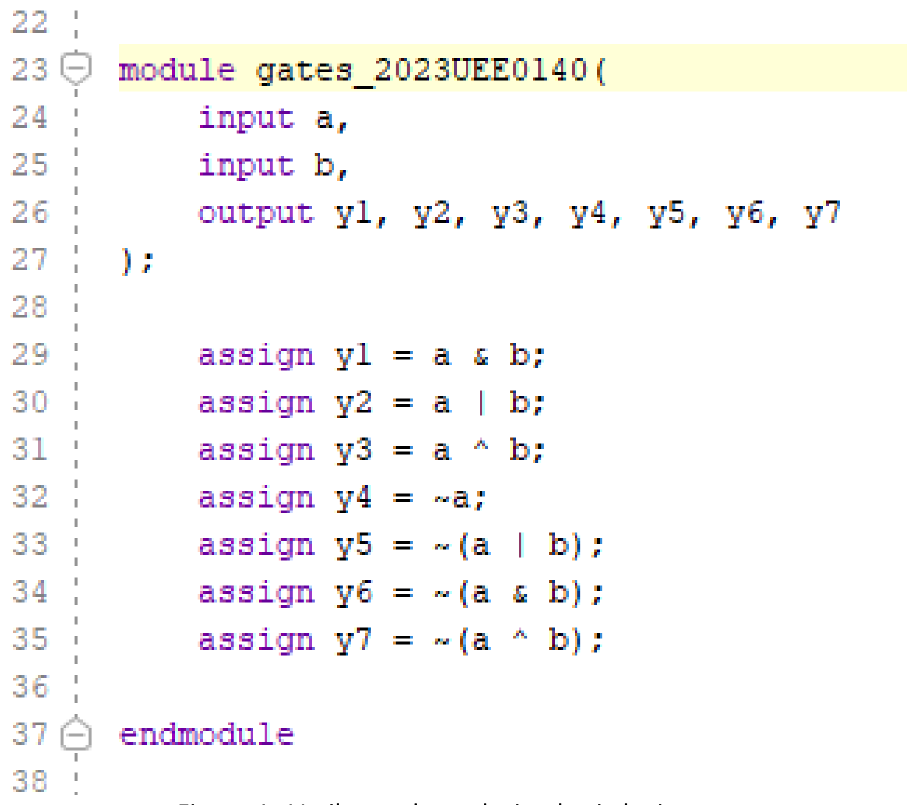
Krishna

2023UEE0140

Digital Design Lab Report

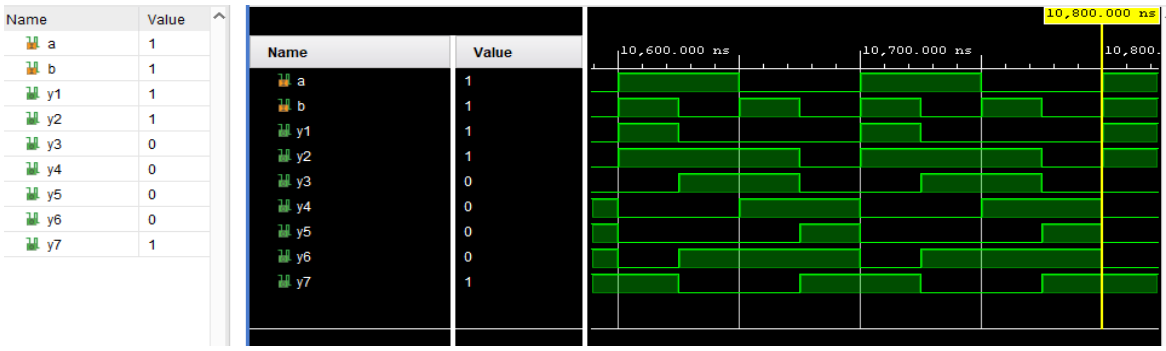
Aim: Implement basic Logic Gates in Verilog

This section covers the implementation of basic logic gates such as AND, OR, and NOT using Verilog. The logic gates are fundamental building blocks in digital design, and their implementation showcases how digital systems process binary information.



```verilog

module gates_2023UEE0140(
    input a,
    input b,
    output y1, y2, y3, y4, y5, y6, y7
);

    assign y1 = a & b;
    assign y2 = a | b;
    assign y3 = a ^ b;
    assign y4 = ~a;
    assign y5 = ~(a | b);
    assign y6 = ~(a & b);
    assign y7 = ~(a ^ b);

endmodule

```

Figure 1: Verilog code to design basic logic gates



Name	Value
a	1
b	1
y1	1
y2	1
y3	0
y4	0
y5	0
y6	0
y7	1

Figure 2: Simulation

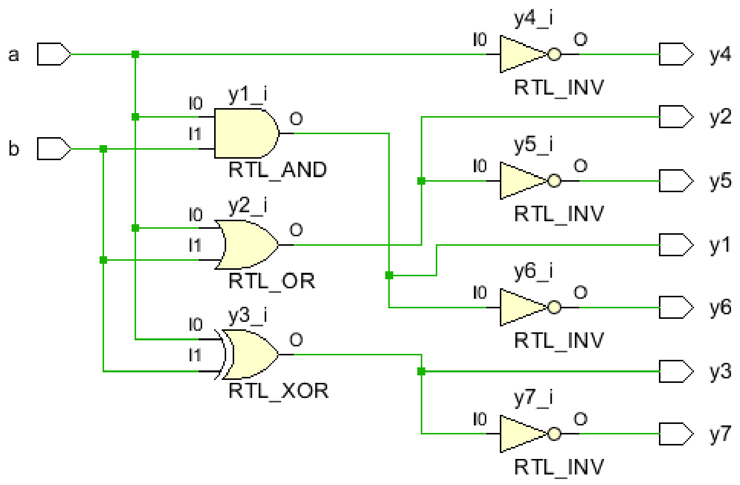
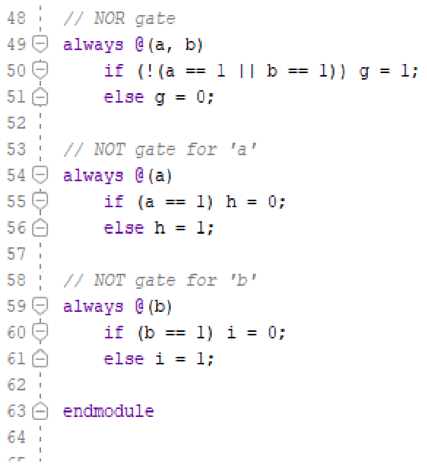


Figure 3: Schematic

Aim: To implement basic logic gates with if-else in Verilog

In this section, we implement basic logic gates using the if-else conditional statements. This approach helps understand how control flow can be used in digital design to achieve similar results

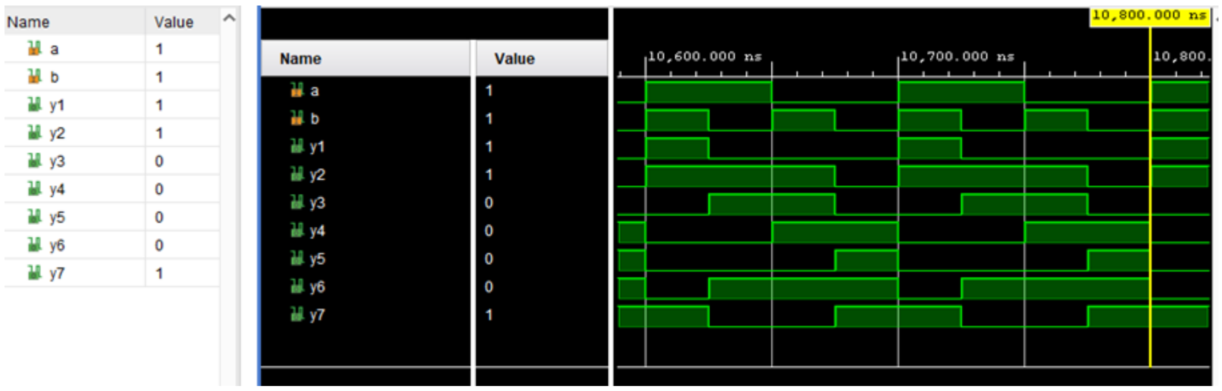
```
module gates_ifelse_2023UEE0140(
    input a, b,              // Inputs a and b
    output reg c, d, e, f, g, h, i  // Outputs for different gates
);

// AND gate
always @(a, b)
    if (a == 1 && b == 1) c = 1;
    else c = 0;

// OR gate
always @(a, b)
    if (a == 1 || b == 1) d = 1;
    else d = 0;

// XOR gate
always @(a, b)
    if (a != b) e = 1;
    else e = 0;

// NAND gate
always @(a, b)
    if (!(a == 1 && b == 1)) f = 1;
    else f = 0;

```



```
// NOR gate
always @(a, b)
    if (!(a == 1 || b == 1)) g = 1;
    else g = 0;

// NOT gate for 'a'
always @(a)
    if (a == 1) h = 0;
    else h = 1;

// NOT gate for 'b'
always @(b)
    if (b == 1) i = 0;
    else i = 1;

endmodule
```

Figure 4, 5 Verilog Code to design basic logic gates using If-Else

4



Name	Value
a	1
b	1
y1	1
y2	1
y3	0
y4	0
y5	0
y6	0
y7	1

Figure 6: Simulation

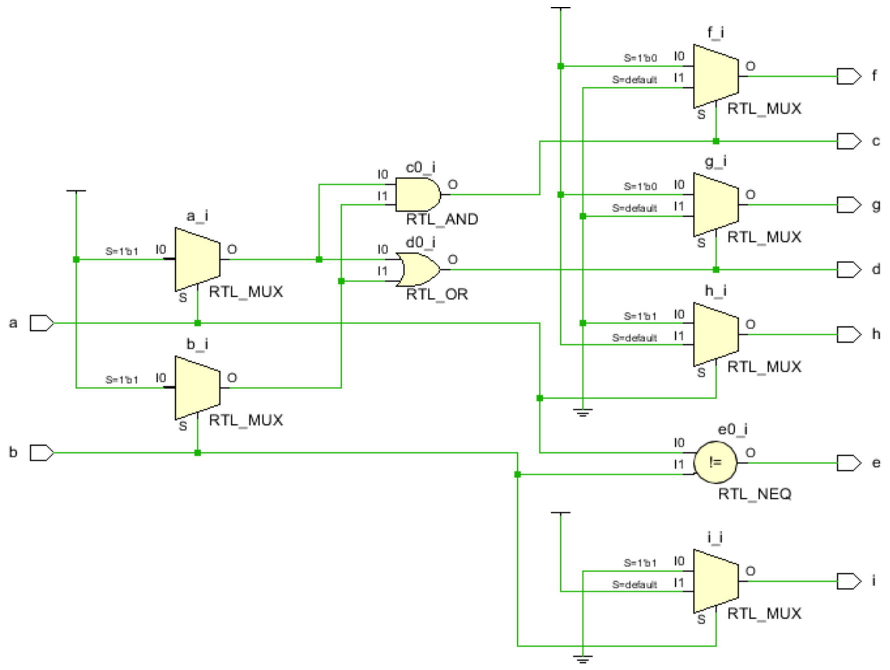
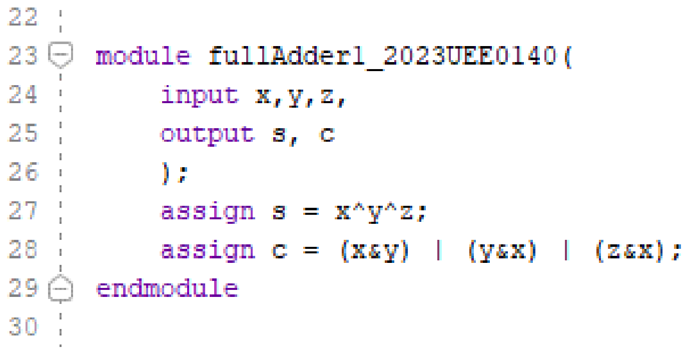


Figure 7: Schematic

Aim: To implement Full Adder using OR and And Gates in Verilog and also with XOR gate.

The full adder is a crucial component in arithmetic logic units. This section details the implementation of a full adder using both AND/OR gates and XOR gates, highlighting the differences in design approaches.



```
module fullAdder1_2023UEE0140(
    input x,y,z,
    output s, c
    );
    assign s = x^y^z;
    assign c = (x&y) | (y&x) | (z&x);
endmodule
```

Figure 8: Verilog Code for Full Adder using And and OR Gates

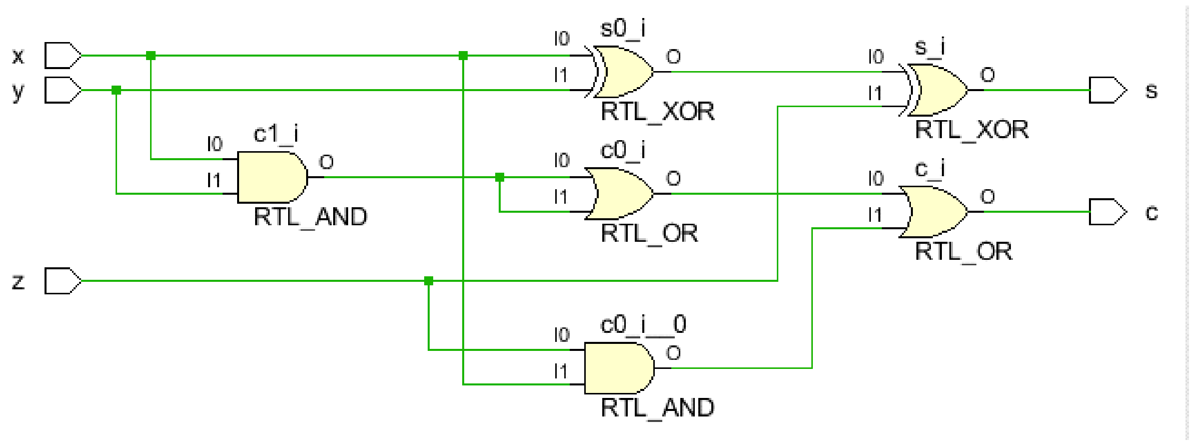
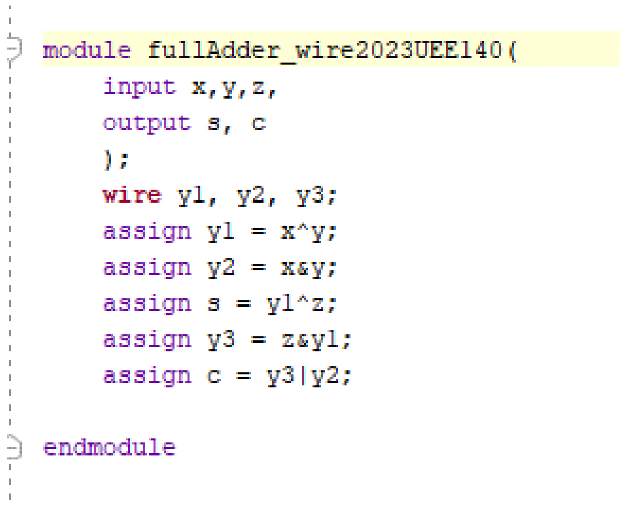


Figure 9: Schematic



```
module fullAdder_wire2023UEE140(
    input x,y,z,
    output s, c
    );
    wire y1, y2, y3;
    assign y1 = x^y;
    assign y2 = x&y;
    assign s = y1^z;
    assign y3 = z&y1;
    assign c = y3|y2;

endmodule
```

Figure 10: Full Adder using wires and XOR gate

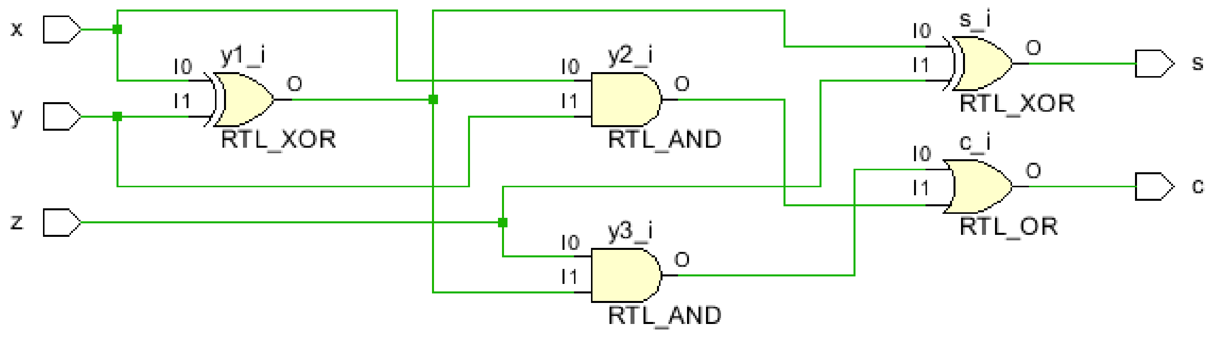
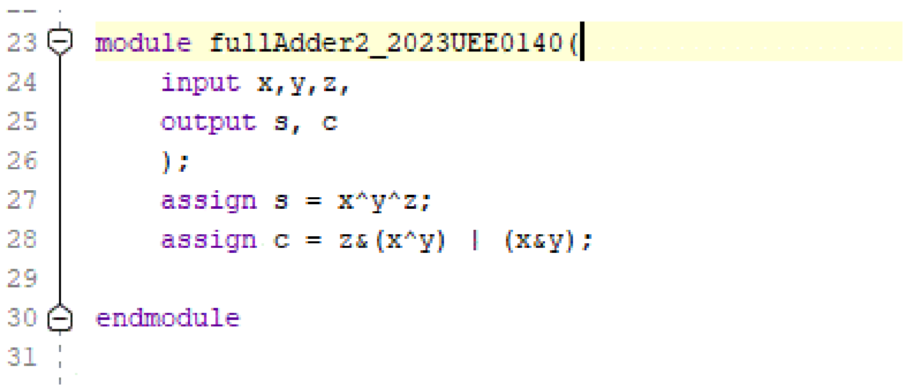


Figure 11: Schematic



```
module fullAdder2_2023UEE0140(
    input x,y,z,
    output s, c
    );
    assign s = x^y^z;
    assign c = z&(x^y) | (x&y);

endmodule
```

Figure 12: Verilog Code for Full Adder using XOR Gate

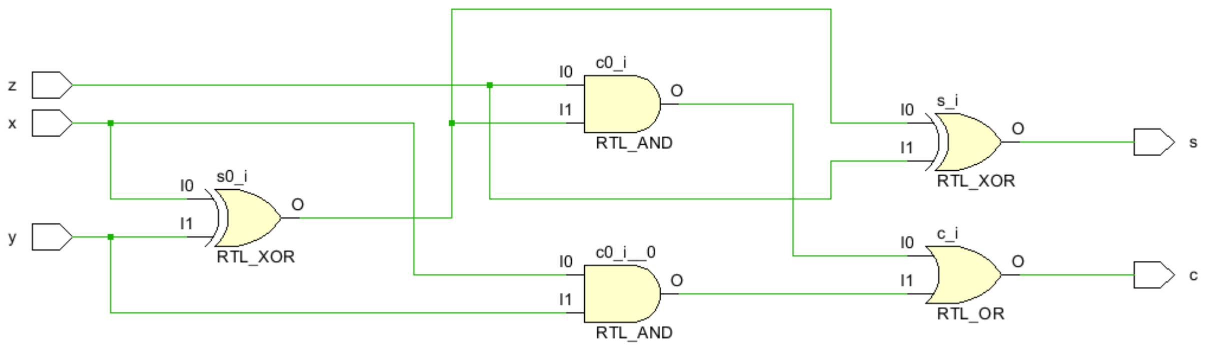
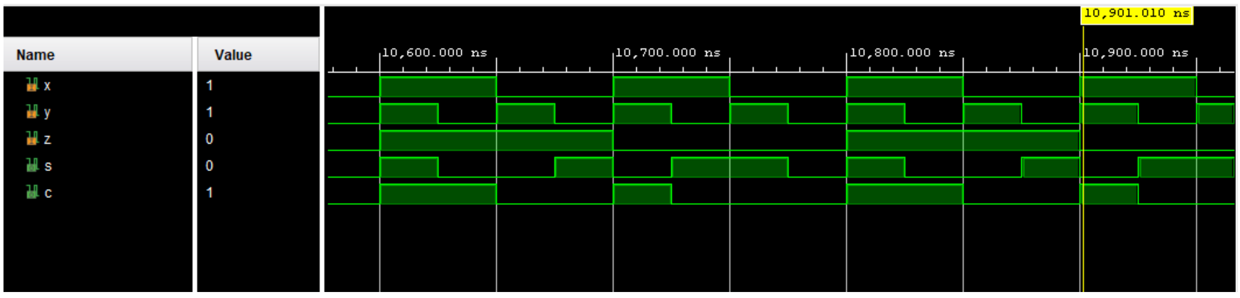


Figure 13: Schematic

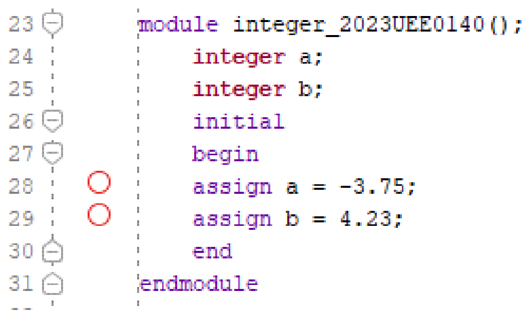


Name	Value
x	1
y	1
z	0
s	0
c	1

Figure 14: Simulation

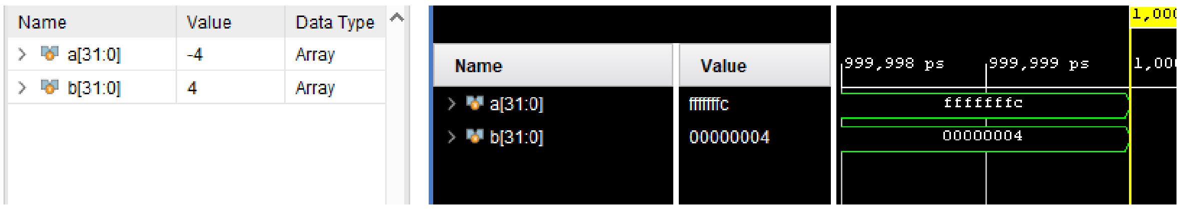
Aim: To understand Data Assignment (Real/Integer) in Verilog

This section explores data assignment in Verilog, focusing on real and integer types. Understanding these data types is essential for managing data accurately in digital systems.



```
module integer_2023UEE0140();
    integer a;
    integer b;
    initial
    begin
    assign a = -3.75;
    assign b = 4.23;
    end
endmodule
```

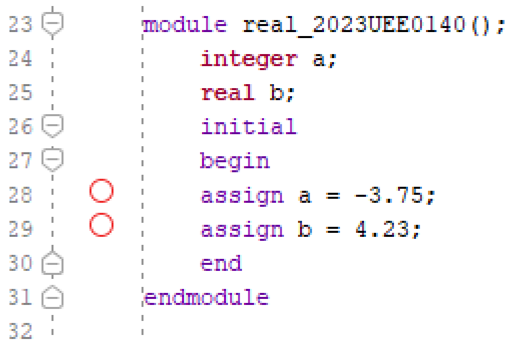
Figure 15: Integer Data Type



Name	Value	Data Type
a[31:0]	-4	Array
b[31:0]	4	Array

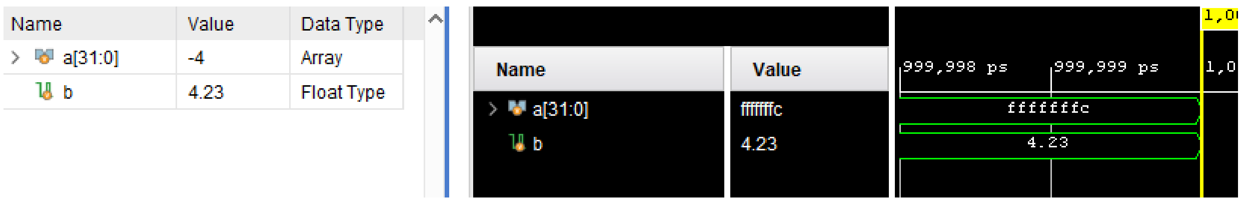
Name	Value
a[31:0]	fffffffc
b[31:0]	00000004

Figure 16: Simulation



```
module real_2023UEE0140();
    integer a;
    real b;
    initial
    begin
    assign a = -3.75;
    assign b = 4.23;
    end
endmodule
```

Figure 17: Real Data Type



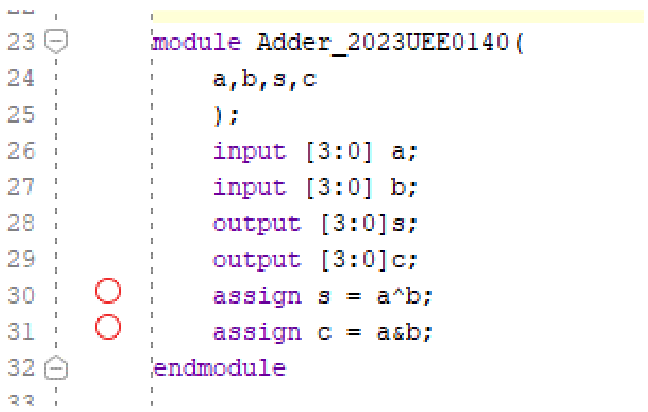
Name	Value	Data Type
a[31:0]	-4	Array
b	4.23	Float Type

Name	Value
a[31:0]	fffffffc
b	4.23

Figure 18: Simulation

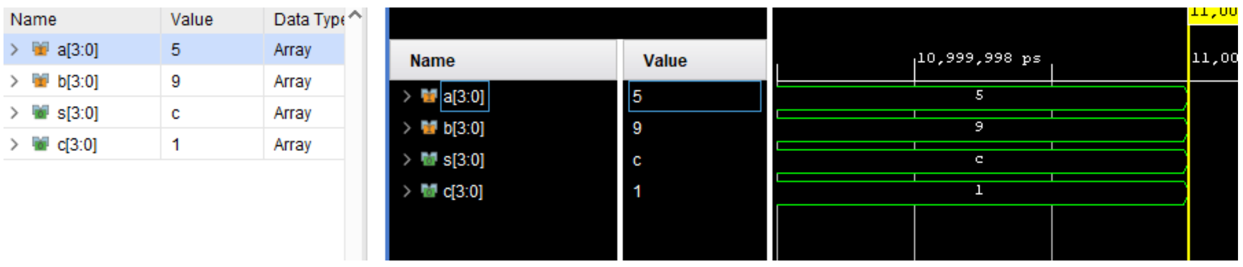
Aim: To implement Four Bit Half Adder

In this section, we implement a four-bit half adder, which extends the concept of a basic half adder to handle multi-bit inputs.



```
module Adder_2023UEE0140(
    a,b,s,c
    );
    input [3:0] a;
    input [3:0] b;
    output [3:0]s;
    output [3:0]c;
    assign s = a^b;
    assign c = a&b;
endmodule
```

Figure 19: Verilog code for Half Adder



Name	Value	Data Type
a[3:0]	5	Array
b[3:0]	9	Array
s[3:0]	c	Array
c[3:0]	1	Array

Name	Value
a[3:0]	5
b[3:0]	9
s[3:0]	c
c[3:0]	1

Figure 20: Simulation

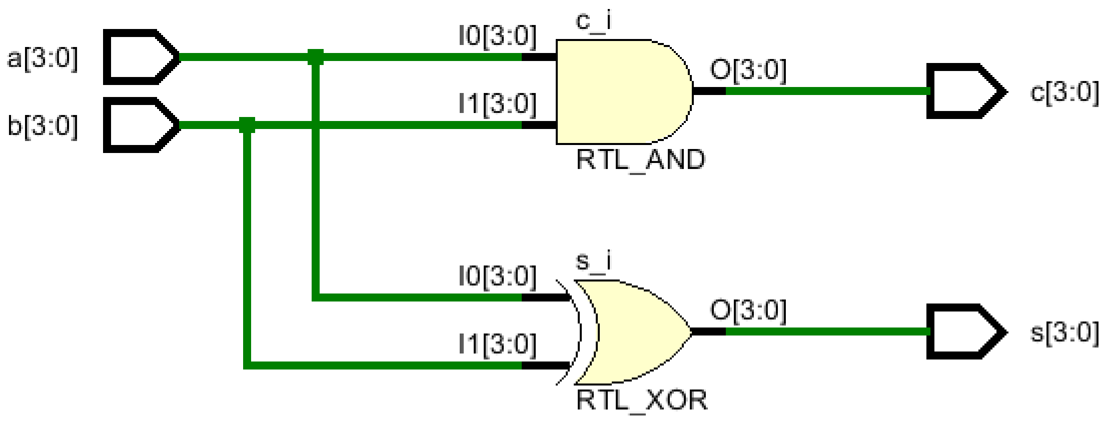
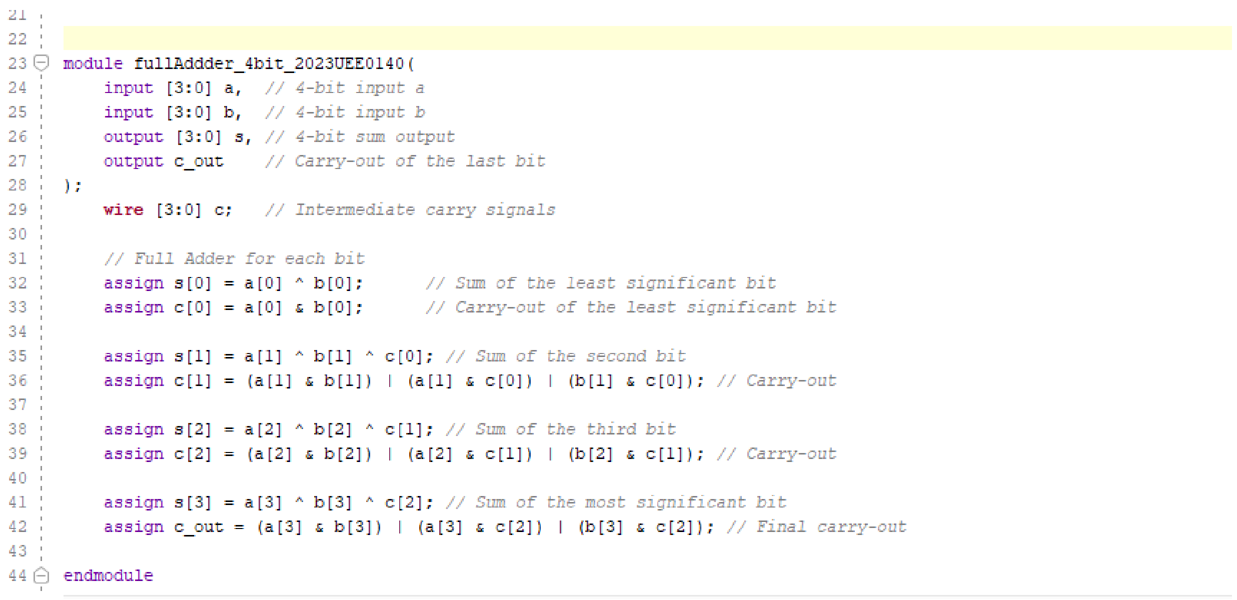


Figure 21: Schematic

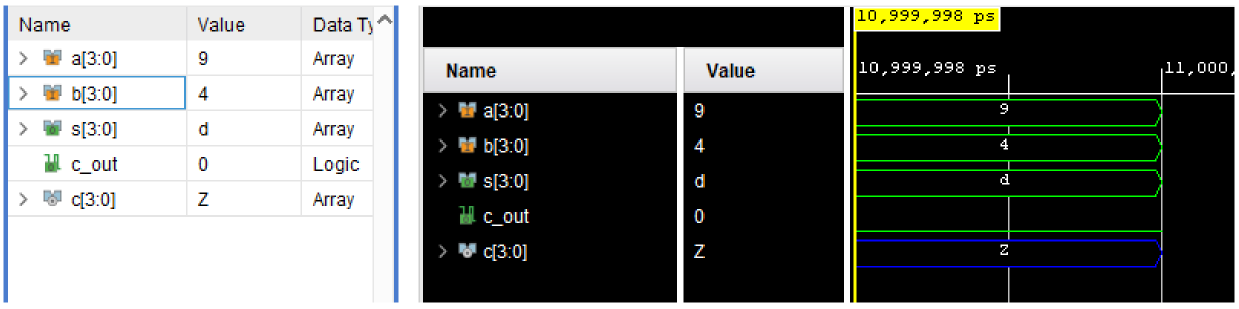
Aim: To implement Four Bit Full Adder

This section describes the implementation of a four-bit full adder, which adds two four-bit binary numbers and produces a sum and carry output.



```verilog

module fullAddder_4bit_2023UEE0140(
    input [3:0] a,  // 4-bit input a
    input [3:0] b,  // 4-bit input b
    output [3:0] s, // 4-bit sum output
    output c_out    // Carry-out of the last bit
);
    wire [3:0] c;   // Intermediate carry signals

    // Full Adder for each bit
    assign s[0] = a[0] ^ b[0];        // Sum of the least significant bit
    assign c[0] = a[0] & b[0];        // Carry-out of the least significant bit

    assign s[1] = a[1] ^ b[1] ^ c[0]; // Sum of the second bit
    assign c[1] = (a[1] & b[1]) | (a[1] & c[0]) | (b[1] & c[0]); // Carry-out

    assign s[2] = a[2] ^ b[2] ^ c[1]; // Sum of the third bit
    assign c[2] = (a[2] & b[2]) | (a[2] & c[1]) | (b[2] & c[1]); // Carry-out

    assign s[3] = a[3] ^ b[3] ^ c[2]; // Sum of the most significant bit
    assign c_out = (a[3] & b[3]) | (a[3] & c[2]) | (b[3] & c[2]); // Final carry-out

endmodule
```

Figure 22: Verilog code for four-bit Full Adder



Name	Value	Data Ty
a[3:0]	9	Array
b[3:0]	4	Array
s[3:0]	d	Array
c_out	0	Logic
c[3:0]	Z	Array

```
10,999,998 ps

Name      Value    10,999,998 ps          11,000,
a[3:0]    9        9
b[3:0]    4        4
s[3:0]    d        d
c_out     0
c[3:0]    Z        Z
```

Figure 23: Simulation

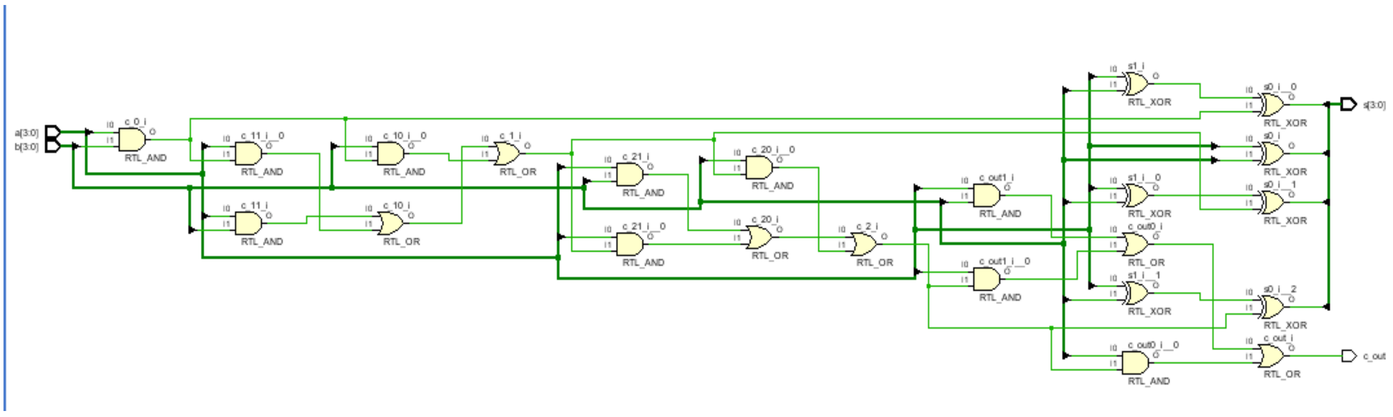
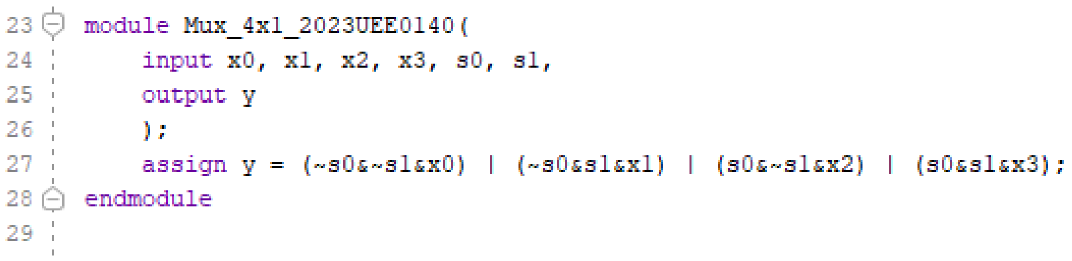


Figure 24: Schematic

Aim: To implement 4 X 1 MUX with Gates

In this section, we implement a 4-to-1 multiplexer (MUX) using basic gates. A MUX selects one of several input signals and forwards the selected input to a single output line.



```
module Mux_4x1_2023UEE0140(
    input x0, x1, x2, x3, s0, s1,
    output y
    );
    assign y = (~s0&~s1&x0) | (~s0&s1&x1) | (s0&~s1&x2) | (s0&s1&x3);
endmodule
```

Figure 25: MUX using gates

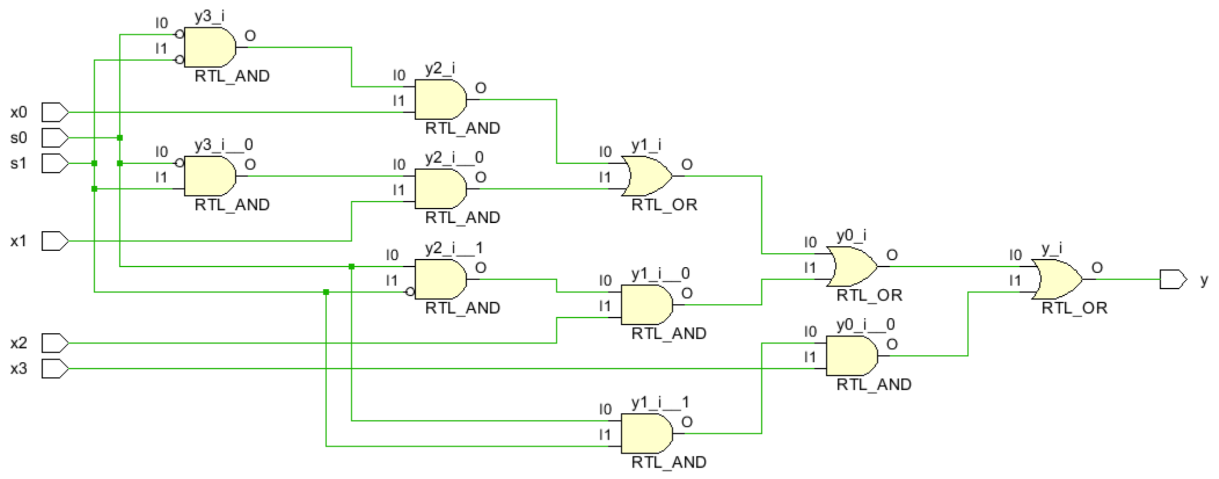
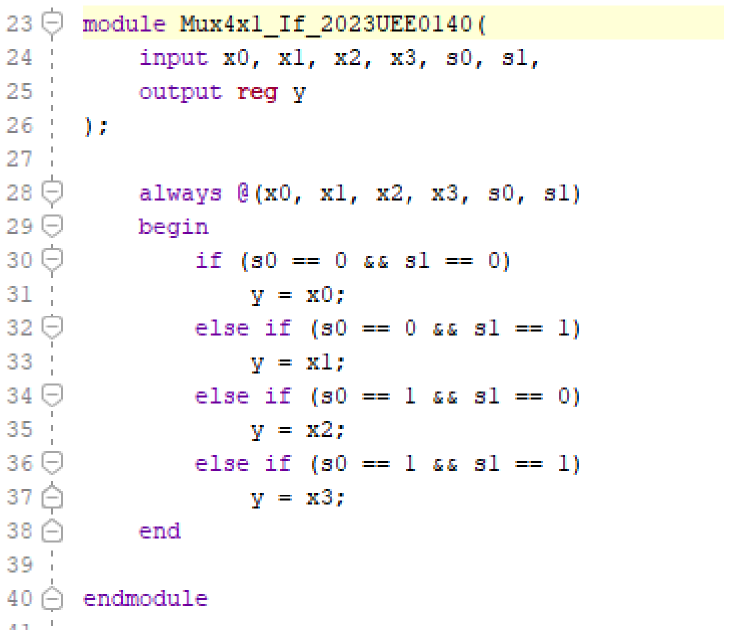


Figure 26: Schematic



```
module Mux4x1_If_2023UEE0140(
    input x0, x1, x2, x3, s0, s1,
    output reg y
);

    always @(x0, x1, x2, x3, s0, s1)
    begin
        if (s0 == 0 && s1 == 0)
            y = x0;
        else if (s0 == 0 && s1 == 1)
            y = x1;
        else if (s0 == 1 && s1 == 0)
            y = x2;
        else if (s0 == 1 && s1 == 1)
            y = x3;
    end

endmodule
```

Figure 27: MUX using IF-Else

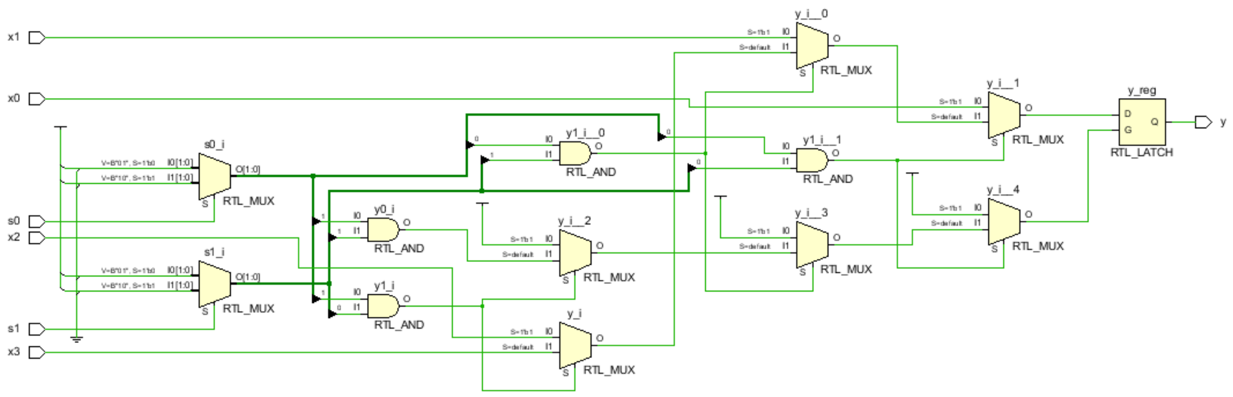
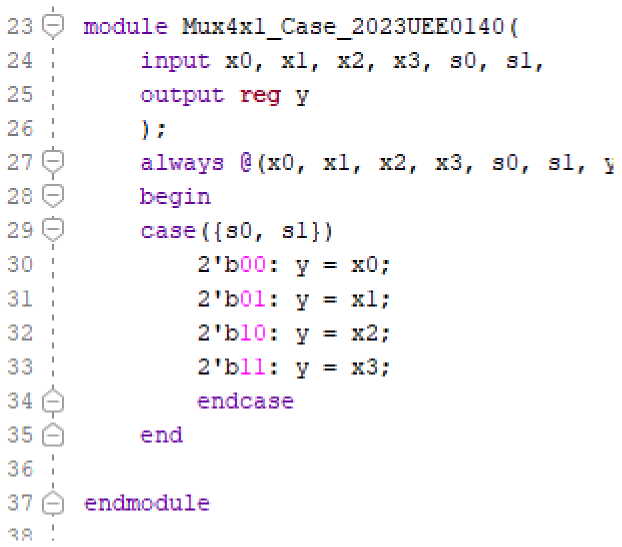


Figure 28: Schematic



```
module Mux4x1_Case_2023UEE0140(
    input x0, x1, x2, x3, s0, s1,
    output reg y
    );
    always @(x0, x1, x2, x3, s0, s1, y
    begin
    case({s0, s1})
        2'b00: y = x0;
        2'b01: y = x1;
        2'b10: y = x2;
        2'b11: y = x3;
        endcase
    end

endmodule
```

Figure 29: MUX using Case statement

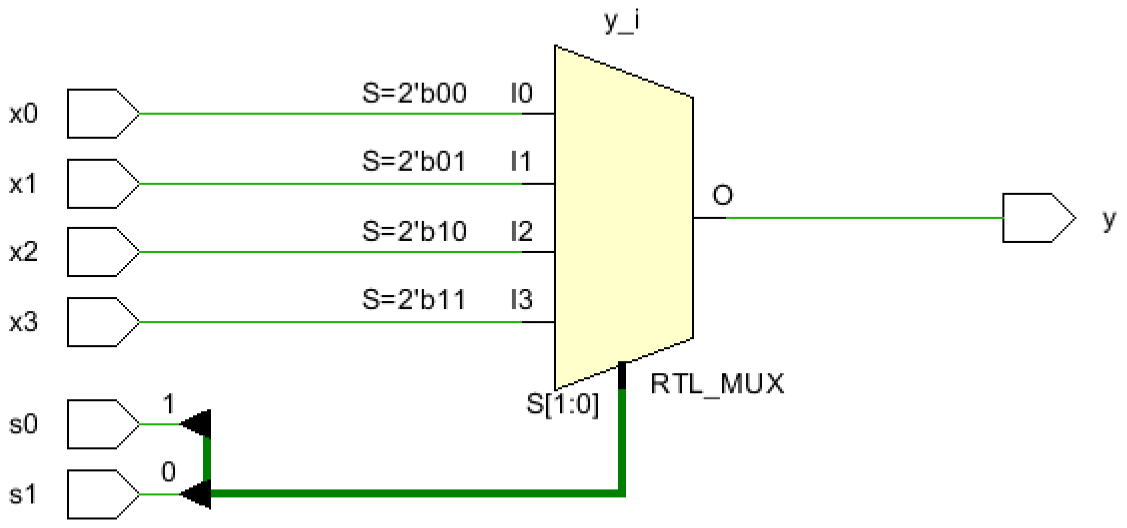


Figure 30: Schematic

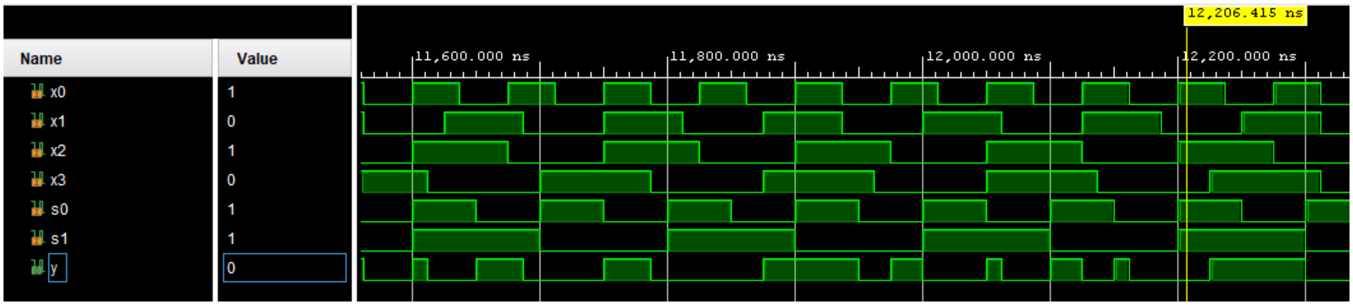
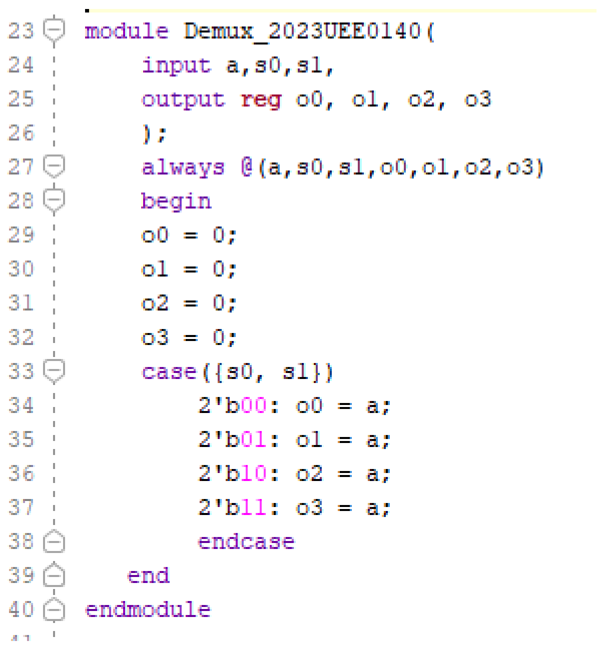


Figure 31: Schematic

Aim: To implement DEMUX

The aim of this section is to design a Demultiplexer (DEMUX), which is a device that takes a single input and routes it to one of several outputs based on the value of control signals. This implementation is crucial for understanding how data can be selectively routed in digital systems.



```
module Demux_2023UEE0140(
    input a,s0,s1,
    output reg o0, o1, o2, o3
    );
    always @(a,s0,s1,o0,o1,o2,o3)
    begin
    o0 = 0;
    o1 = 0;
    o2 = 0;
    o3 = 0;
    case({s0, s1})
        2'b00: o0 = a;
        2'b01: o1 = a;
        2'b10: o2 = a;
        2'b11: o3 = a;
        endcase
   end
endmodule
```

Figure 32: Demux Verilog Code

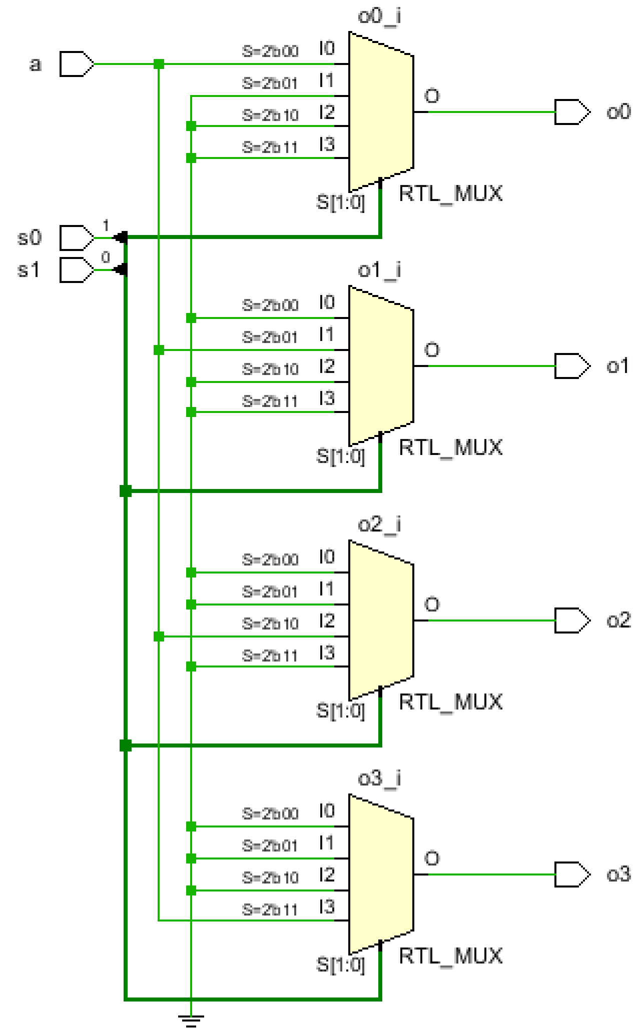
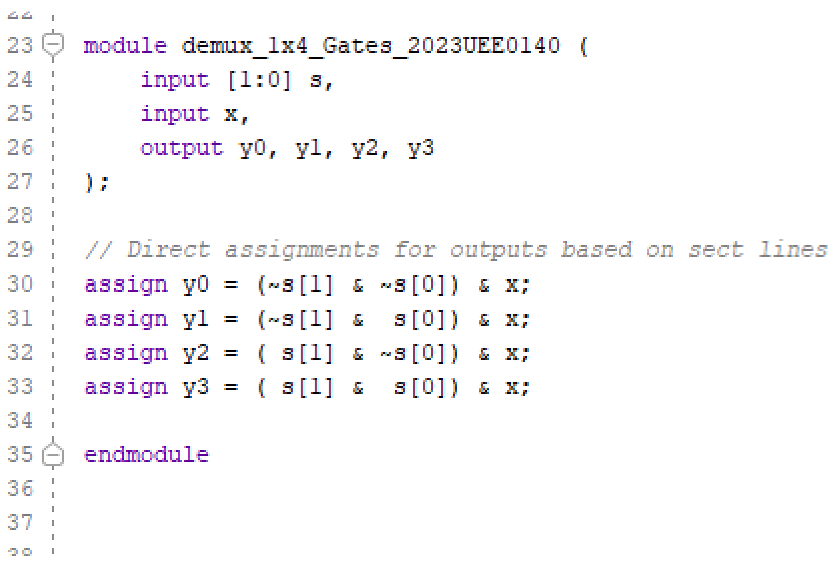


Figure 33: Schematic



```
module demux_1x4_Gates_2023UEE0140 (
    input [1:0] s,
    input x,
    output y0, y1, y2, y3
);

// Direct assignments for outputs based on sect lines
assign y0 = (~s[1] & ~s[0]) & x;
assign y1 = (~s[1] &  s[0]) & x;
assign y2 = ( s[1] & ~s[0]) & x;
assign y3 = ( s[1] &  s[0]) & x;

endmodule
```

Figure 34: Demux using Logic Gates

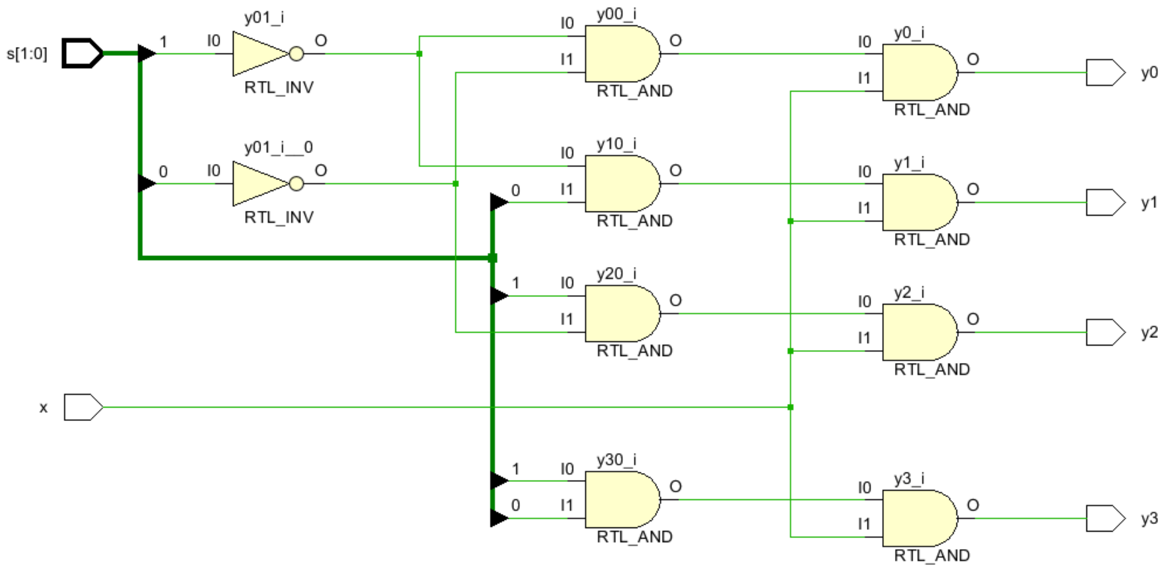
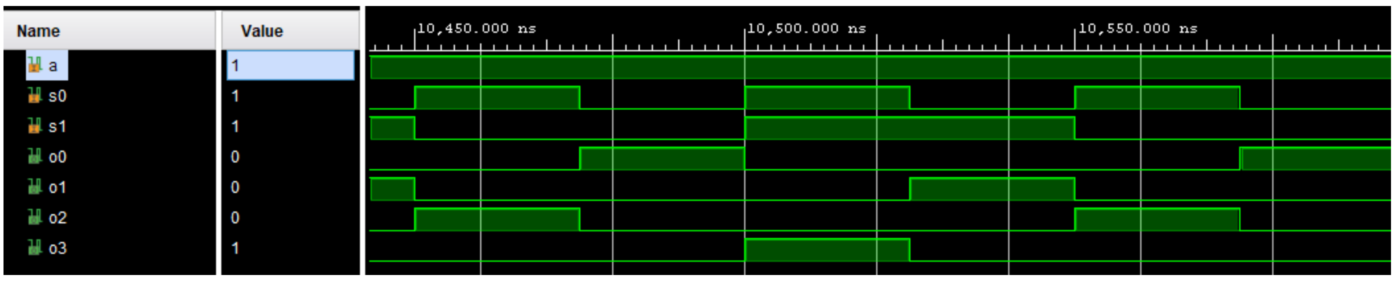


Figure 35: Schematic

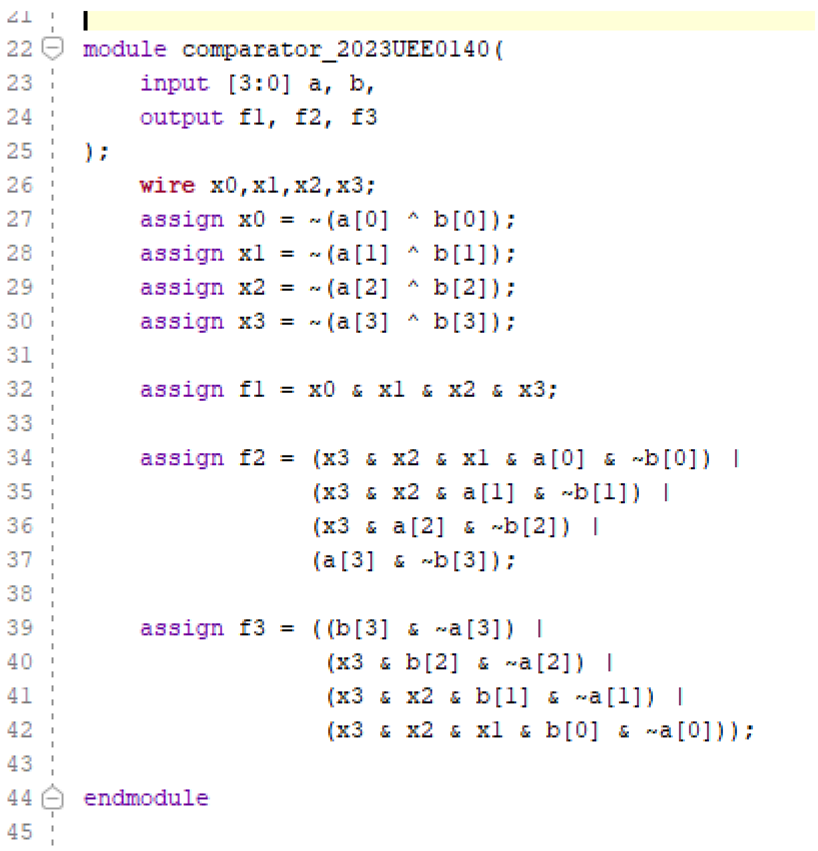


Name	Value
a	1
s0	1
s1	1
o0	0
o1	0
o2	0
o3	1

Figure 36: Simulation

Aim: To implement a Four Bit Comparator

In this section, we implement a four-bit comparator, which is a digital circuit that compares two binary numbers and determines their relative magnitude (greater than, less than, or equal to). This is essential in digital systems for decision-making processes.



```verilog

module comparator_2023UEE0140(
    input [3:0] a, b,
    output f1, f2, f3
);
    wire x0,x1,x2,x3;
    assign x0 = ~(a[0] ^ b[0]);
    assign x1 = ~(a[1] ^ b[1]);
    assign x2 = ~(a[2] ^ b[2]);
    assign x3 = ~(a[3] ^ b[3]);

    assign f1 = x0 & x1 & x2 & x3;

    assign f2 = (x3 & x2 & x1 & a[0] & ~b[0]) |
                (x3 & x2 & a[1] & ~b[1]) |
                (x3 & a[2] & ~b[2]) |
                (a[3] & ~b[3]);

    assign f3 = ((b[3] & ~a[3]) |
                 (x3 & b[2] & ~a[2]) |
                 (x3 & x2 & b[1] & ~a[1]) |
                 (x3 & x2 & x1 & b[0] & ~a[0]));

endmodule

```

Figure 37: Comparator using Gates

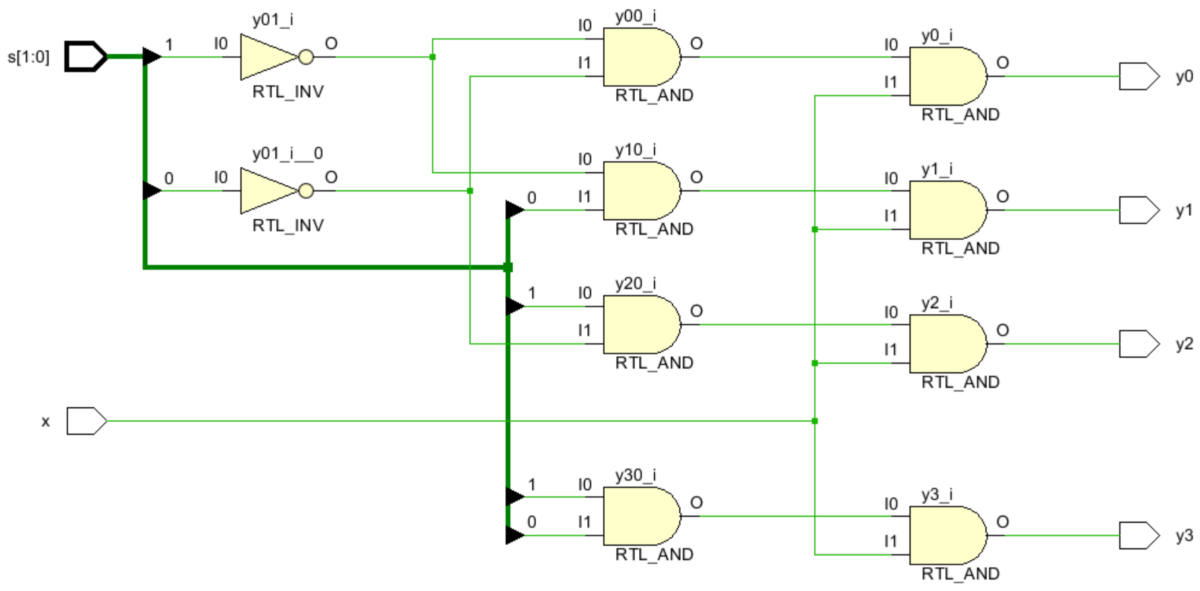
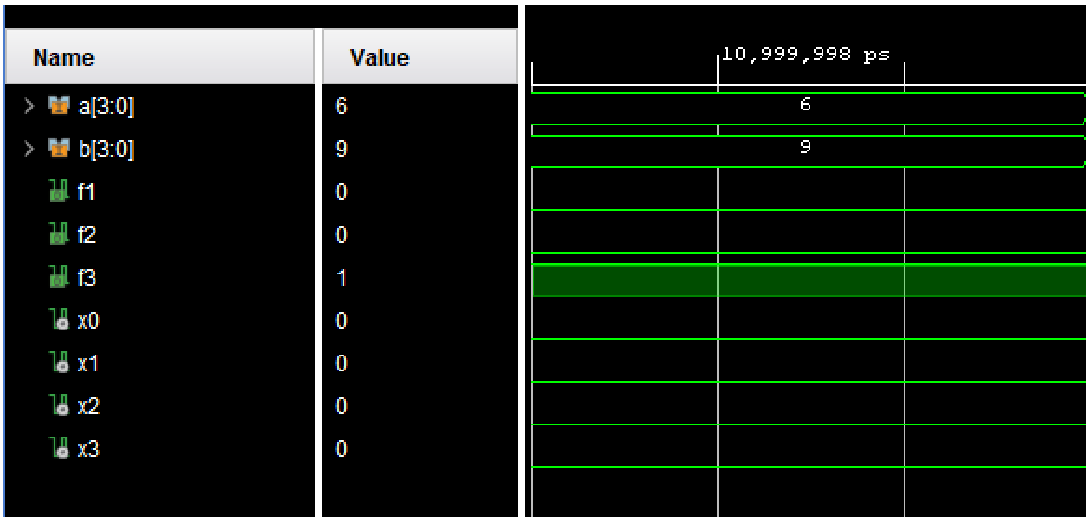
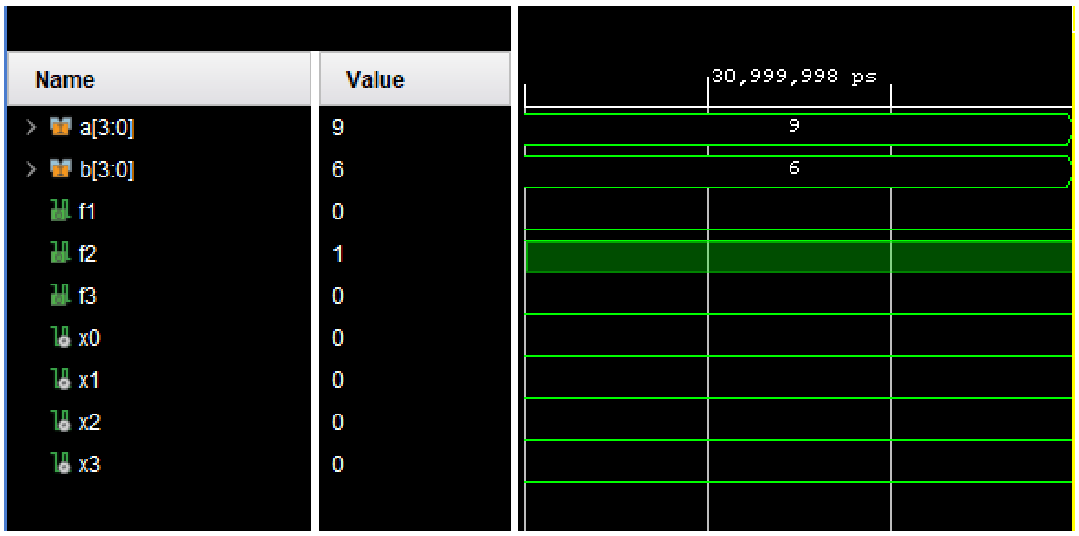


Figure 38: Schematic



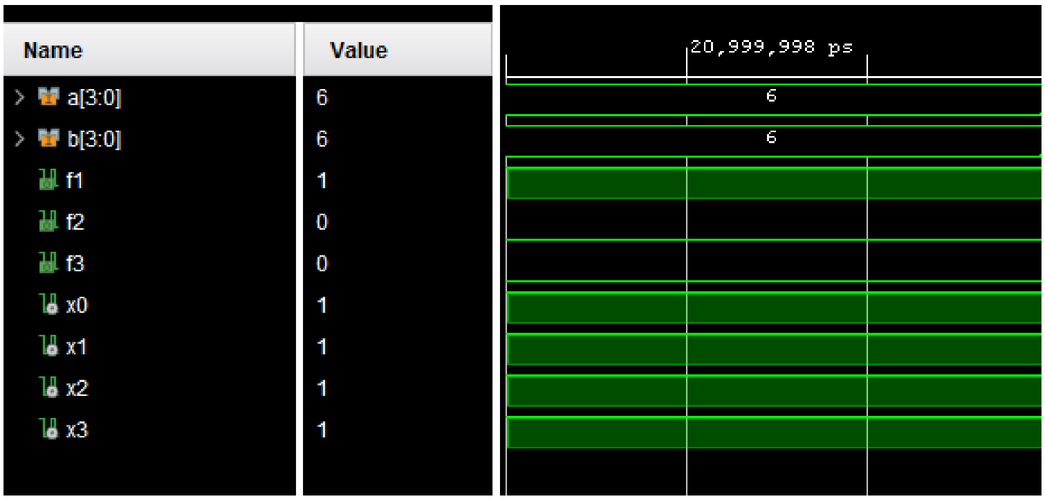
Name	Value
a[3:0]	6
b[3:0]	9
f1	0
f2	0
f3	1
x0	0
x1	0
x2	0
x3	0

Figure 39: Simulation for b > a



Name	Value
a[3:0]	9
b[3:0]	6
f1	0
f2	1
f3	0
x0	0
x1	0
x2	0
x3	0

Figure 40: Simulation for b < a

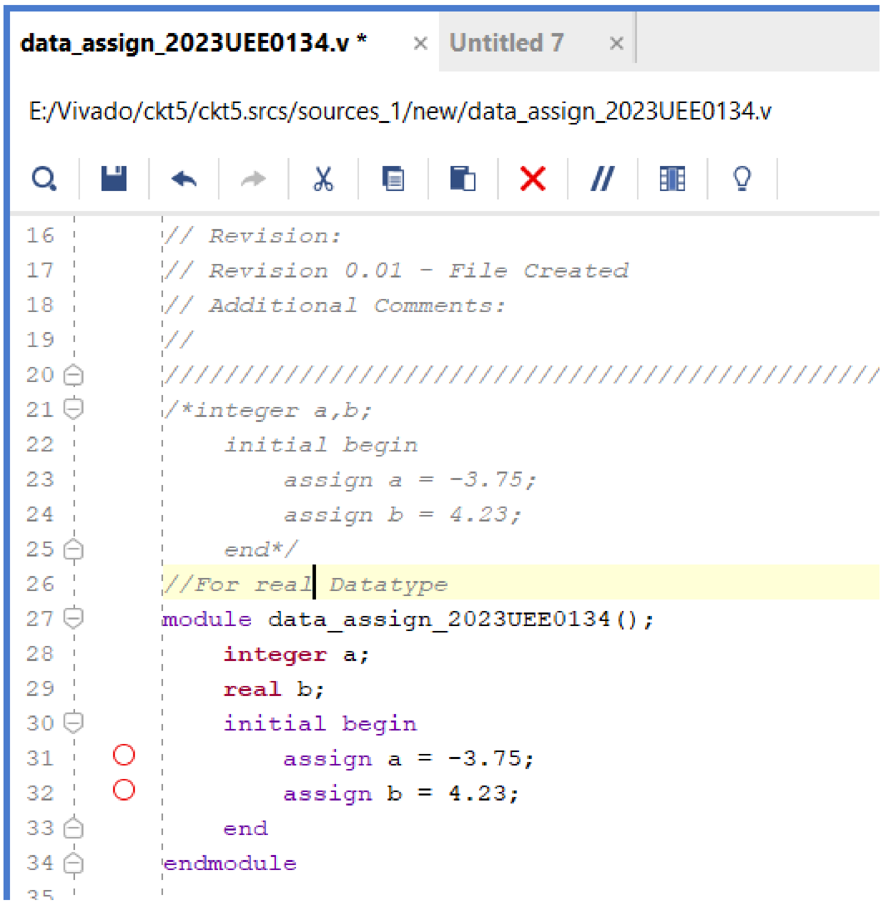


Name	Value
a[3:0]	6
b[3:0]	6
f1	1
f2	0
f3	0
x0	1
x1	1
x2	1
x3	1

Figure 41: Simulation for b =a

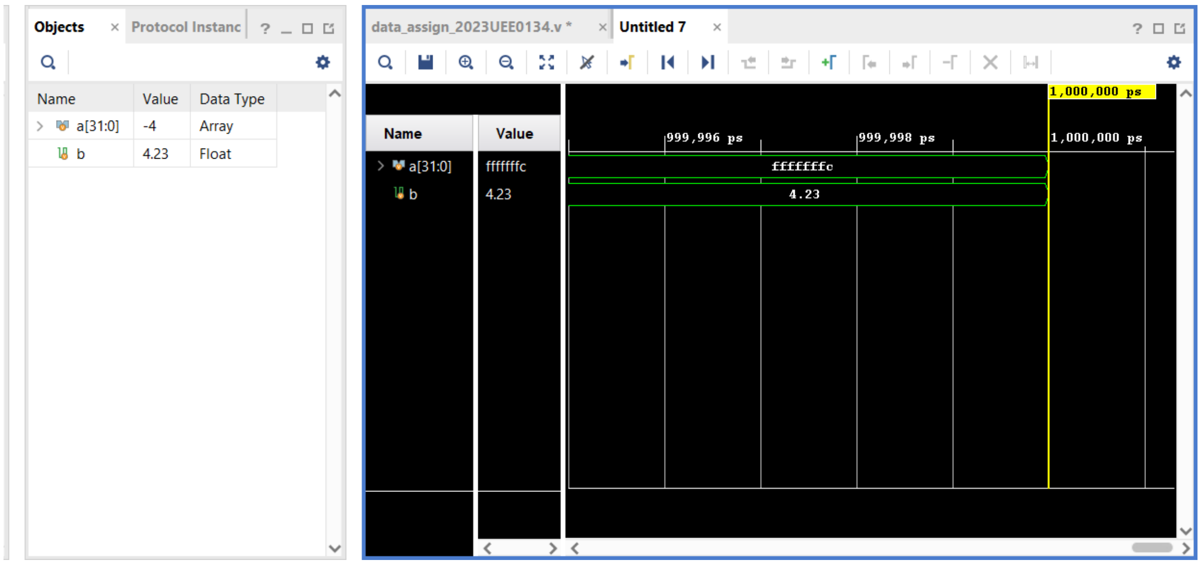
Conclusion

The implemented codes and designs for basic logic gates, multiplexers, demultiplexers,half adders,full adders and comparators demonstrated the versatility and effectiveness of Verilog in digital system design. Each Combinational logic circuit highlighted the significance of various design approaches, enabling the successful execution of complex logic functions essential for digital circuits.


data_assign_2023UEE0134.v *
Untitled 7
E:/Vivado/ckt5/ckt5.srcs/sources_1/new/data_assign_2023UEE0134.v

```
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////
/*integer a,b;
    initial begin
        assign a = -3.75;
        assign b = 4.23;
    end*/
//For real Datatype
module data_assign_2023UEE0134();
    integer a;
    real b;
    initial begin
        assign a = -3.75;
        assign b = 4.23;
    end
endmodule
```

Figure 16: Real Data Type



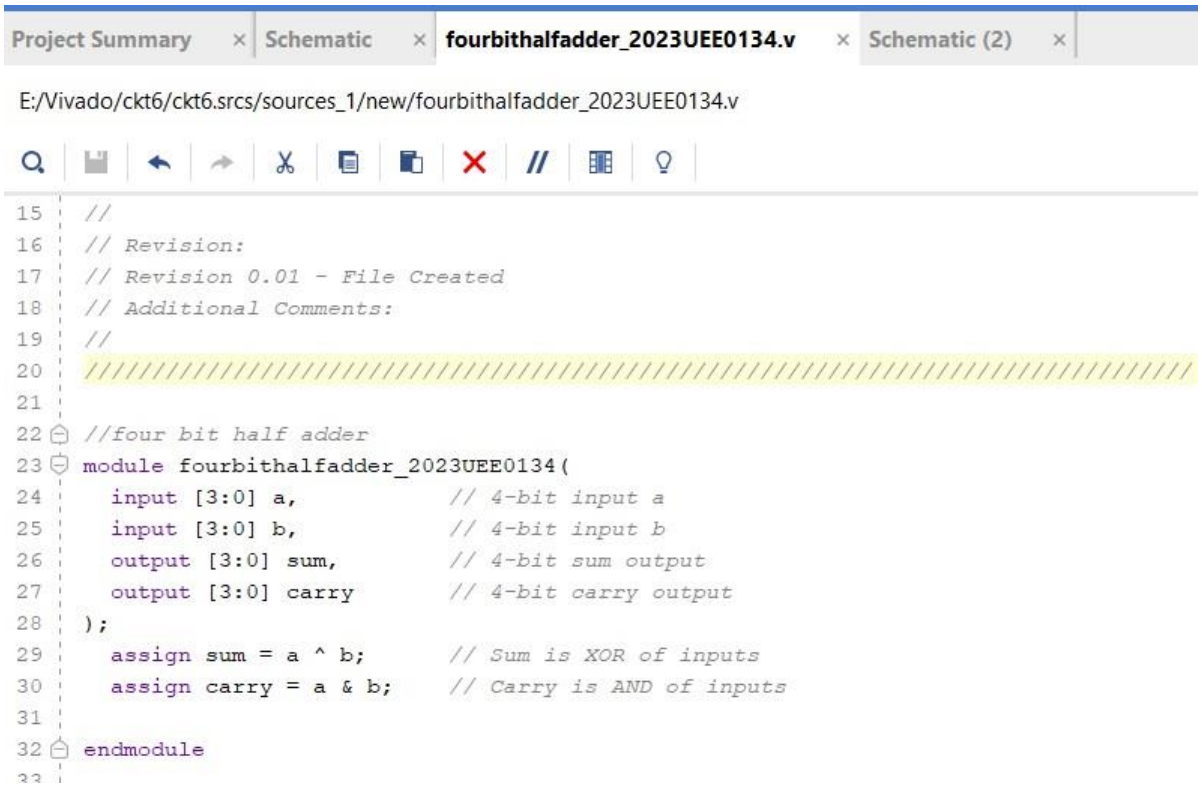
Name	Value	Data Type
a[31:0]	-4	Array
b	4.23	Float

Name	Value
a[31:0]	fffffffc
b	4.23

Figure 17: Simulation

Aim: To implement Four Bit Half Adder

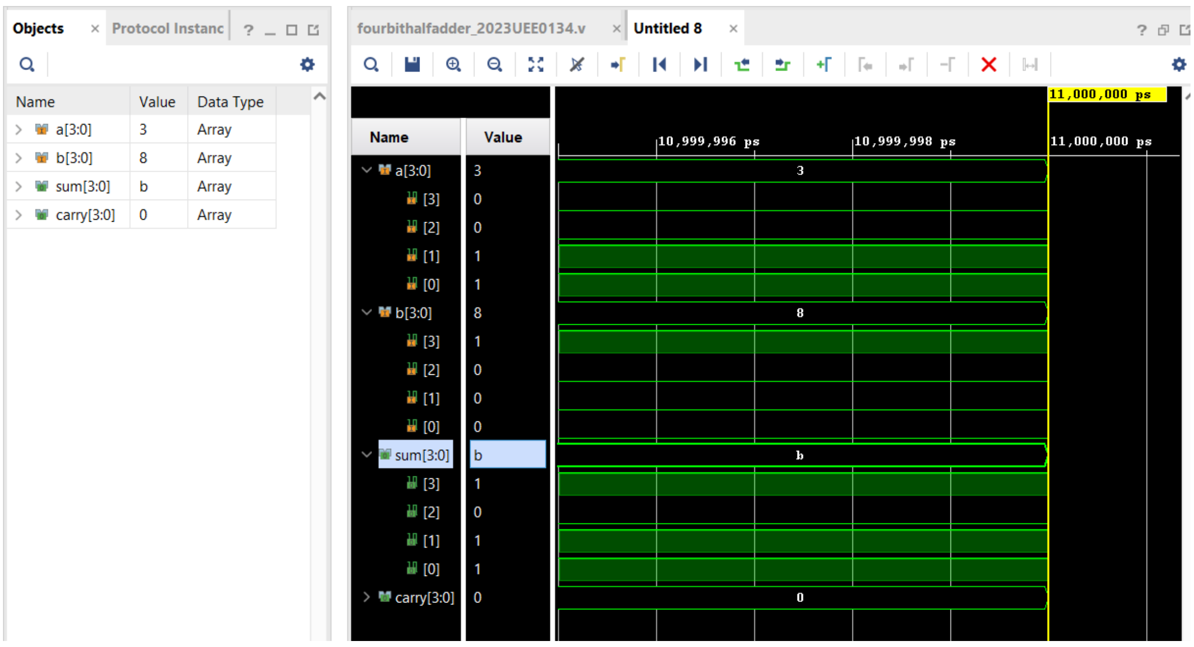
In this section, we implement a four-bit half adder, which extends the concept of a basic half adder to handle multi-bit inputs.



E:/Vivado/ckt6/ckt6.srcs/sources_1/new/fourbithalfadder_2023UEE0134.v

```
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

//four bit half adder
module fourbithalfadder_2023UEE0134(
    input [3:0] a,          // 4-bit input a
    input [3:0] b,          // 4-bit input b
    output [3:0] sum,       // 4-bit sum output
    output [3:0] carry      // 4-bit carry output
);
    assign sum = a ^ b;     // Sum is XOR of inputs
    assign carry = a & b;   // Carry is AND of inputs

endmodule
```

Figure 18: Verilog code for Half Adder



Name	Value	Data Type
a[3:0]	3	Array
b[3:0]	8	Array
sum[3:0]	b	Array
carry[3:0]	0	Array

Name	Value
a[3:0]	3
[3]	0
[2]	0
[1]	1
[0]	1
b[3:0]	8
[3]	1
[2]	0
[1]	0
[0]	0
sum[3:0]	b
[3]	1
[2]	0
[1]	1
[0]	1
carry[3:0]	0

Figure 19: Simulation

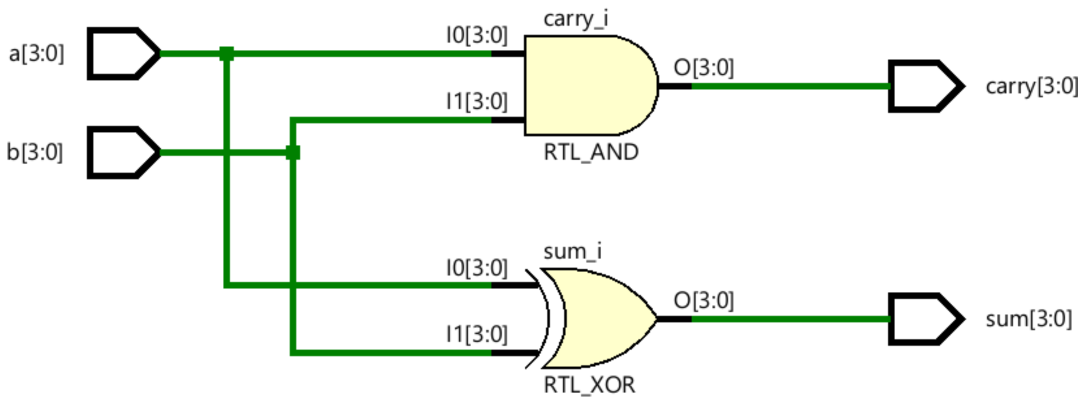
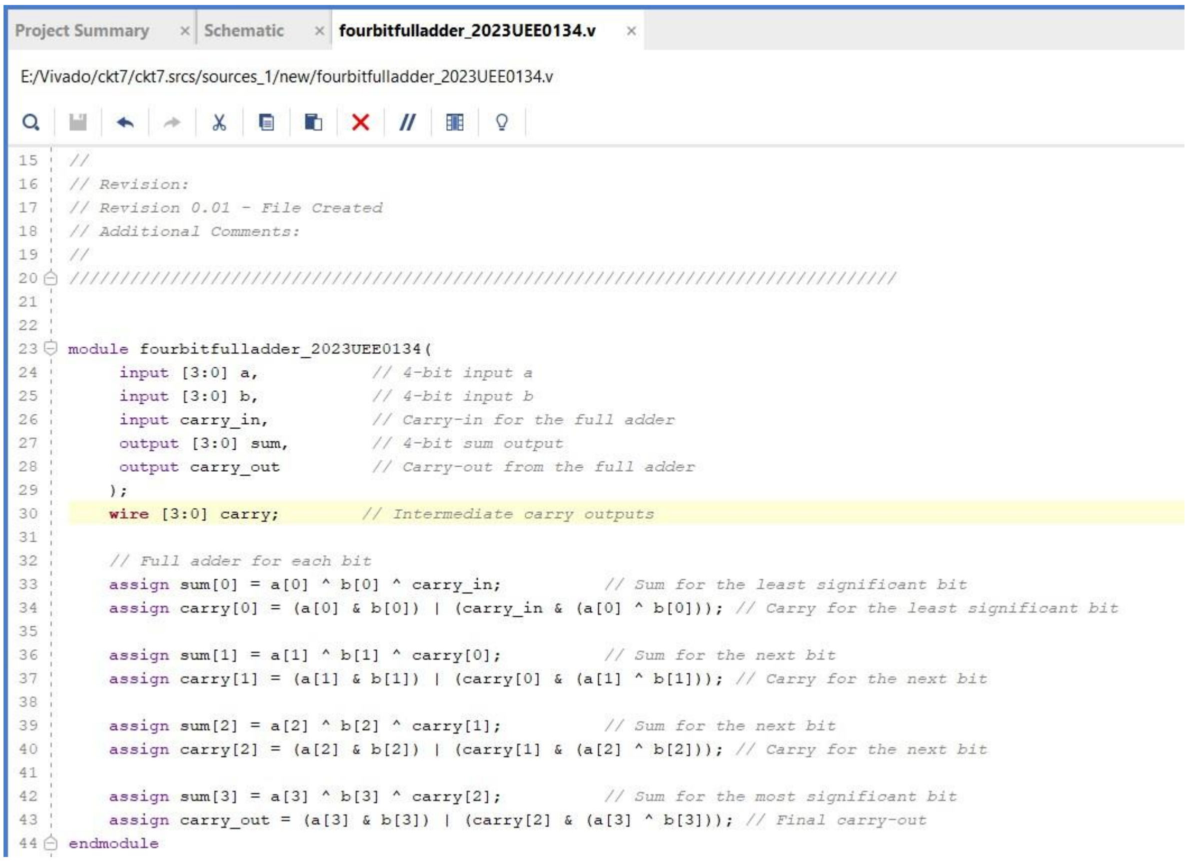


Figure 20: Schematic

Aim: To implement Four Bit Full Adder

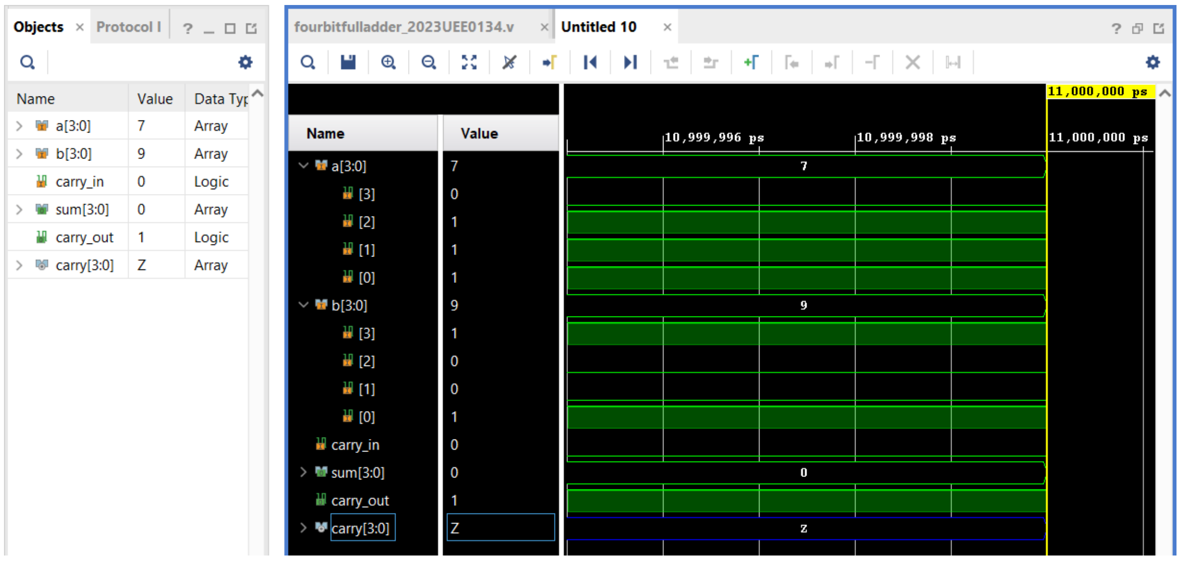
This section describes the implementation of a four-bit full adder, which adds two four-bit binary numbers and produces a sum and carry output.



E:/Vivado/ckt7/ckt7.srcs/sources_1/new/fourbitfulladder_2023UEE0134.v

```
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module fourbitfulladder_2023UEE0134(
     input [3:0] a,          // 4-bit input a
     input [3:0] b,          // 4-bit input b
     input carry_in,         // Carry-in for the full adder
     output [3:0] sum,       // 4-bit sum output
     output carry_out        // Carry-out from the full adder
    );
    wire [3:0] carry;        // Intermediate carry outputs

    // Full adder for each bit
    assign sum[0] = a[0] ^ b[0] ^ carry_in;          // Sum for the least significant bit
    assign carry[0] = (a[0] & b[0]) | (carry_in & (a[0] ^ b[0])); // Carry for the least significant bit

    assign sum[1] = a[1] ^ b[1] ^ carry[0];          // Sum for the next bit
    assign carry[1] = (a[1] & b[1]) | (carry[0] & (a[1] ^ b[1])); // Carry for the next bit

    assign sum[2] = a[2] ^ b[2] ^ carry[1];          // Sum for the next bit
    assign carry[2] = (a[2] & b[2]) | (carry[1] & (a[2] ^ b[2])); // Carry for the next bit

    assign sum[3] = a[3] ^ b[3] ^ carry[2];          // Sum for the most significant bit
    assign carry_out = (a[3] & b[3]) | (carry[2] & (a[3] ^ b[3])); // Final carry-out
endmodule
```

Figure 21: Verilog code for Full Adder



Objects | Protocol I

Name	Value	Data Typ
a[3:0]	7	Array
b[3:0]	9	Array
carry_in	0	Logic
sum[3:0]	0	Array
carry_out	1	Logic
carry[3:0]	Z	Array

fourbitfulladder_2023UEE0134.v | Untitled 10

11,000,000 ps

10,999,996 ps | 10,999,998 ps | 11,000,000 ps

Name	Value
a[3:0]	7
[3]	0
[2]	1
[1]	1
[0]	1
b[3:0]	9
[3]	1
[2]	0
[1]	0
[0]	1
carry_in	0
sum[3:0]	0
carry_out	1
carry[3:0]	Z

Figure 22: Simulation

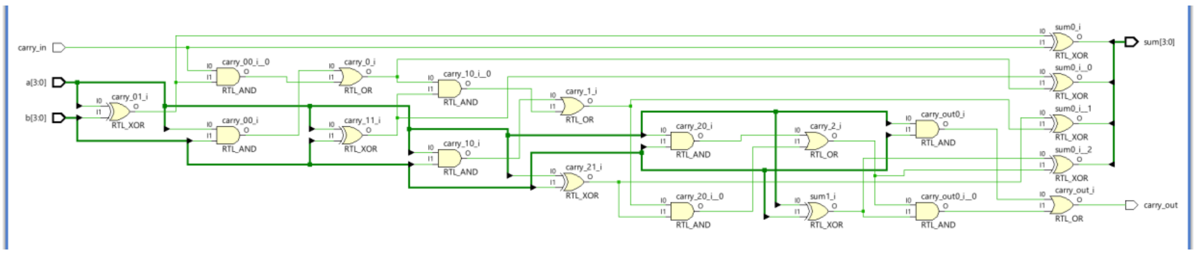
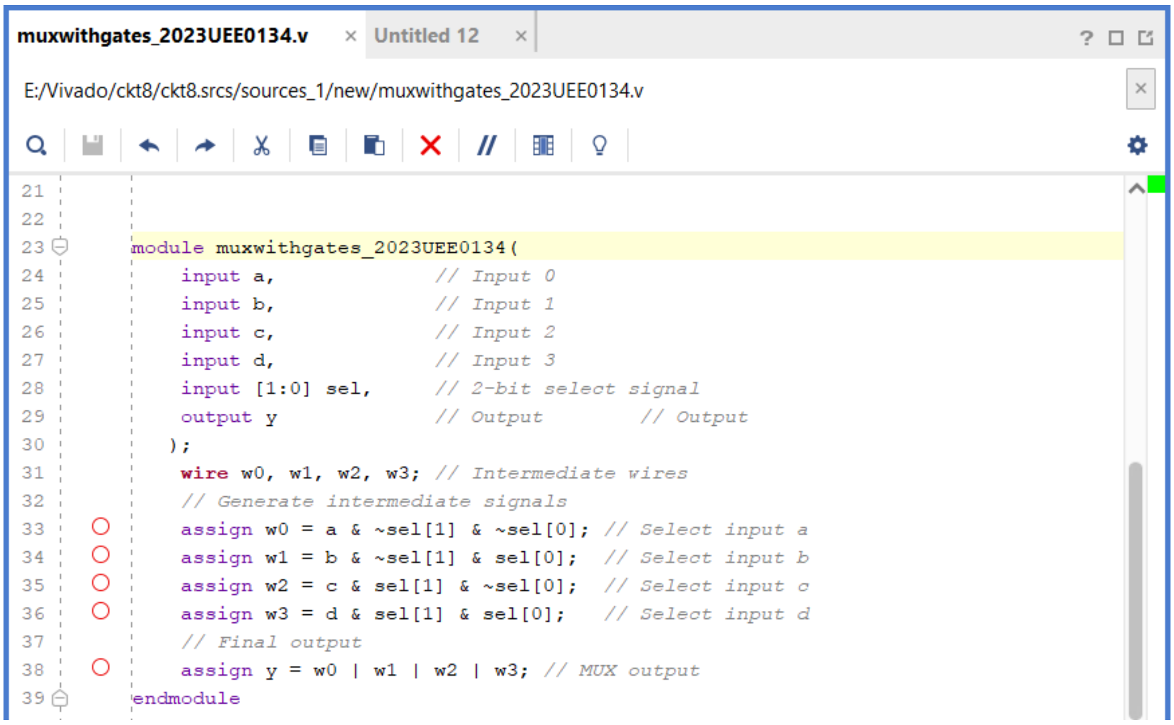


Figure 23: Schematic

Aim: To implement 4 X 1 MUX with Gates

In this section, we implement a 4-to-1 multiplexer (MUX) using basic gates. A MUX selects one of several input signals and forwards the selected input to a single output line.

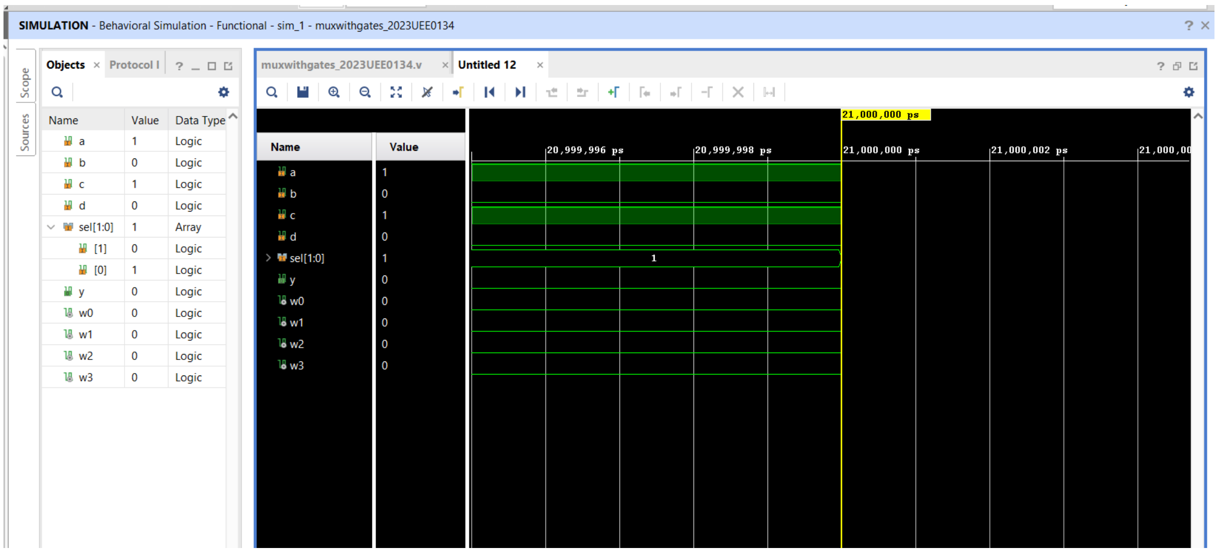


muxwithgates_2023UEE0134.v × Untitled 12 ×

E:/Vivado/ckt8/ckt8.srcs/sources_1/new/muxwithgates_2023UEE0134.v

```verilog


module muxwithgates_2023UEE0134(
    input a,            // Input 0
    input b,            // Input 1
    input c,            // Input 2
    input d,            // Input 3
    input [1:0] sel,    // 2-bit select signal
    output y            // Output          // Output
   );
    wire w0, w1, w2, w3; // Intermediate wires
    // Generate intermediate signals
    assign w0 = a & ~sel[1] & ~sel[0]; // Select input a
    assign w1 = b & ~sel[1] & sel[0];  // Select input b
    assign w2 = c & sel[1] & ~sel[0];  // Select input c
    assign w3 = d & sel[1] & sel[0];   // Select input d
    // Final output
    assign y = w0 | w1 | w2 | w3; // MUX output
endmodule
```

Figure 24: Verilog Code for 4 X 1 MUX



SIMULATION - Behavioral Simulation - Functional - sim_1 - muxwithgates_2023UEE0134

Objects × Protocol I

Name	Value	Data Type
a	1	Logic
b	0	Logic
c	1	Logic
d	0	Logic
sel[1:0]	1	Array
[1]	0	Logic
[0]	1	Logic
y	0	Logic
w0	0	Logic
w1	0	Logic
w2	0	Logic
w3	0	Logic

muxwithgates_2023UEE0134.v × Untitled 12 ×

Name	Value
a	1
b	0
c	1
d	0
sel[1:0]	1
y	0
w0	0
w1	0
w2	0
w3	0

Figure 25: Simulation

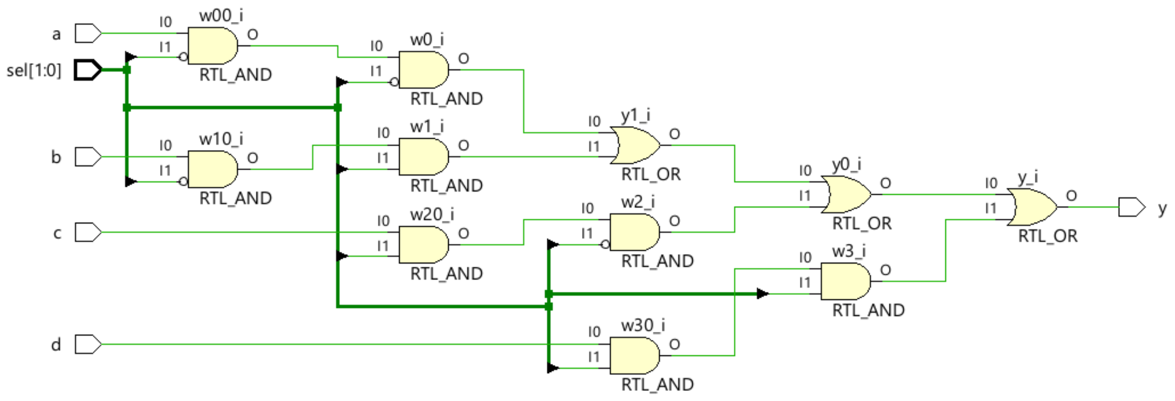
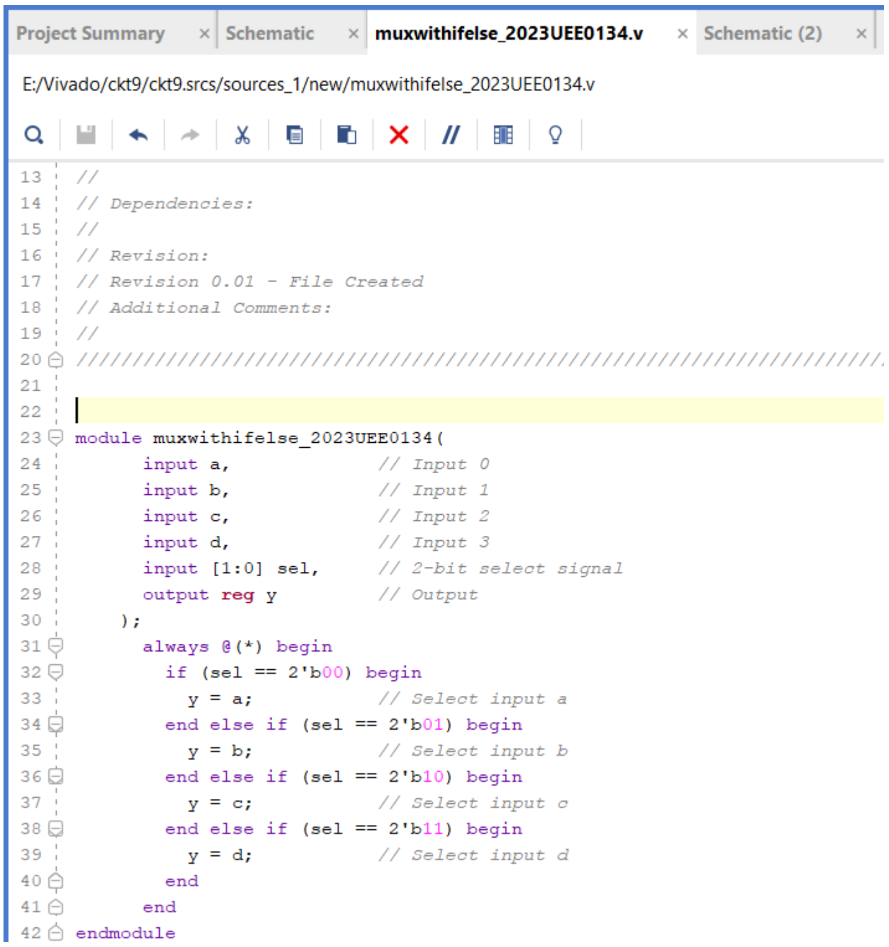


Figure 26: Schematic

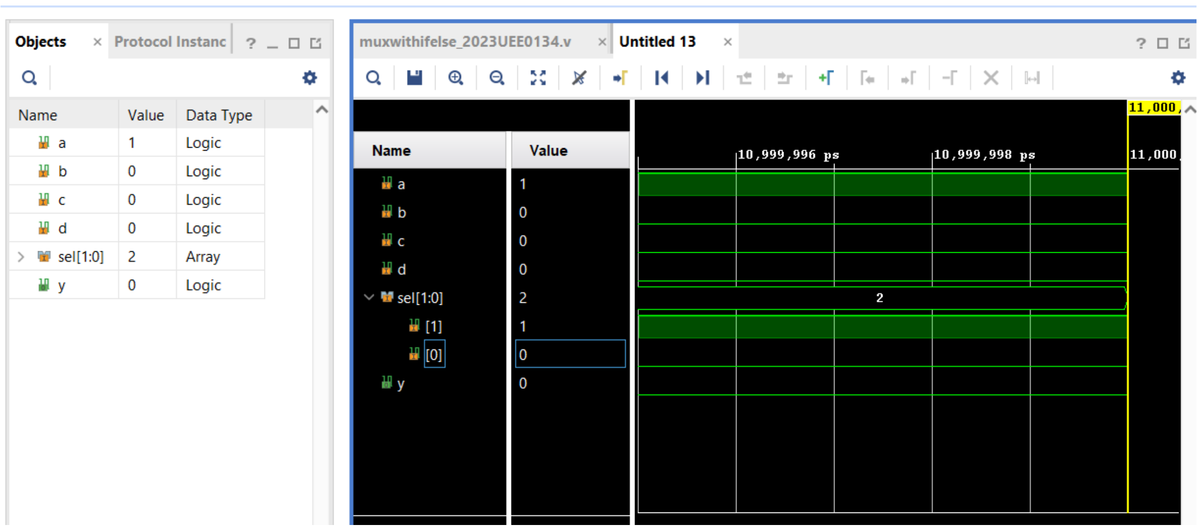
Aim: To implement 4 X 1 MUX with If Else Statements

This section implements a 4-to-1 multiplexer using if-else statements, showcasing an alternative approach to MUX design.


Project Summary
Schematic
muxwithifelse_2023UEE0134.v
Schematic (2)
E:/Vivado/ckt9/ckt9.srcs/sources_1/new/muxwithifelse_2023UEE0134.v

```
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
///////////////////////////////////////////////////////////////////////


module muxwithifelse_2023UEE0134(
        input a,                // Input 0
        input b,                // Input 1
        input c,                // Input 2
        input d,                // Input 3
        input [1:0] sel,        // 2-bit select signal
        output reg y            // Output
    );
        always @(*) begin
          if (sel == 2'b00) begin
            y = a;              // Select input a
          end else if (sel == 2'b01) begin
            y = b;              // Select input b
          end else if (sel == 2'b10) begin
            y = c;              // Select input c
          end else if (sel == 2'b11) begin
            y = d;              // Select input d
          end
        end
endmodule
```

Figure 27: Verilog Code for 4X1 MUX with if-else statements



Name	Value	Data Type
a	1	Logic
b	0	Logic
c	0	Logic
d	0	Logic
sel[1:0]	2	Array
y	0	Logic

Name	Value
a	1
b	0
c	0
d	0
sel[1:0]	2
[1]	1
[0]	0
y	0

Figure 28: Simulation

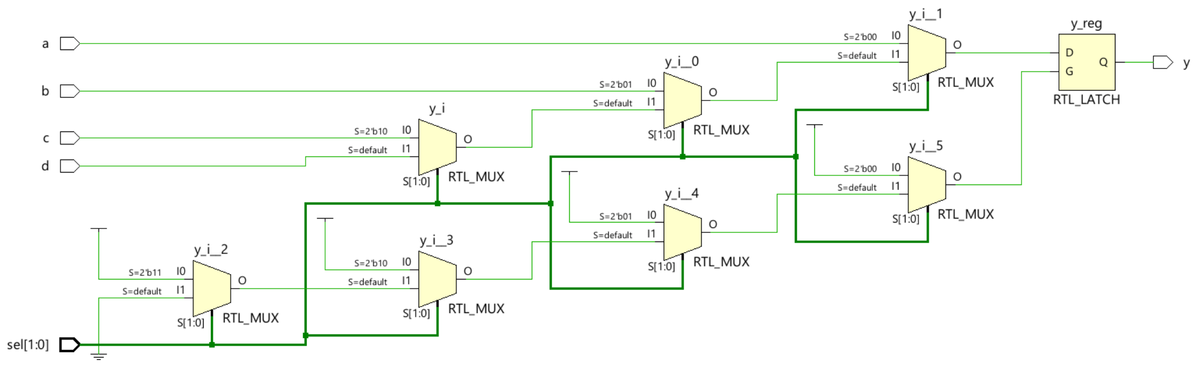
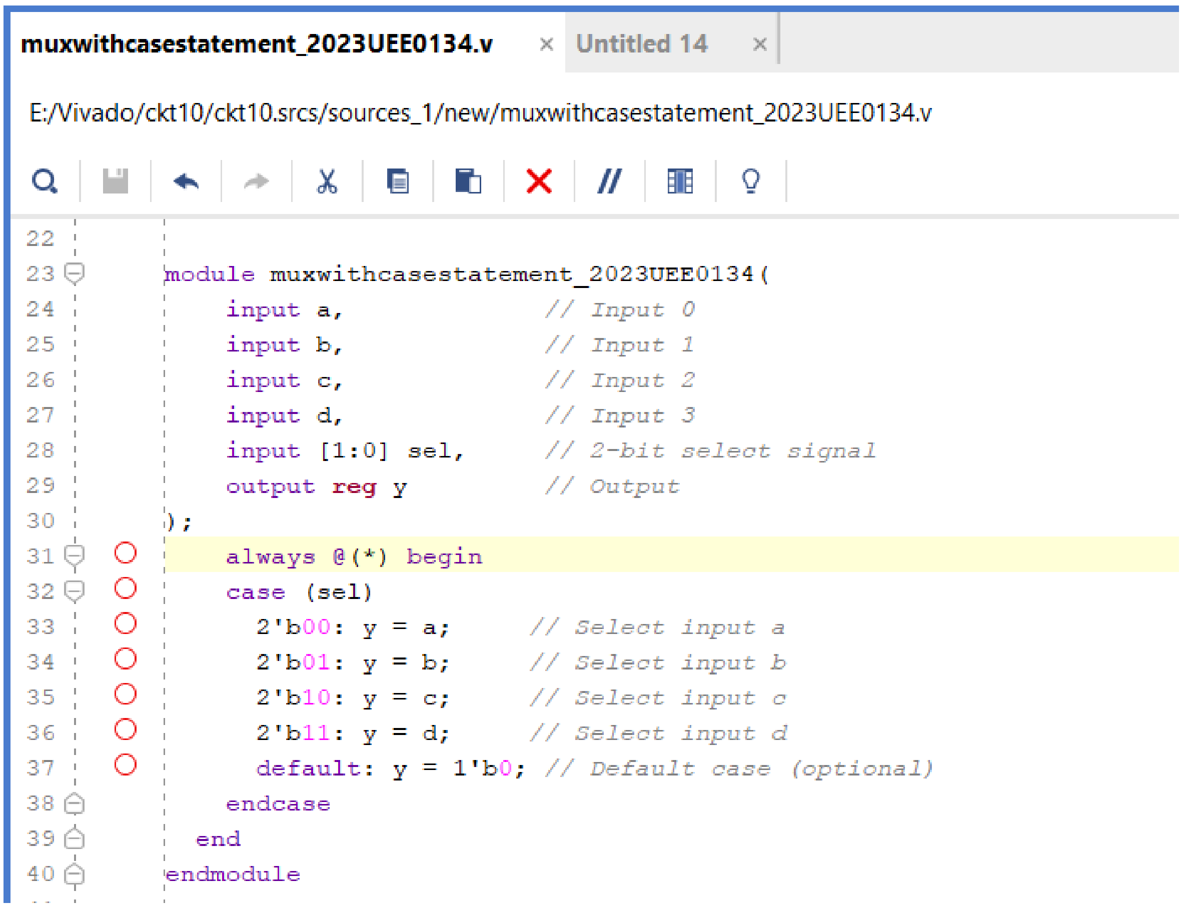


Figure 29: Schematic

Aim: To implement 4 X 1 MUX with Case Statement

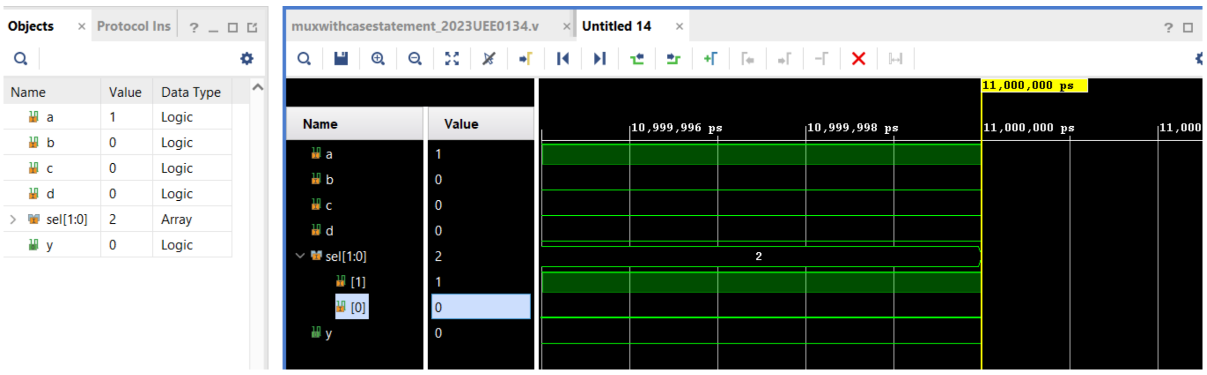
In this section, we design a 4-to-1 multiplexer using a case statement, further diversifying the design approaches for the MUX.



E:/Vivado/ckt10/ckt10.srcs/sources_1/new/muxwithcasestatement_2023UEE0134.v

```verilog

module muxwithcasestatement_2023UEE0134(
    input a,              // Input 0
    input b,              // Input 1
    input c,              // Input 2
    input d,              // Input 3
    input [1:0] sel,      // 2-bit select signal
    output reg y          // Output
);
    always @(*) begin
    case (sel)
      2'b00: y = a;      // Select input a
      2'b01: y = b;      // Select input b
      2'b10: y = c;      // Select input c
      2'b11: y = d;      // Select input d
      default: y = 1'b0; // Default case (optional)
    endcase
  end
endmodule
```

Figure 30: Verilog Code for 4X1 MUX with case statement



Name	Value	Data Type
a	1	Logic
b	0	Logic
c	0	Logic
d	0	Logic
sel[1:0]	2	Array
y	0	Logic

Name	Value
a	1
b	0
c	0
d	0
sel[1:0]	2
[1]	1
[0]	0
y	0

Figure 31: Simulation

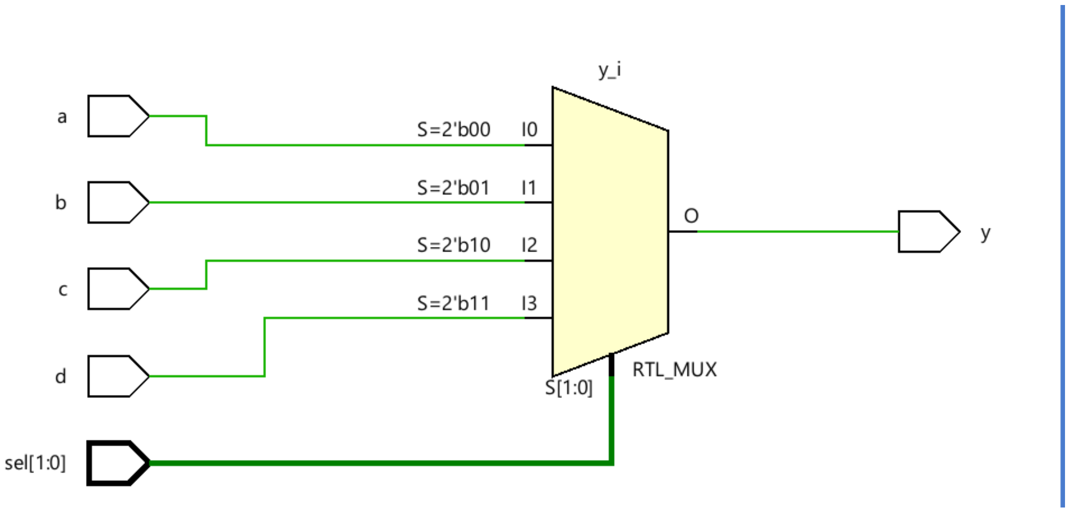
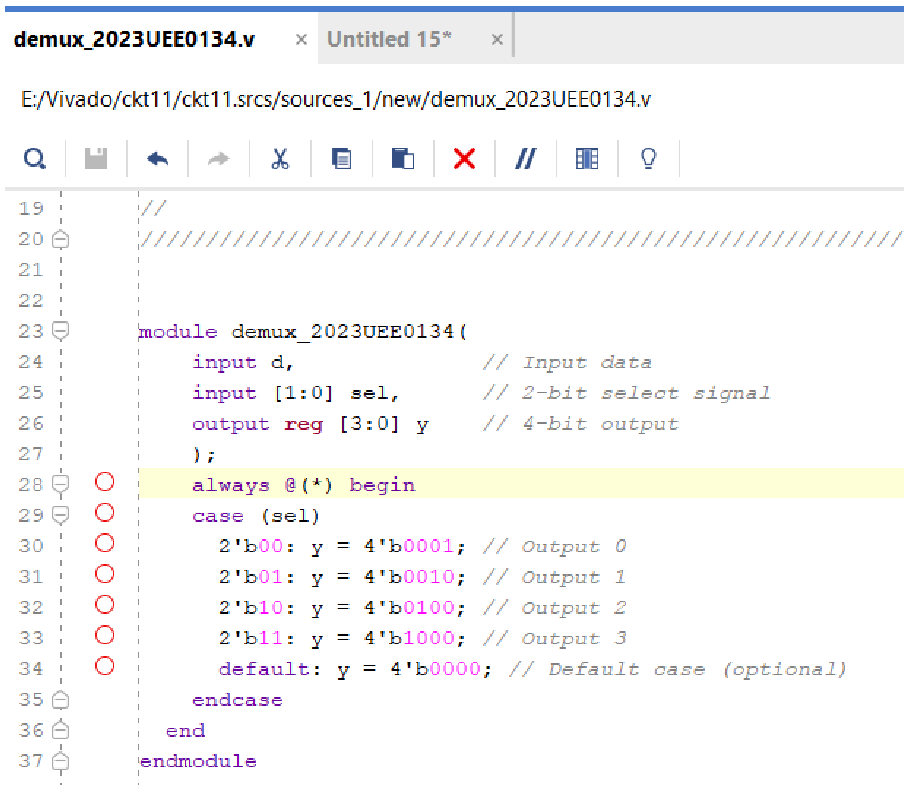


Figure 32: Schematic

Aim: To implement DEMUX

The aim of this section is to design a Demultiplexer (DEMUX), which is a device that takes a single input and routes it to one of several outputs based on the value of control signals. This implementation is crucial for understanding how data can be selectively routed in digital systems.

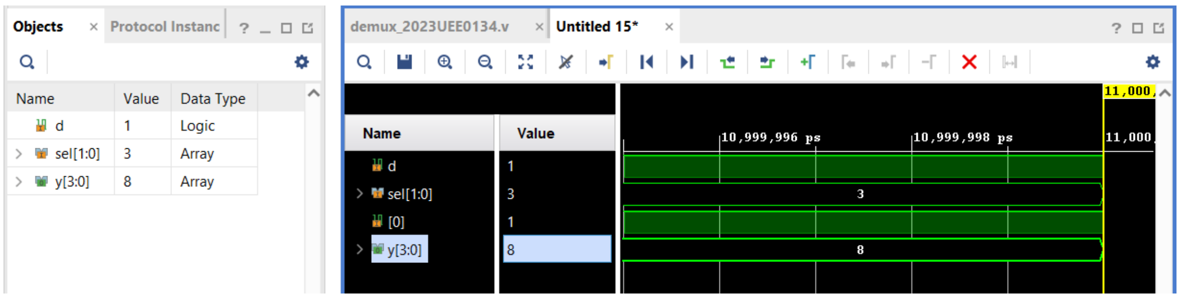


demux_2023UEE0134.v × Untitled 15* ×

E:/Vivado/ckt11/ckt11.srcs/sources_1/new/demux_2023UEE0134.v

```
//
/////////////////////////////////////////////////////////////


module demux_2023UEE0134(
    input d,              // Input data
    input [1:0] sel,      // 2-bit select signal
    output reg [3:0] y    // 4-bit output
    );
    always @(*) begin
    case (sel)
      2'b00: y = 4'b0001; // Output 0
      2'b01: y = 4'b0010; // Output 1
      2'b10: y = 4'b0100; // Output 2
      2'b11: y = 4'b1000; // Output 3
      default: y = 4'b0000; // Default case (optional)
    endcase
  end
endmodule
```

Figure 33: Verilog Code for Demux



Name	Value	Data Type
d	1	Logic
sel[1:0]	3	Array
y[3:0]	8	Array

Name	Value
d	1
sel[1:0]	3
[0]	1
y[3:0]	8

Figure 34: Simulation

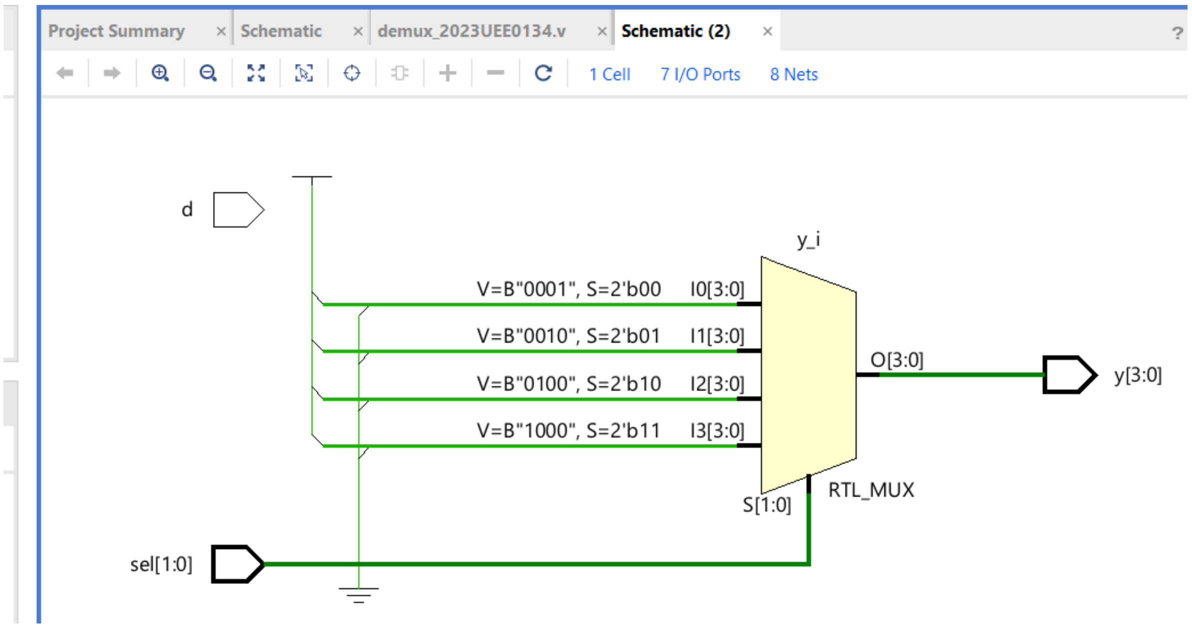
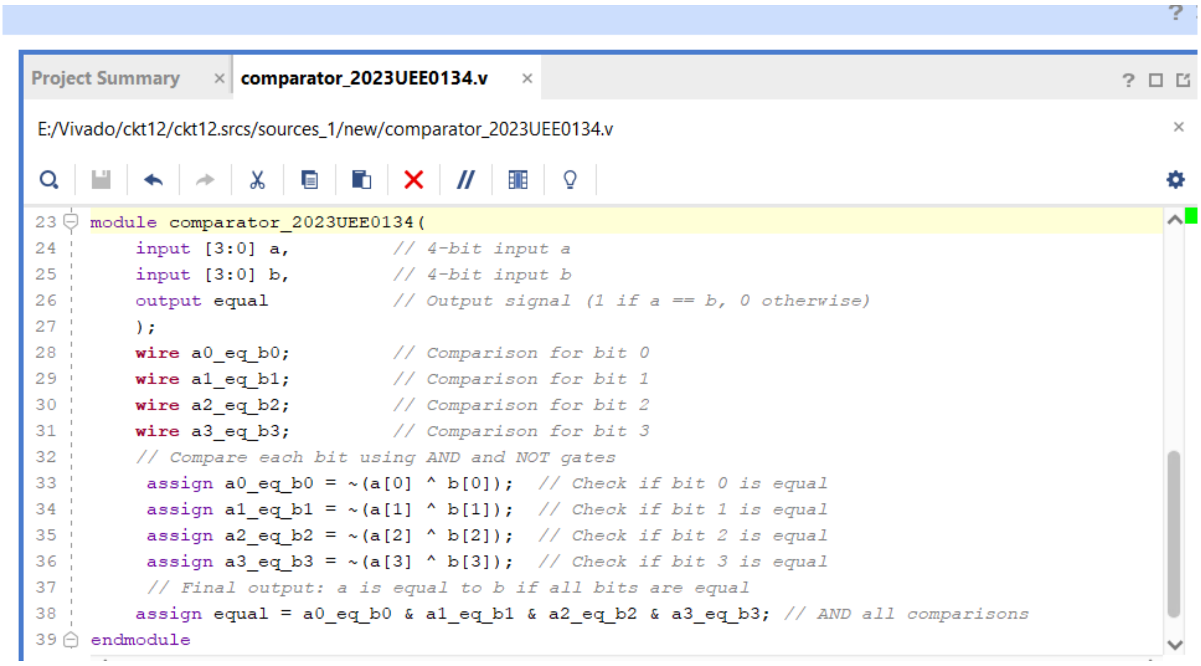


Figure 35: Schematic

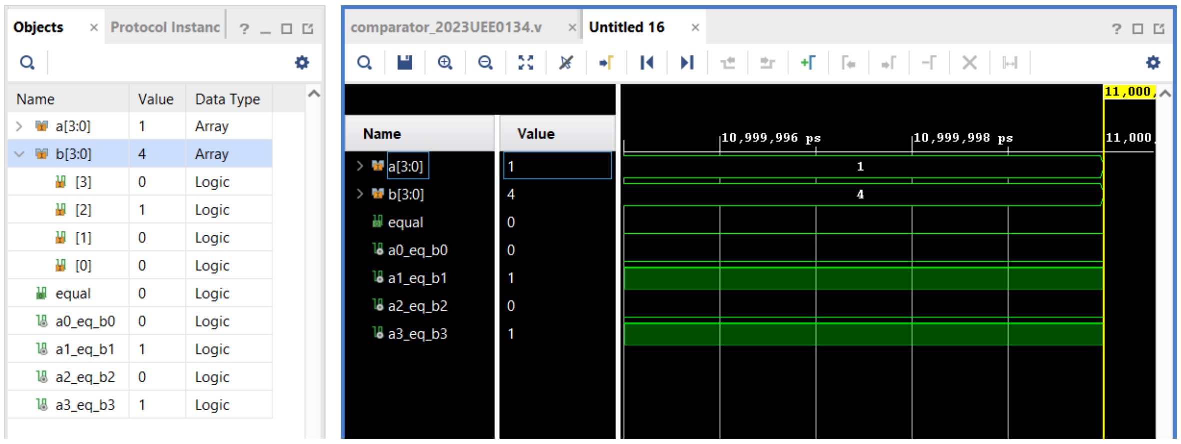
Aim: To implement a Four Bit Comparator

In this section, we implement a four-bit comparator, which is a digital circuit that compares two binary numbers and determines their relative magnitude (greater than, less than, or equal to). This is essential in digital systems for decision-making processes.


Project Summary
comparator_2023UEE0134.v
E:/Vivado/ckt12/ckt12.srcs/sources_1/new/comparator_2023UEE0134.v

```
module comparator_2023UEE0134(
    input [3:0] a,          // 4-bit input a
    input [3:0] b,          // 4-bit input b
    output equal            // Output signal (1 if a == b, 0 otherwise)
    );
    wire a0_eq_b0;          // Comparison for bit 0
    wire a1_eq_b1;          // Comparison for bit 1
    wire a2_eq_b2;          // Comparison for bit 2
    wire a3_eq_b3;          // Comparison for bit 3
    // Compare each bit using AND and NOT gates
     assign a0_eq_b0 = ~(a[0] ^ b[0]);  // Check if bit 0 is equal
     assign a1_eq_b1 = ~(a[1] ^ b[1]);  // Check if bit 1 is equal
     assign a2_eq_b2 = ~(a[2] ^ b[2]);  // Check if bit 2 is equal
     assign a3_eq_b3 = ~(a[3] ^ b[3]);  // Check if bit 3 is equal
     // Final output: a is equal to b if all bits are equal
    assign equal = a0_eq_b0 & a1_eq_b1 & a2_eq_b2 & a3_eq_b3; // AND all comparisons
endmodule
```

Figure 36: Verilog Code for 4 bit Comparator



Name	Value	Data Type
a[3:0]	1	Array
b[3:0]	4	Array
[3]	0	Logic
[2]	1	Logic
[1]	0	Logic
[0]	0	Logic
equal	0	Logic
a0_eq_b0	0	Logic
a1_eq_b1	1	Logic
a2_eq_b2	0	Logic
a3_eq_b3	1	Logic

Name	Value
a[3:0]	1
b[3:0]	4
equal	0
a0_eq_b0	0
a1_eq_b1	1
a2_eq_b2	0
a3_eq_b3	1

Figure 37: Simulation

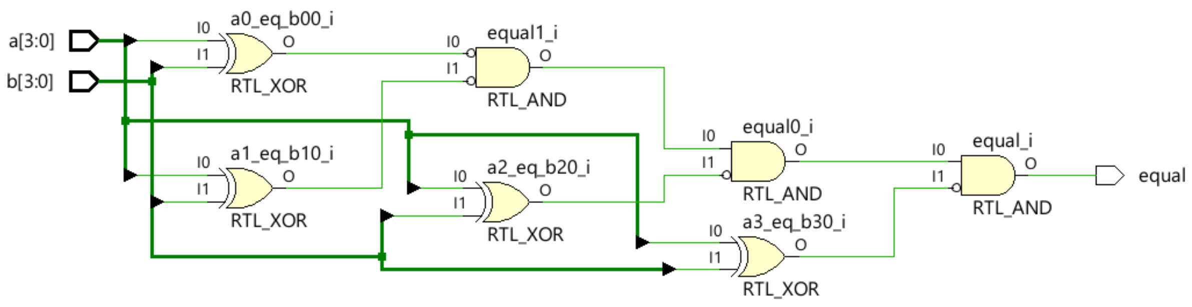


Figure 38: Schematic

Conclusion

The implemented codes and designs for basic logic gates, multiplexers, demultiplexers,half adders,full adders and comparators demonstrated the versatility and effectiveness of Verilog in digital system design. Each Combinational logic circuit highlighted the significance of various design approaches, enabling the successful execution of complex logic functions essential for digital circuits.