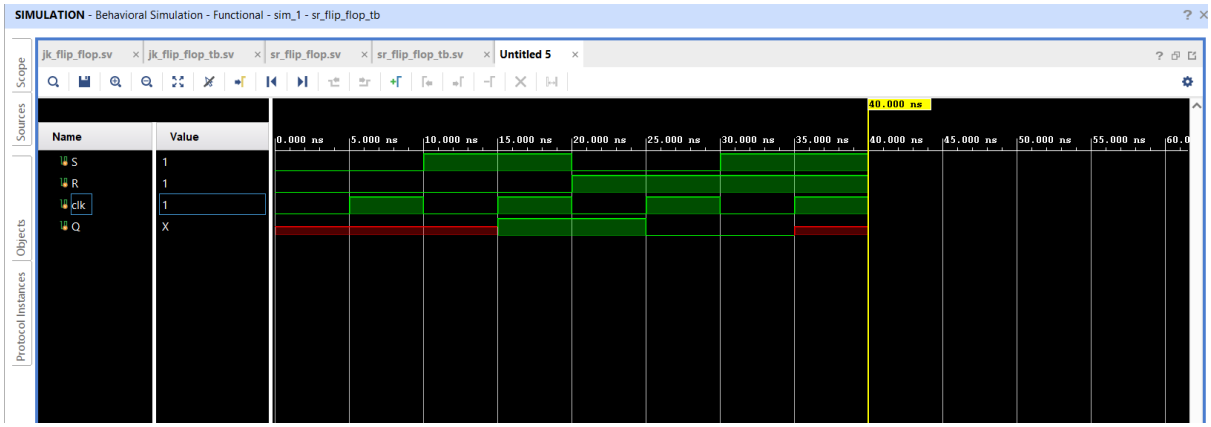


Figure 16: Simulation results of the SR flip-flop

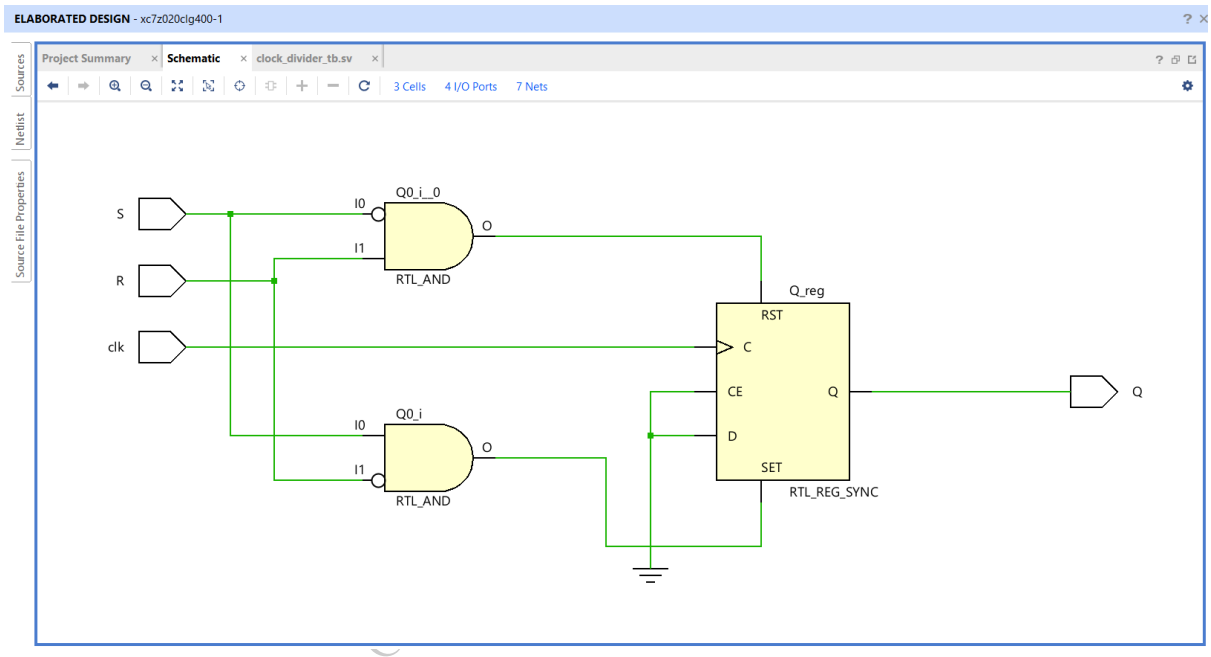


Figure 17: Schematic of the SR flip-flop

### 9.0.5 Schematic

### 9.0.6 Advantages

- Simple design and easy to understand.
- Useful for basic set and reset operations.

### 9.0.7 Disadvantages

- Invalid state when both set and reset inputs are high.
- Limited flexibility compared to other flip-flops.

### 9.0.8 Applications

- Simple control circuits.
- Basic memory elements.

# 10 Master-Slave JK Flip-Flop

## 10.1 Description

The Master-Slave JK flip-flop is an improved version of the standard JK flip-flop. It is designed to eliminate problems such as glitches and race conditions that can occur in the standard JK flip-flop. The master-slave configuration consists of two flip-flops connected in series: the master flip-flop captures the input state on the rising edge of the clock, and the slave flip-flop captures the output of the master on the falling edge of the clock. This arrangement ensures that the flip-flop changes state only once per clock cycle and is less prone to timing issues.

## 10.2 RTL Code

Listing 18: Master-Slave JK Flip-Flop RTL Code

```
module master_slave_jk_flip_flop (
    input logic J,
    input logic K,
    input logic clk,
    input logic reset,  // Optional: for synchronous reset
    output logic Q
);
    logic Q_master, Q_slave;

    // Master flip-flop
    always_ff @(posedge clk or posedge reset) begin
        if (reset)
            Q_master <= 0;  // Reset master Q to 0
        else if (J && ~K)
            Q_master <= 1;  // Set
        else if (~J && K)
            Q_master <= 0;  // Reset
        else if (J && K)
            Q_master <= ~Q_master;  // Toggle
    end

    // Slave flip-flop
    always_ff @(negedge clk or posedge reset) begin
        if (reset)
            Q_slave <= 0;  // Reset slave Q to 0
        else
            Q_slave <= Q_master;  // Capture output from master
    end

    assign Q = Q_slave;
endmodule
```

## 10.3 Testbench

Listing 19: Master-Slave JK Flip-Flop Testbench

```
module test_master_slave_jk_flip_flop;
    logic J, K, clk, reset;
    logic Q;

    master_slave_jk_flip_flop uut (
        .J(J),
```

```
        .K(K),
        .clk(clk),
        .reset(reset),
        .Q(Q)
    );

    // Generate clock signal
    initial begin
        clk = 0;
        forever #5 clk = ~clk; // 10ns clock period
    end

    // Testbench procedure
    initial begin
        // Initialize inputs
        J = 0; K = 0;
        reset = 1;
        #10 reset = 0;  // Release reset after 10ns

        // Apply test vectors
        J = 1; K = 0;  // Set state
        #20;
        J = 0; K = 1;  // Reset state
        #20;
        J = 1; K = 1;  // Toggle state
        #20;

        $stop;  // Stop simulation
    end
endmodule
```

## 10.4 Simulation Results

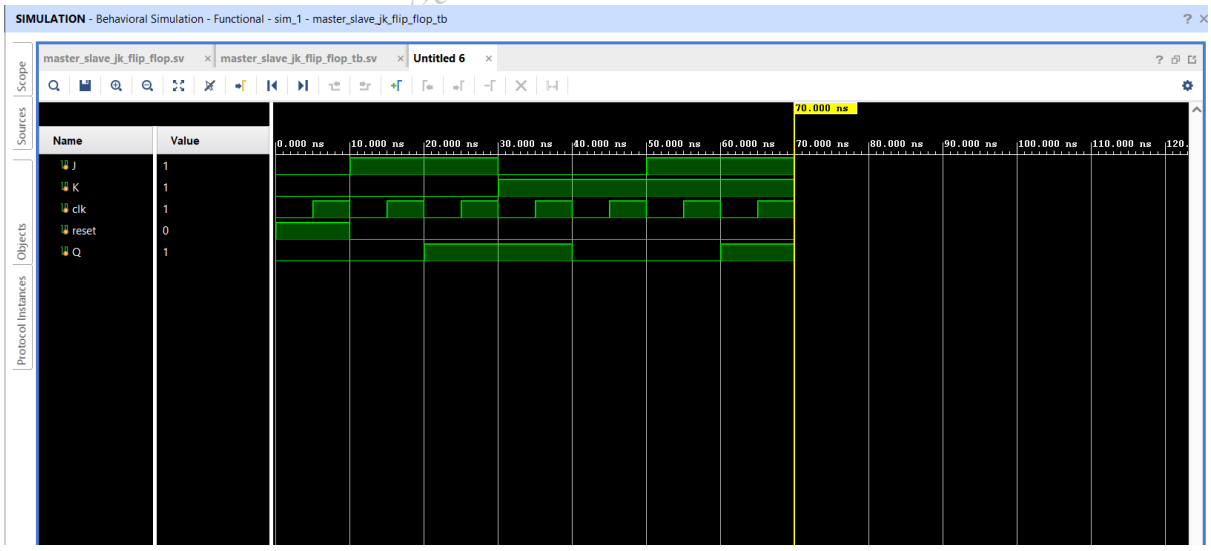


Figure 18: Simulation results of the Master-Slave JK flip-flop

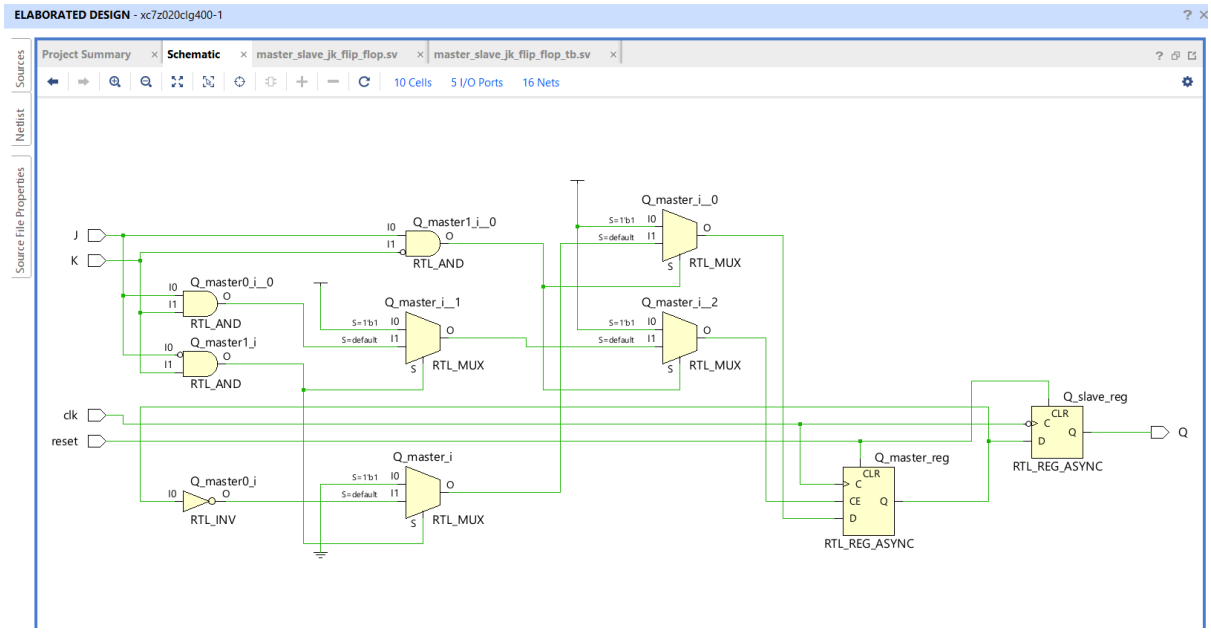


Figure 19: Schematic of the Master-Slave JK flip-flop

## 10.5 Schematic

## 10.6 Advantages

- Eliminates Glitches: The master-slave configuration prevents glitches that can occur in standard JK flip-flops due to simultaneous changes in J and K inputs.
- Reduces Race Conditions: By using separate master and slave stages, the master-slave JK flip-flop avoids race conditions that occur in a single JK flip-flop.
- Stable Output: The output changes only once per clock cycle, ensuring stable operation.



## 10.7 Disadvantages

- Complex Design: The master-slave configuration adds complexity compared to a standard JK flip-flop.
- Increased Propagation Delay: The additional stage can increase the propagation delay compared to simpler flip-flops.

## 10.8 Problems with Standard Flip-Flops

- JK Flip-Flop:
  - Glitches: Unpredictable changes in output when both J and K inputs are active during a clock transition.
  - Race Conditions: Occur when timing issues cause the output to become unstable or incorrect.
- D Flip-Flop:
  - Metastability: Can occur if the data input changes close to the clock edge, leading to unstable output.
- T Flip-Flop:
  - Glitches: Similar to the JK flip-flop, toggling can cause glitches if the T input changes around the clock edge.

- SR Flip-Flop:
  - Invalid State: When both set and reset inputs are high, the output becomes indeterminate, leading to unpredictable behavior.

# 11 Counter

## 11.1 Description

Counters are sequential circuits used to count the number of occurrences of an event. They can be used for various applications, including event counting and frequency division.

## 11.2 RTL Code

Listing 20: 4-bit Counter RTL Code

```
module counter_4bit (
    input logic clk,
    input logic reset,
    output logic [3:0] count
);
    always_ff @(posedge clk or posedge reset) begin
        if (reset)
            count <= 4'b0000;
        else
            count <= count + 1;
    end
endmodule
```

## 11.3 Testbench

Listing 21: 4-bit Counter Testbench

```
module test_counter_4bit;
    logic clk, reset;
    logic [3:0] count;

    counter_4bit uut (
        .clk(clk),
        .reset(reset),
        .count(count)
    );

    initial begin
        // Initialize clock
        clk = 0;
        forever #5 clk = ~clk;
    end

    initial begin
        // Test vectors
        reset = 1; #10; // Reset counter
        reset = 0;
        #100 $stop; // Run simulation for 100 time units
    end
endmodule
```

## 11.4 Simulation Results

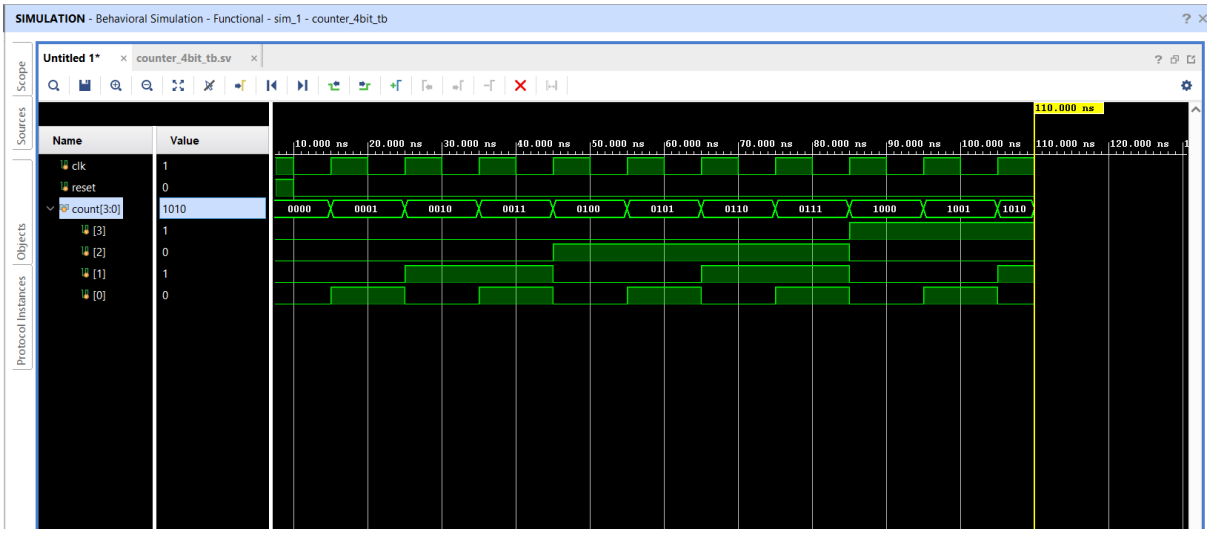


Figure 20: Simulation results of the 4-bit counter

## 11.5 Schematic

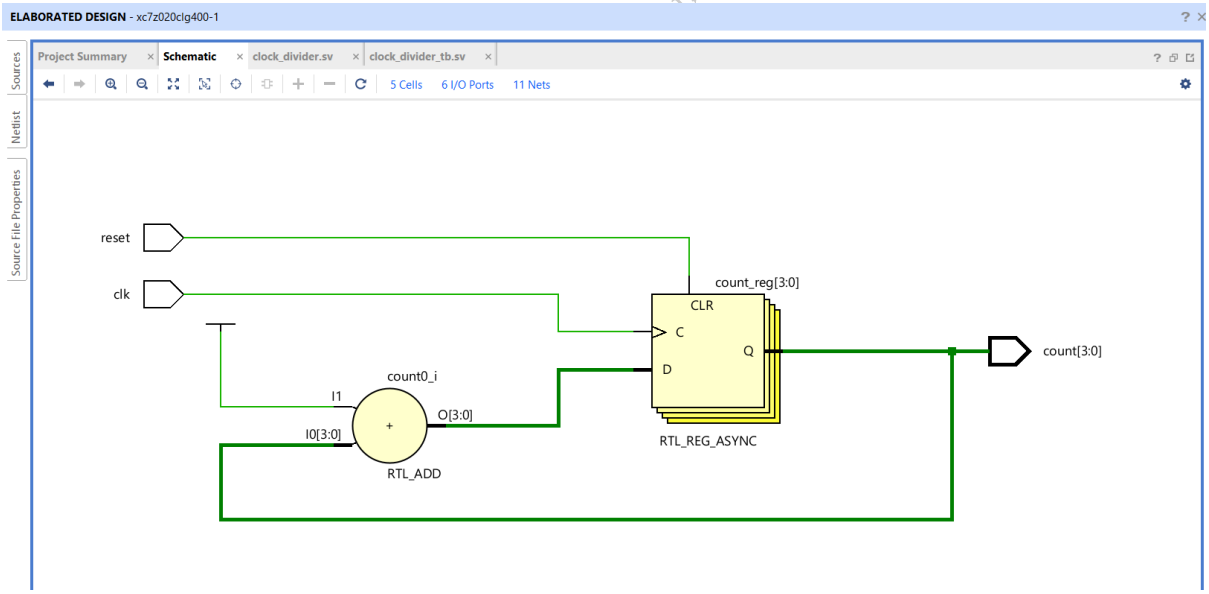


Figure 21: Schematic of the 4-bit counter

## 11.6 Advantages

- Simple design for counting events or cycles.
- Can be easily extended to count higher values by increasing bit width.

## 11.7 Disadvantages

- Limited by the bit width; overflow occurs if the count exceeds the maximum value.
- Ripple counters can have propagation delays.

## 11.8 Applications

- Digital clocks.
- Event counting.
- Frequency division.

# 12 Clock Divider

## 12.1 Description

A clock divider is a sequential circuit used to reduce the frequency of an input clock signal. It generates a slower clock signal by counting the number of input clock cycles and toggling the output clock signal accordingly. The division factor determines how much the frequency is reduced.

## 12.2 RTL Code

Listing 22: Clock Divider RTL Code

```
module clock_divider (
    input logic clk_in,
    input logic reset,
    output logic clk_out
);
    logic [31:0] counter;
    parameter DIVISOR = 25; // Define the division factor

    always_ff @(posedge clk_in or posedge reset) begin
        if (reset) begin
            counter <= 0;
            clk_out <= 0;
        end else begin
            if (counter == (DIVISOR/2 - 1)) begin
                counter <= 0;
                clk_out <= ~clk_out;
            end else begin
                counter <= counter + 1;
            end
        end
    end
endmodule
```

## 12.3 Testbench

Listing 23: Clock Divider Testbench

```
module test_clock_divider;
    logic clk_in, reset;
    logic clk_out;

    clock_divider #(.DIVISOR(25)) uut (
        .clk_in(clk_in),
        .reset(reset),
        .clk_out(clk_out)
    );

```

```
    initial begin
        // Initialize clock
        clk_in = 0;
        forever #1 clk_in = ~clk_in; // 2 time units per clock cycle
    end

    initial begin
        // Test vectors
        reset = 1; #10; // Reset clock divider
        reset = 0;
        #200 $stop; // Run simulation for 200 time units
    end
endmodule
```

## 12.4 Simulation Results

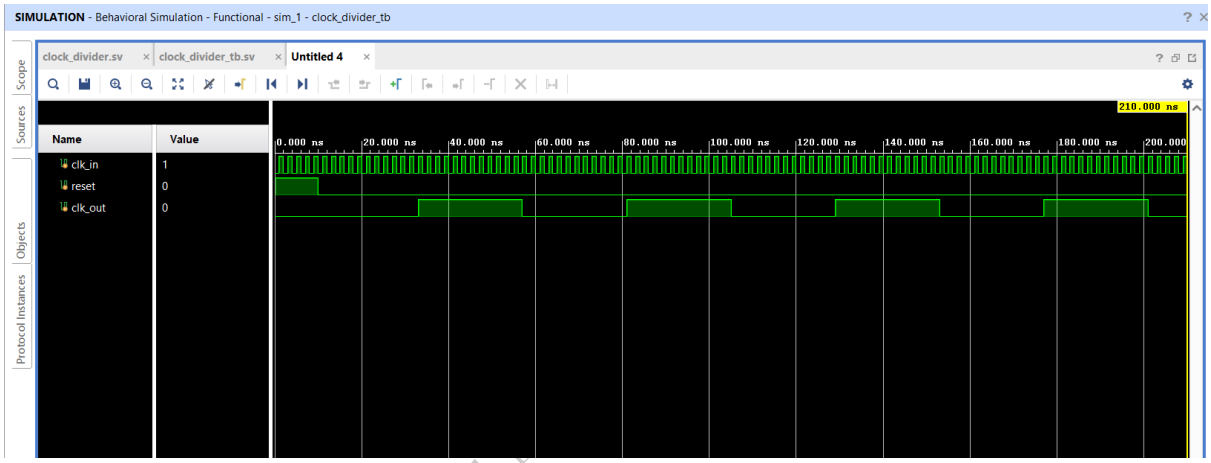


Figure 22: Simulation results of the clock divider



## 12.5 Schematic

## 12.6 Frequency Division Explanation

To divide the input clock frequency by a factor of 25, the 'DIVISOR' parameter in the 'clock$_d$ivider'$moduleissetto25.Theco$
1'.$Thisresultsinanoutputclockfrequencythatis\frac{1}{25}$ of the input clock frequency.

## 12.7 Code Explanation

### 12.7.1 Clock Divider Module

The 'clock$_d$ivider'$moduleworksasfollows$ :

**Inputs and Outputs**: It has an input clock ('clk$_i$n'), $aresetsignal('reset'), andanoutputclock('clk_out').$**Counter** :
$A32-bitcounterkeepstrackofthenumberofinputclockcycles.Theparameter'DIVISOR'setsthethresholdforthecou$

**Reset Logic**: When the 'reset' signal is high, the counter and 'clk$_o$ut'$areresettozero.$**Counter Increment and Output**
$Oneveryrisingedgeof'clk_in', thecounterincrements.Whenthecounterreaches'DIVISOR/2-1', itresetstozeroandtoggle$

### 12.7.2 Frequency Division Calculation

The output frequency is determined by the 'DIVISOR' parameter. Specifically:

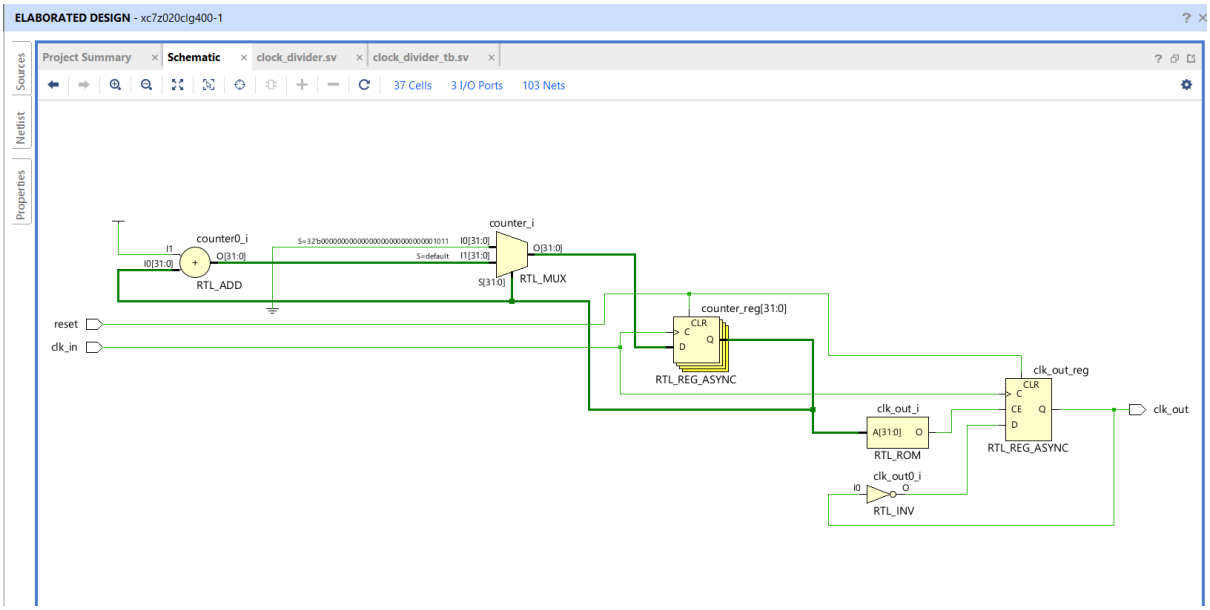


Figure 23: Schematic of the clock divider

- The ‘DIVISOR‘ parameter sets how much the input clock frequency is divided. For example, setting ‘DIVISOR‘ to 25 will divide the input frequency by 25.
- The counter toggles ‘clk$_out$‘$when it reaches$‘$DIVISOR/2-1$‘$because the counter counts from 0 to$‘$DIVISOR/2-1‘. $This produces an output clock frequency that is \frac{1}{\text{DIVISOR}}$ of the input clock frequency.

To adjust the division factor:

- To divide by 10, set ‘DIVISOR‘ to 10. This will produce an output clock frequency that is $\frac{1}{10}$ of the input clock frequency.
- To divide by 35, set ‘DIVISOR‘ to 35. This will produce an output clock frequency that is $\frac{1}{35}$ of the input clock frequency.



## 12.8 Advantages

- Reduces the frequency of a clock signal, useful for timing control.
- Simple implementation with counters.

## 12.9 Disadvantages

- Fixed division ratio; flexibility requires additional logic.
- May introduce delays in the clock signal.

## 12.10 Applications

- Frequency scaling for different parts of a circuit.
- Timing generation in communication protocols.
- Power management in digital systems.

# 13 Half Adder

## 13.1 Description

A half adder is a combinational circuit that adds two single-bit binary numbers and produces a sum and a carry-out.

## 13.2 RTL Code

Listing 24: Half Adder RTL Code

```
module half_adder (
    input logic A,
    input logic B,
    output logic Sum,
    output logic Carry
);
    assign Sum = A ^ B;      // XOR for Sum
    assign Carry = A & B;    // AND for Carry
endmodule
```



## 13.3 Testbench

Listing 25: Half Adder Testbench

```
module test_half_adder;
    logic A, B;
    logic Sum, Carry;

    half_adder uut (
        .A(A),
        .B(B),
        .Sum(Sum),
        .Carry(Carry)
    );

    initial begin
        // Test vectors
        A = 0; B = 0;
        #10 A = 0; B = 1;
        #10 A = 1; B = 0;
        #10 A = 1; B = 1;
        #10 $stop;
    end
endmodule
```

## 13.4 Simulation Results

## 13.5 Schematic

## 13.6 Advantages

- Simple design and easy to implement.
- Fast operation due to minimal gate delay.

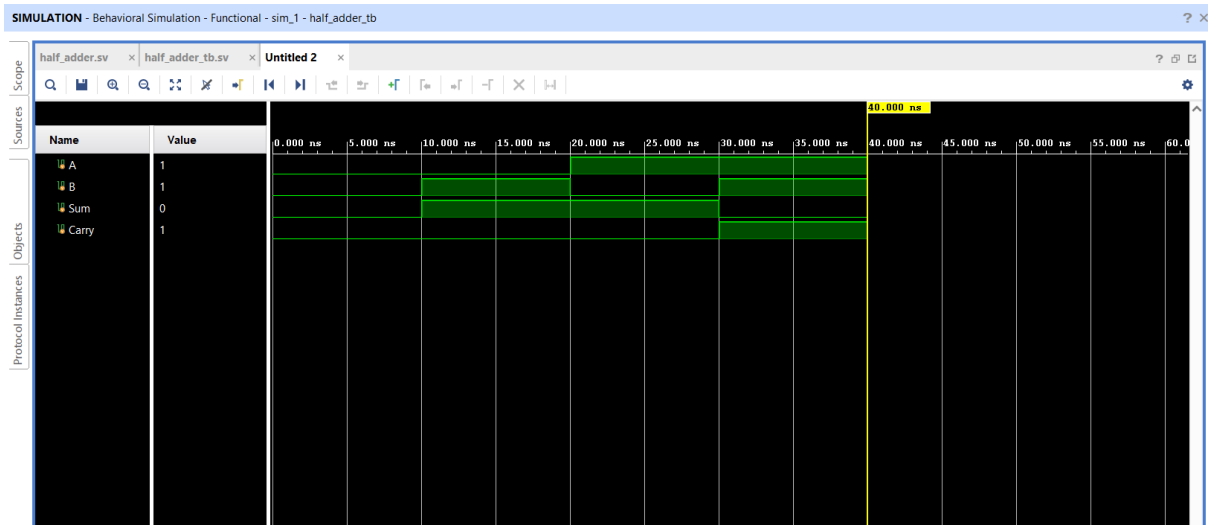


Figure 24: Simulation results of the halfAdder

## 13.7 Disadvantages

- Can only add two single-bit numbers.
- No provision for carry-in, limiting its use in multi-bit addition.

## 13.8 Applications

- Basic arithmetic operations in digital circuits.
- Component in more complex adder circuits like full adders.



# 14 Full Adder

## 14.1 Description

A full adder is a combinational circuit that adds three single-bit binary numbers (two operands and a carry-in) and produces a sum and a carry-out.

## 14.2 RTL Code

Listing 26: Full Adder RTL Code

```
module full_adder (
    input logic A,
    input logic B,
    input logic Cin,
    output logic Sum,
    output logic Cout
);
    assign Sum = A ^ B ^ Cin;             // XOR for Sum
    assign Cout = (A & B)  (Cin & (A ^ B)); // AND-OR for Carry
endmodule
```

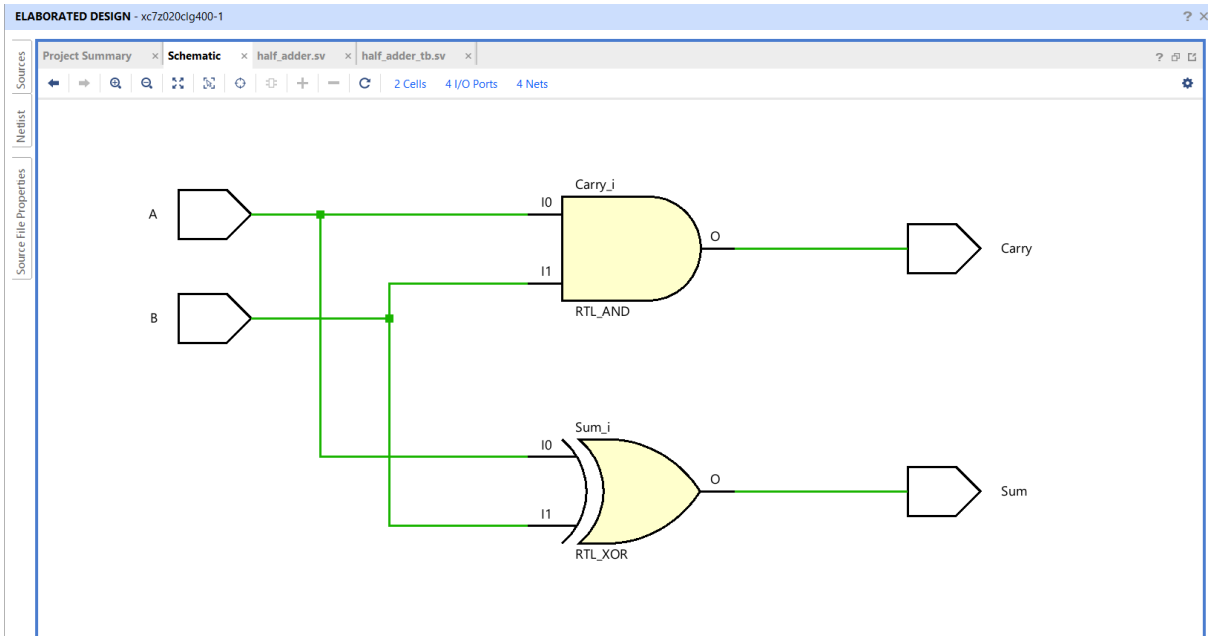


Figure 25: Schematic of half adder

## 14.3 Testbench



Listing 27: Full Adder Testbench

```
module test_full_adder;
    logic A, B, Cin;
    logic Sum, Cout;

    full_adder uut (
        .A(A),
        .B(B),
        .Cin(Cin),
        .Sum(Sum),
        .Cout(Cout)
    );

    initial begin
        // Test vectors
        A = 0; B = 0; Cin = 0;
        #10 A = 0; B = 1; Cin = 0;
        #10 A = 1; B = 0; Cin = 0;
        #10 A = 1; B = 1; Cin = 0;
        #10 A = 0; B = 0; Cin = 1;
        #10 A = 0; B = 1; Cin = 1;
        #10 A = 1; B = 0; Cin = 1;
        #10 A = 1; B = 1; Cin = 1;
        #10 $stop;
    end
endmodule
```

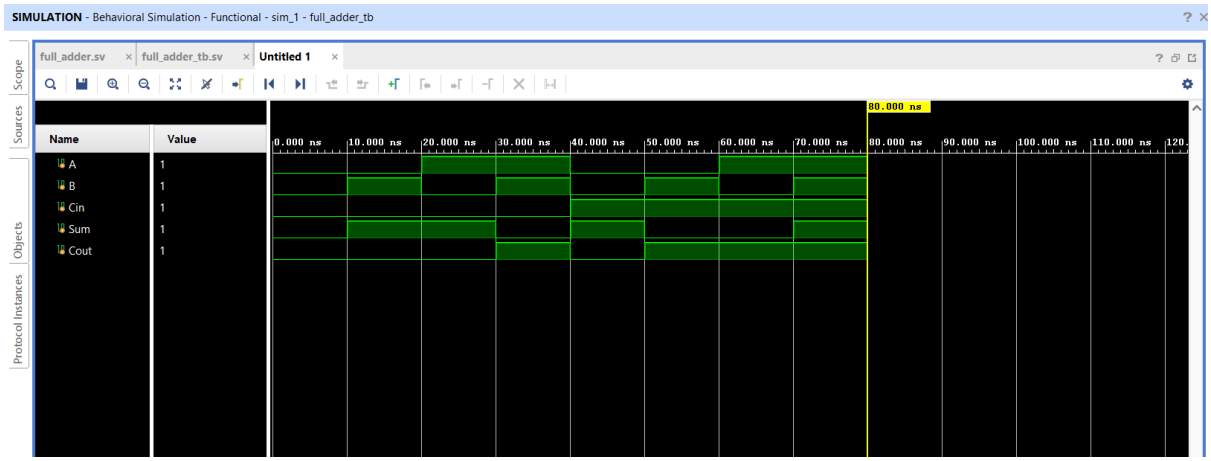


Figure 26: Simulation results of the Full Adder

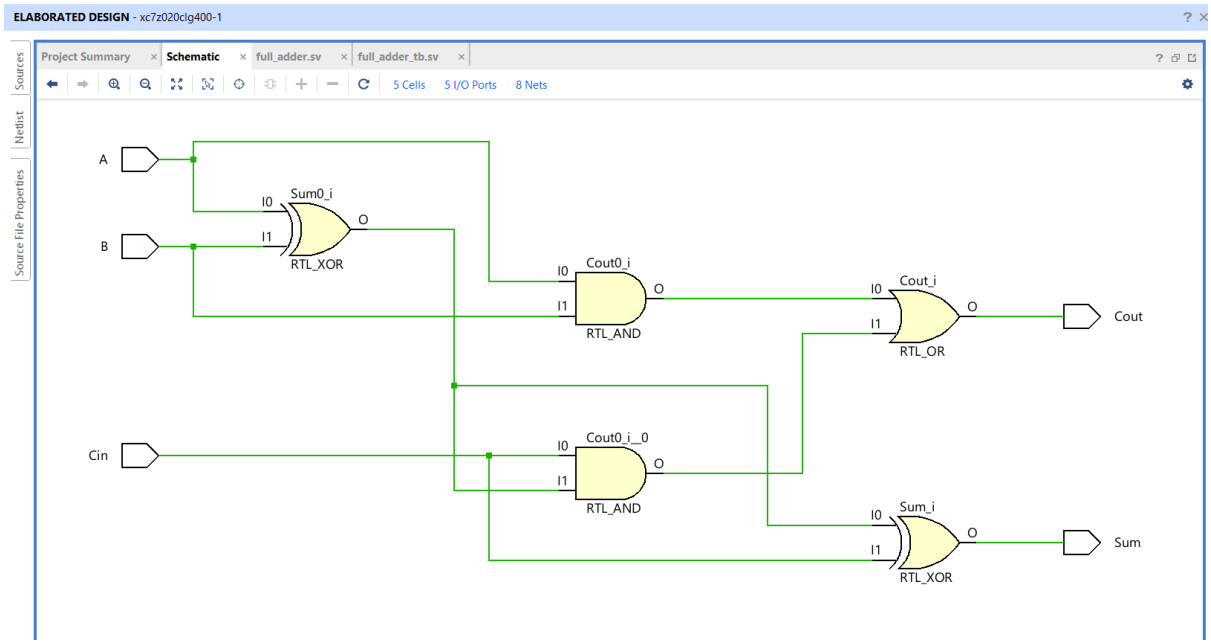


Figure 27: Schematic of adder Subtractor

## 14.4 Simulation Results

## 14.5 Schematic

## 14.6 Advantages

- Can add three single-bit numbers, including a carry-in.
- Essential building block for constructing multi-bit adders.

## 14.7 Disadvantages

- Slightly more complex than a half adder.
- Increased gate delay compared to a half adder.

## 14.8 Applications

- Multi-bit binary addition.
- Arithmetic logic units (ALUs).
- Digital signal processing.

# 15 Half Subtractor

## 15.1 Description

A half subtractor is a combinational circuit that subtracts two single-bit binary numbers and produces a difference and a borrow-out.

## 15.2 RTL Code

Listing 28: Half Subtractor RTL Code

```
module half_subtractor (
    input logic A,
    input logic B,
    output logic Diff,
    output logic Borrow
);
    assign Diff = A ^ B;            // XOR for Difference
    assign Borrow = ~A & B;         // AND-NOT for Borrow
endmodule
```



## 15.3 Testbench

Listing 29: Half Subtractor Testbench

```
module test_half_subtractor;
    logic A, B;
    logic Diff, Borrow;

    half_subtractor uut (
        .A(A),
        .B(B),
        .Diff(Diff),
        .Borrow(Borrow)
    );

    initial begin
        // Test vectors
        A = 0; B = 0;
        #10 A = 0; B = 1;
        #10 A = 1; B = 0;
        #10 A = 1; B = 1;
        #10 $stop;
    end
endmodule
```

## 15.4 Simulation Results

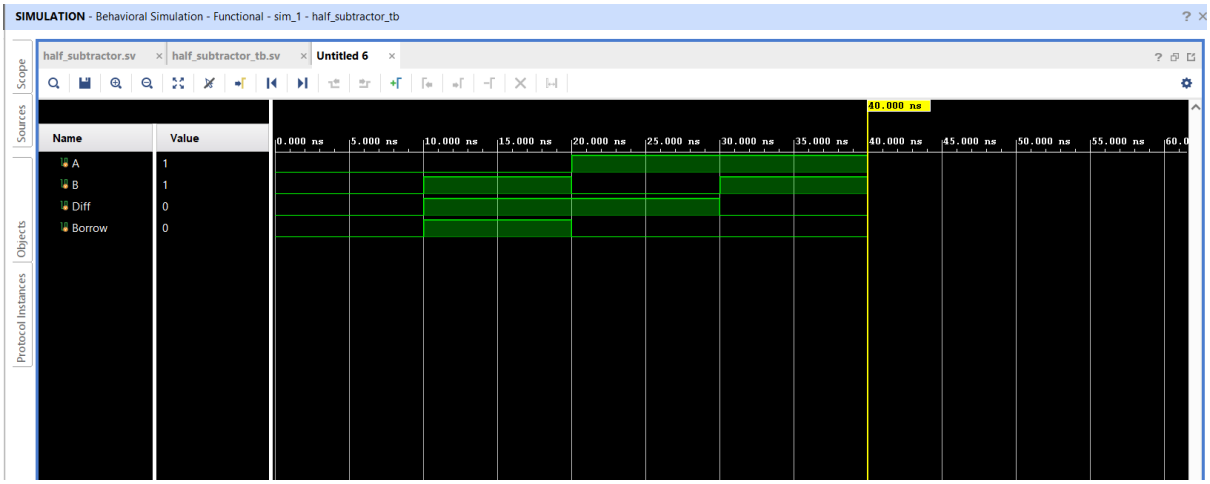


Figure 28: Simulation results of the half subtractor

## 15.5 Schematic

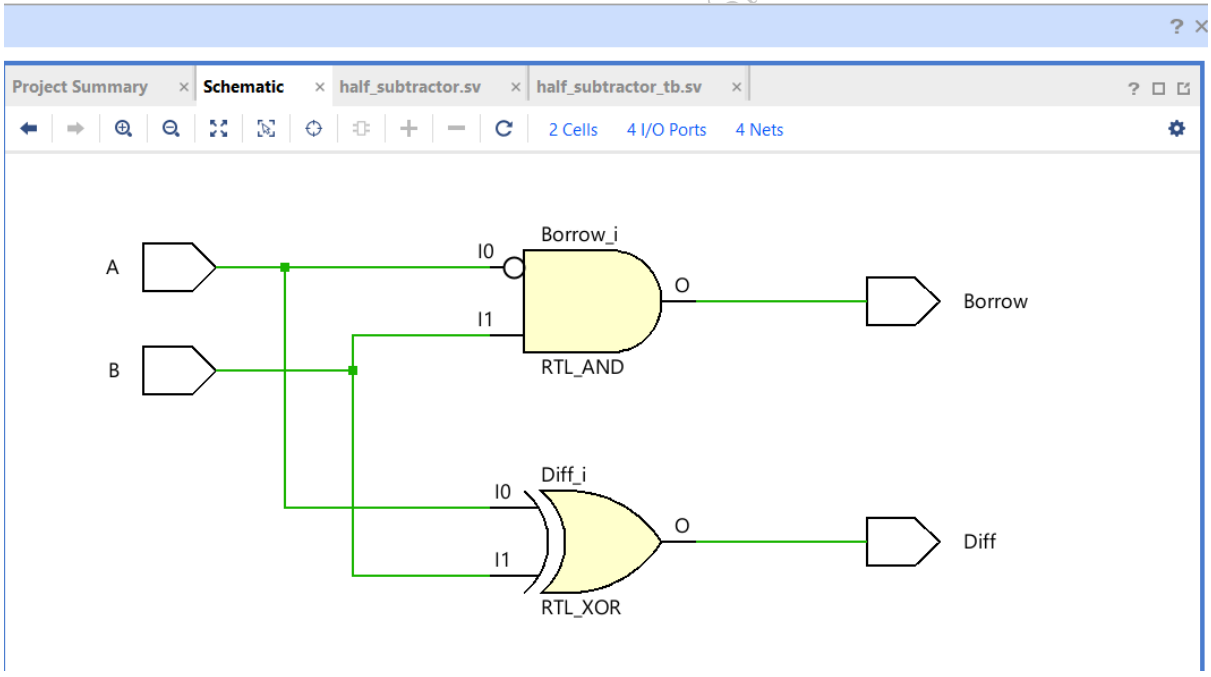


Figure 29: Schematic of half Subtractor

## 15.6 Advantages

- Simple design for basic subtraction operations.
- Fast operation due to minimal gate delay.

## 15.7 Disadvantages

- Can only subtract two single-bit numbers.
- No provision for borrow-in, limiting its use in multi-bit subtraction.

## 15.8 Applications

- Basic arithmetic operations in digital circuits.
- Component in more complex subtractor circuits like full subtractors.

# 16 Full Subtractor

## 16.1 Description

A full subtractor is a combinational circuit that subtracts three single-bit binary numbers (two operands and a borrow-in) and produces a difference and a borrow-out.

## 16.2 RTL Code

Listing 30: Full Subtractor RTL Code

```
module full_subtractor (
    input logic A,
    input logic B,
    input logic Bin,
    output logic Diff,
    output logic Bout
);
    assign Diff = A ^ B ^ Bin;          // XOR for Difference
    assign Bout = (~A & B)  (Bin & (~A ^ B)); // AND-OR for Borrow
endmodule
```

## 16.3 Testbench

Listing 31: Full Subtractor Testbench

```
module test_full_subtractor;
    logic A, B, Bin;
    logic Diff, Bout;

    full_subtractor uut (
        .A(A),
        .B(B),
        .Bin(Bin),
        .Diff(Diff),
        .Bout(Bout)
    );

    initial begin
        // Test vectors
        A = 0; B = 0; Bin = 0;
        #10 A = 0; B = 1; Bin = 0;
```

```
        #10 A = 1; B = 0; Bin = 0;
        #10 A = 1; B = 1; Bin = 0;
        #10 A = 0; B = 0; Bin = 1;
        #10 A = 0; B = 1; Bin = 1;
        #10 A = 1; B = 0; Bin = 1;
        #10 A = 1; B = 1; Bin = 1;
        #10 $stop;
    end
endmodule
```

## 16.4 Simulation Results

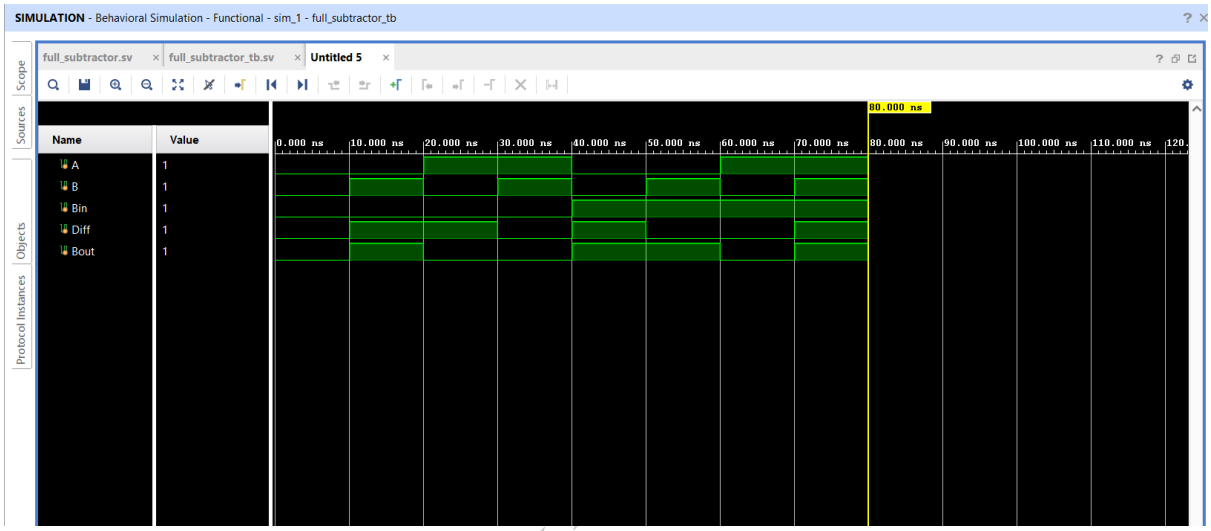


Figure 30: Simulation results of the Full Subtractor



## 16.5 Schematic

## 16.6 Advantages

- Can subtract three single-bit numbers, including a borrow-in.
- Essential building block for constructing multi-bit subtractors.

## 16.7 Disadvantages

- Slightly more complex than a half subtractor.
- Increased gate delay compared to a half subtractor.

## 16.8 Applications

- Multi-bit binary subtraction.
- Arithmetic logic units (ALUs).
- Digital signal processing.

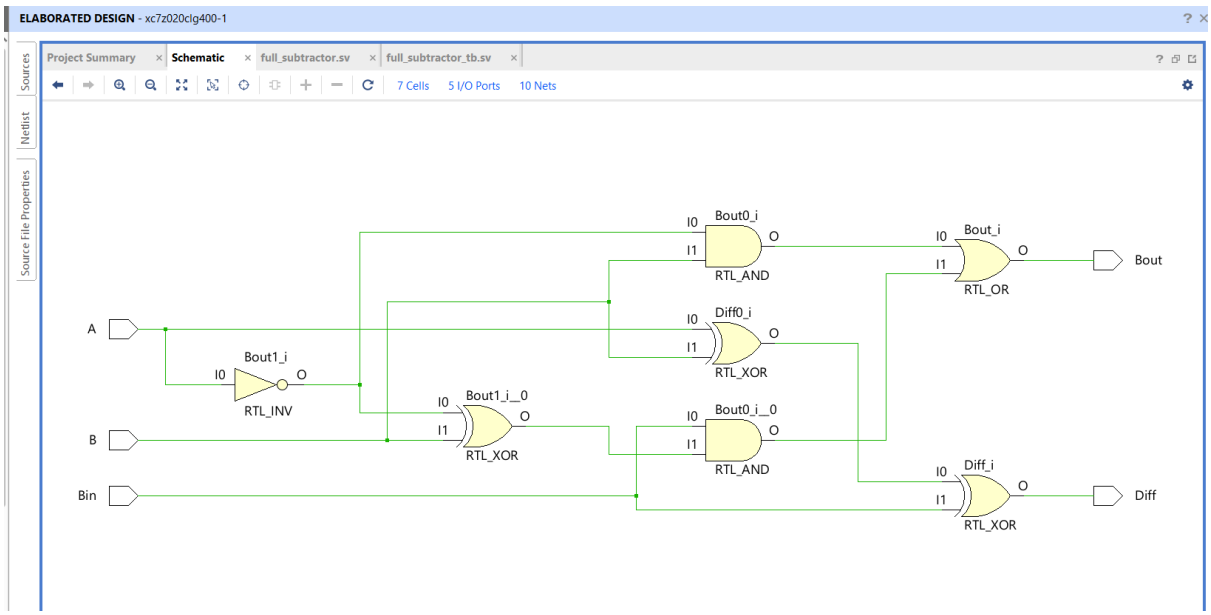


Figure 31: Schematic of Full Subtractor