10/5/2025

DHARSHINI M

B.E ECE @SKCET

***Projects in Verilog: A Beginner’s Journey into Logic Circuits***

**TABLE OF CONTENTS:**

|  |  |  |
| --- | --- | --- |
| S.NO | TITLE | PAGE NO |
| 1. | INTRODUCTION | 2 |
| 2. | DATA TYPES | 3 |
| 3. | PROJECTS |  |
|  | 3.1 HALF ADDER | 4 |
|  | 3.2 FULL ADDER | 7 |
|  | 3.3 D- FLIP FLOP | 10 |
|  | 3.4 4-1 MUX | 12 |
|  | 3.5 8-1 MUX | 16 |
|  | 3.6 4-BIT RIPPLE CARRY ADDER | 20 |
|  | 3.7 SEVEN SEGMENT | 24 |
|  | 3.8 ALU | 27 |
|  | 3.9 TRAFFIC CONTROL SYSTEM | 31 |
| 4. | CONCLUSION | 36 |

**Introduction**

**Verilog** is a powerful and widely-used **hardware description language (HDL)** that enables the modeling, simulation, and design of digital systems such as microprocessors, memory units, and other complex circuits. It allows designers to describe both the structure and behavior of hardware at various abstraction levels, ranging from **gate-level to system-level design**. By offering modularity and flexibility, Verilog has become an essential tool in modern VLSI design and verification processes.

To simulate and verify Verilog designs effectively, tools like **Icarus Verilog and GTKWave** are invaluable. Icarus Verilog is an open-source simulator that provides a reliable environment for compiling and simulating Verilog code.

It helps users identify errors, debug designs, and test the functionality of circuits before physical implementation. Complementing this, GTKWave is a powerful waveform viewer that **visually represents signal behavior over time**. With GTKWave, designers can analyze the outputs of Verilog simulations, track signal transitions, and ensure circuit behavior aligns with specifications.

Together, Icarus Verilog and GTKWave form a complete ecosystem for learning and practicing digital design. These tools simplify the process of verifying functionality, debugging errors, and gaining insights into how digital systems behave under various conditions, making them indispensable for both beginners and advanced learners in the field of VLSI.

**Data Types**

1. **Net Data Types**

Nets represent connections between hardware elements.

* **wire**: The most common net type, used to represent physical connections between gates or modules in your design. Example use cases include connecting outputs of combinational logic to other components.

*wire a, b, c;*

* **tri**: Used for tri-state logic, typically seen in shared data buses.

*tri a;*

* **wand**: Wired AND, used when multiple drivers are connected to the same net, with an AND operation applied to their values.

*wand a;*

* **wor**: Wired OR, similar to wand but applies an OR operation instead.

*wor a;*

1. **Register Data Types**

Registers store values between clock cycles or for sequential logic.

* **reg**: Used for procedural assignments inside always or initial blocks. This is most commonly used for storing state in flip-flops or sequential circuits.

*reg a;* // Single-bit register

*reg [3:0] b;* // 4-bit register

* **integer**: General-purpose signed integer, often used for counting or loop iterations.

*integer i;*

* **real**: Floating-point number, used for simulation and testbench purposes (not synthesizable).

*real x;*

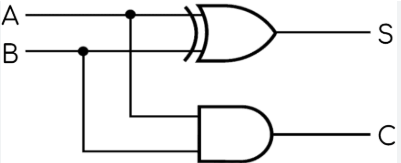
* **time**: Stores simulation time values, useful for timestamping events in simulation.

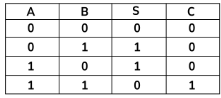
*time t;*

1. **PROJECTS:**
   1. ***Half-adder:***
   2. **Introduction:**

A Half Adder is a combinational circuit used to perform the addition of two single-bit binary numbers. It provides two outputs: **Sum** (the result of XOR operation) and **Carry** (the result of AND operation). The Half Adder is a fundamental building block for constructing more complex arithmetic circuits like Full Adders and Binary Adders.

* 1. **Circuit diagram:**





* 1. **Code:**

*--Verilog*

module half\_adder(

input wire A,B,

output wire Sum, Carry);

assign Sum=A^B;

assign Carry=A&B;

endmodule

* 1. **TestBench code:**

*--verilog*

module half\_adder\_tb;

reg A;

reg B;

wire Sum;

wire Carry;

half\_adder uut(

.A(A),

.B(B),

.Sum(Sum),

.Carry(Carry)

);

initial begin

$dumpfile("half\_adder\_tb.vcd");

$dumpvarse(0,half\_adder\_tb);

$display("A B | Sum Carry");

$monitor("%b %b | %b %b",A,B,Sum,Carry);

A=0;B=0;#10;

A=0;B=1;#10;

A=1;B=0;#10;

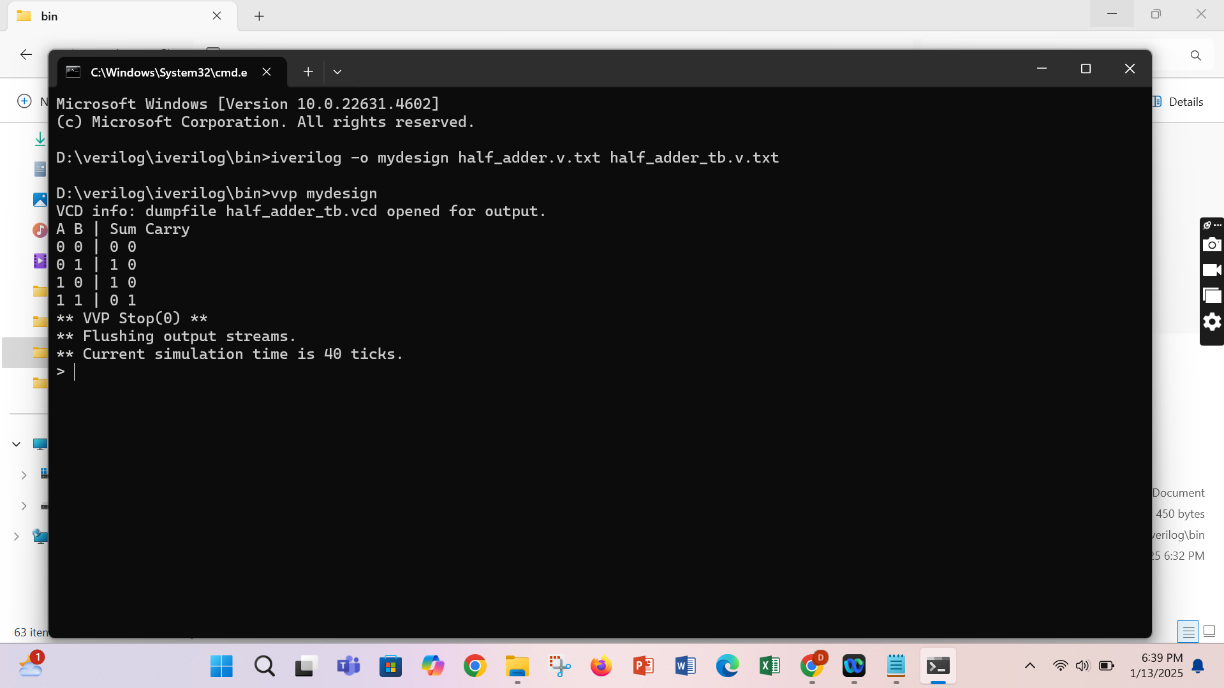
A=1;B=1;#10;

$stop;

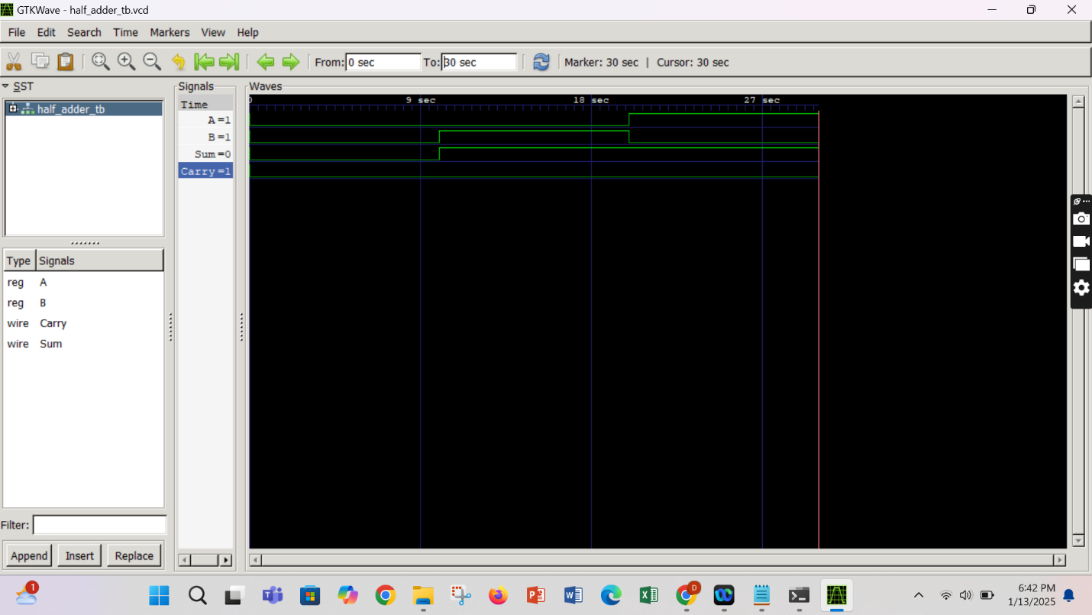
end

endmodule

* 1. **Command Prompt:**

****

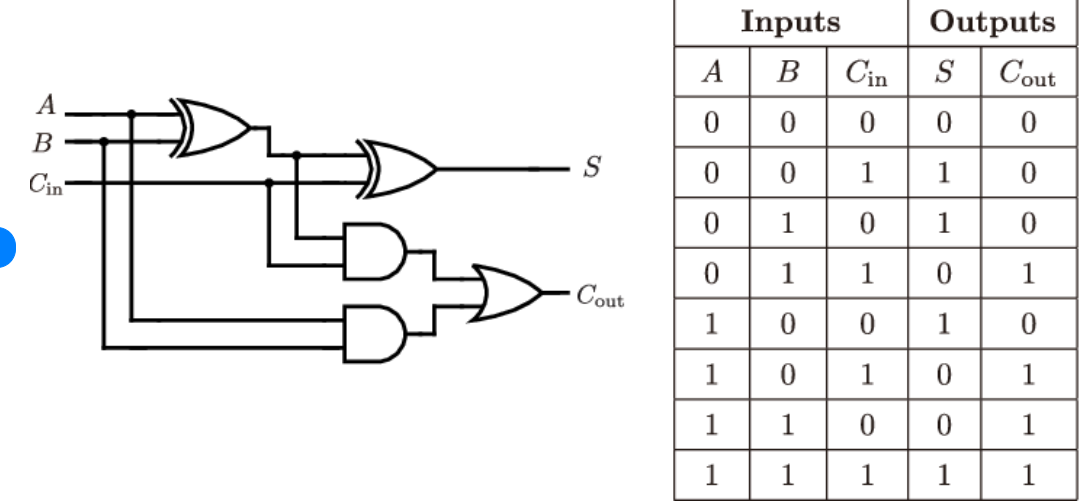
* 1. **GTKWave output:**

****

* 1. ***Full-Adder***
  2. **Introduction:**

A Full Adder is a combinational circuit used to perform the addition of three single-bit binary numbers: two input bits (A and B) and a carry-in bit (Cin). It produces two outputs: **Sum** (result of the addition) and **Carry-out (Cout)**, which is passed to the next significant bit in multi-bit addition.

* 1. **Circuit diagram**



* 1. **Code:**

module full\_adder(

input wire A,B,Cin,

output wire Sum,Carry);

Sum=A ^ B ^ Cin;

Carry=(A & B) | (B & Cin) | (A & Cin);

Endmodule

* 1. **Test Bench code:**

module full\_adder\_tb;

reg A,B,Cin;

wire Sum,Carry;

full\_adder uut(

.A(A),

.B(B),

.Cin(Cin),

.Sum(Sum),

.Carry(Carry)

);

initial begin

$dumpfile("full\_adder\_tb.vcd");

$dumpvars(0,full\_adder\_tb);

$display("A B Cin | Sum Carry");

$monitor("%b %b %b | %b %b", A , B, Cin ,Sum,Carry);

A=0;B=0;Cin=0;#10;

A=0;B=0;Cin=1;#10;

A=0;B=1;Cin=0;#10;

A=0;B=1;Cin=1;#10;

A=1;B=0;Cin=0;#10;

A=1;B=0;Cin=1;#10;

A=1;B=1;Cin=0;#10;

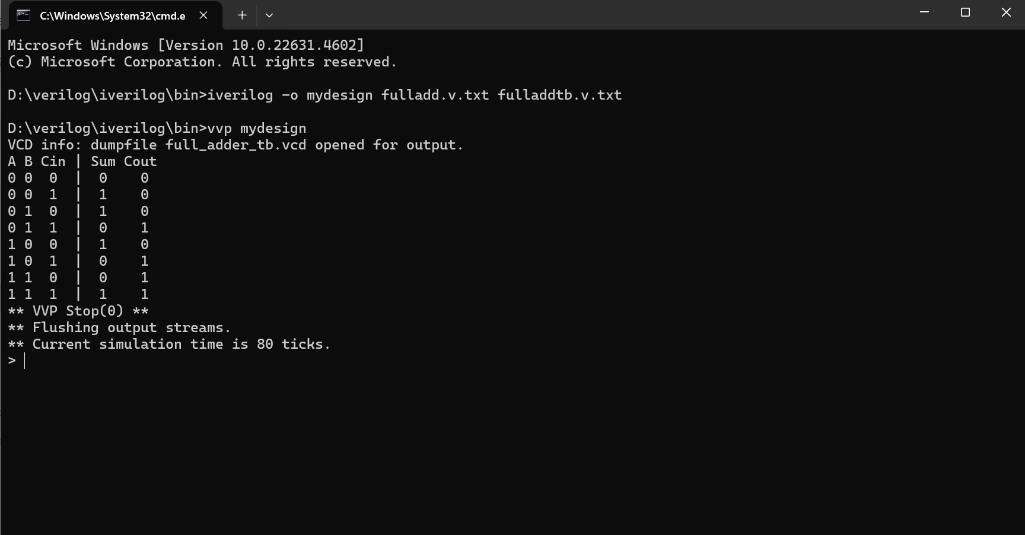
A=1;B=1;Cin=1;#10;

$stop;

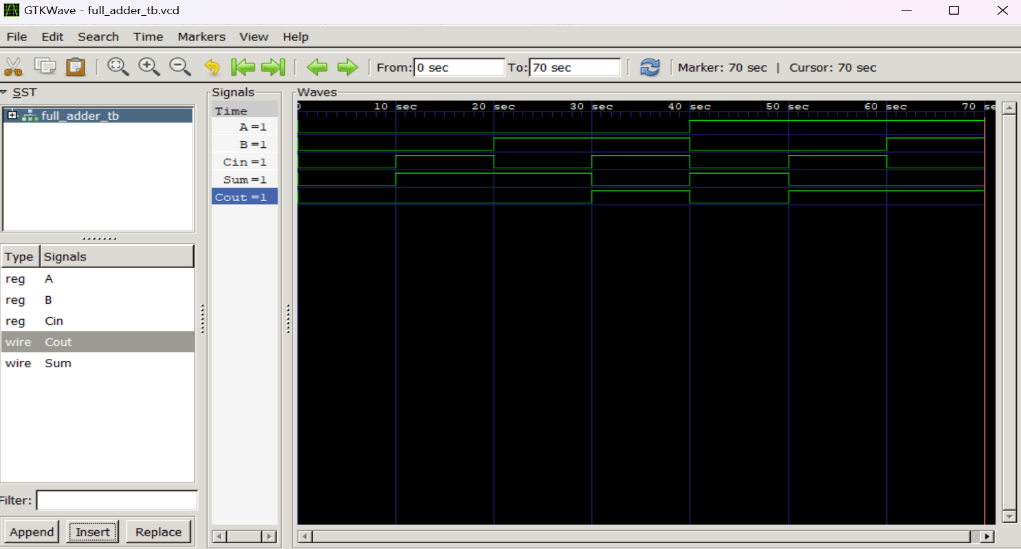
end

endmodule

* 1. **Command Prompt:**

****

* 1. **GTKWave Simulation:**

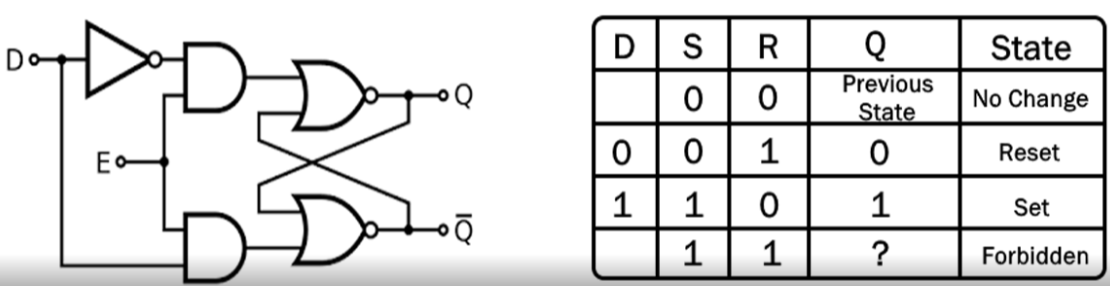
****

* 1. ***D-Flip Flop:***

1. **Introduction:**

A **D Flip-Flop (Data or Delay Flip-Flop)** is a sequential circuit that captures the input (D) at the rising or falling edge of the clock signal and holds it until the next clock edge. It ensures data synchronization and is widely used in registers, memory units, and digital circuits where data storage and timing control are crucial. The D Flip-Flop prevents metastability issues by eliminating the indeterminate state found in SR flip-flops. It has two main outputs: **Q (stored data)** and **Q' (complement of Q)**.

1. **Circuit Diagram:**



1. **Code:**

*--Verilog*

module d\_flipflop(

input wire D,

input wire clk,

input wire reset,

output reg Q

);

always @(posedge clk or posedge reset) begin

if(reset)

Q<=1'b0;

else

Q<=D;

end

endmodule

1. **TestBench:**

module tb\_d\_flipflop;

reg D;

reg clk;

reg reset;

wire Q;

d\_flipflop uut(

.D(D),

.clk(clk),

.reset(reset),

.Q(Q)

);

initial begin

clk=0;

forever #5 clk=~clk;

end

initial begin

$dumpfile("d\_flipflop.vcd");

$dumpvars(0,tb\_d\_flipflop);

D=0;

reset=1;

#10;

#10;

D=1; #10;

D=0; #10;

D=1; #10;

reset=1;#10;

reset=0;#10;

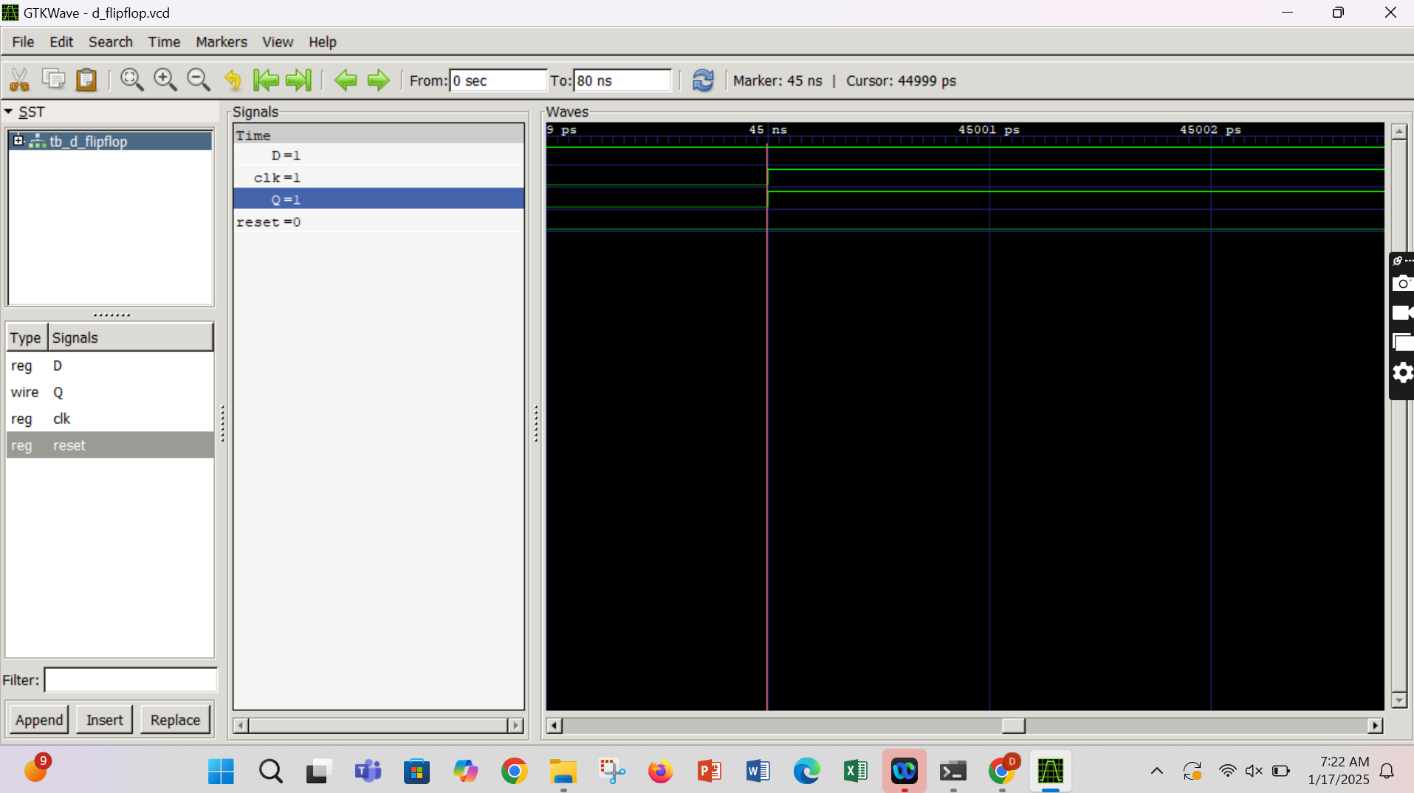
D=1;#10;

$finish;

end

endmodule

1. **GTKWave Simulation:**

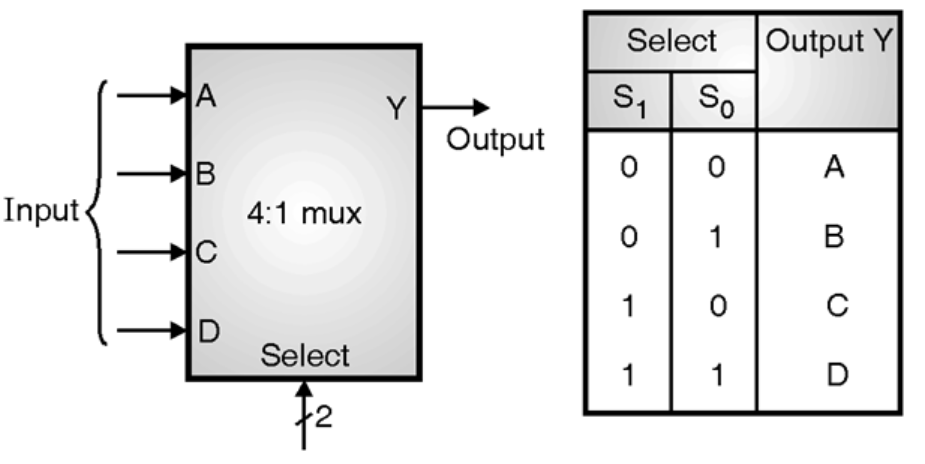


* 1. ***4-1 Mux:***

1. **Introduction:**

A **4-to-1 Multiplexer (MUX)** is a combinational circuit that selects one of four input signals and forwards it to a single output based on the values of two selection lines (**S1** and **S0**). It acts as a data selector, allowing multiple data inputs to share a single transmission channel efficiently.Multiplexers are widely used in digital circuits, including data routing, arithmetic logic units (ALUs), and communication systems.

1. **Circuit Diagram:**

****

1. **Code:**

module mux4to1(

input wire [0:3] i,

input wire [0:1] sel,

output reg y

);

always @(\*) begin

case(sel)

2'b00: y=i[0];

2'b01: y=i[1];

2'b10: y=i[2];

2'b11: y=i[3];

default: y=1'b0;

endcase

end

endmodule

1. **TestBench Code:**

module tb\_mux4to1;

reg [3:0] i;

reg [1:0] sel;

wire y;

mux4to1 uut(

.i(i),

.sel(sel),

.y(y)

);

initial begin

$dumpfile("mux4to1.vcd");

$dumpvars(0,tb\_mux4to1);

$monitor("Time=%0t | i=%b | sel=%b |y=%b", $time, i, sel, y);

i=4'b1010;sel=2'b00; #10;

i=4'b1010;sel=2'b01; #10;

i=4'b1010;sel=2'b10; #10;

i=4'b1010;sel=2'b11; #10;

i=4'b1100;sel=2'b00; #10;

i=4'b1100;sel=2'b01; #10;

i=4'b1100;sel=2'b10; #10;

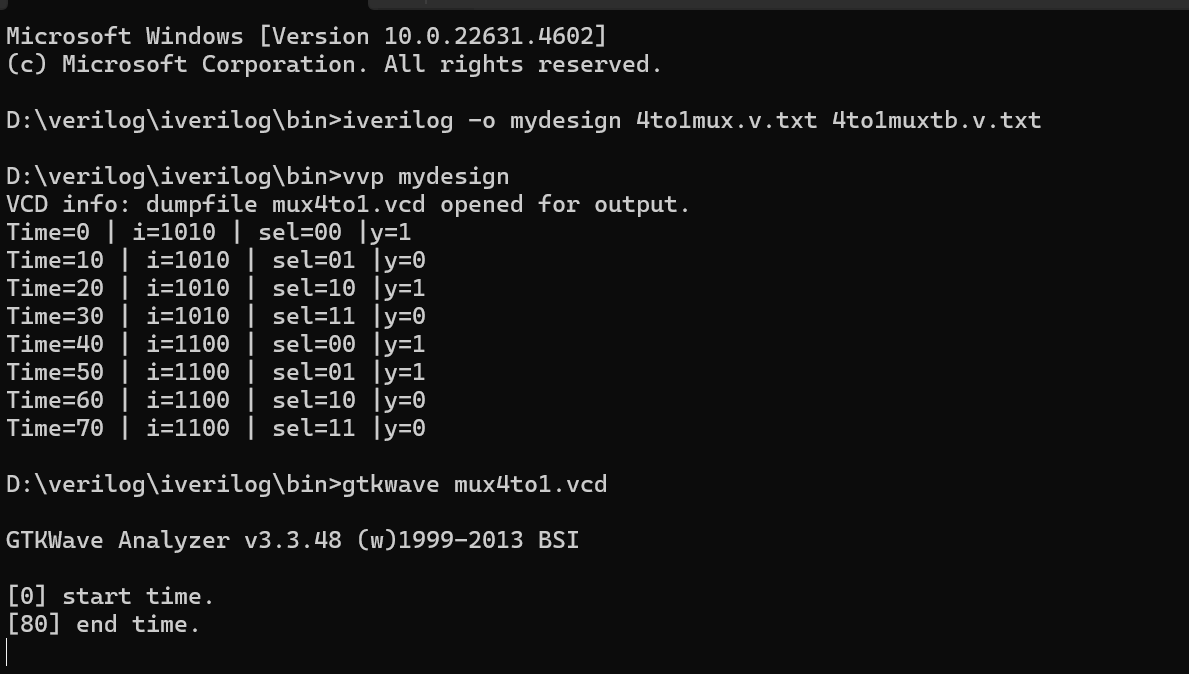
i=4'b1100;sel=2'b11; #10;

$finish;

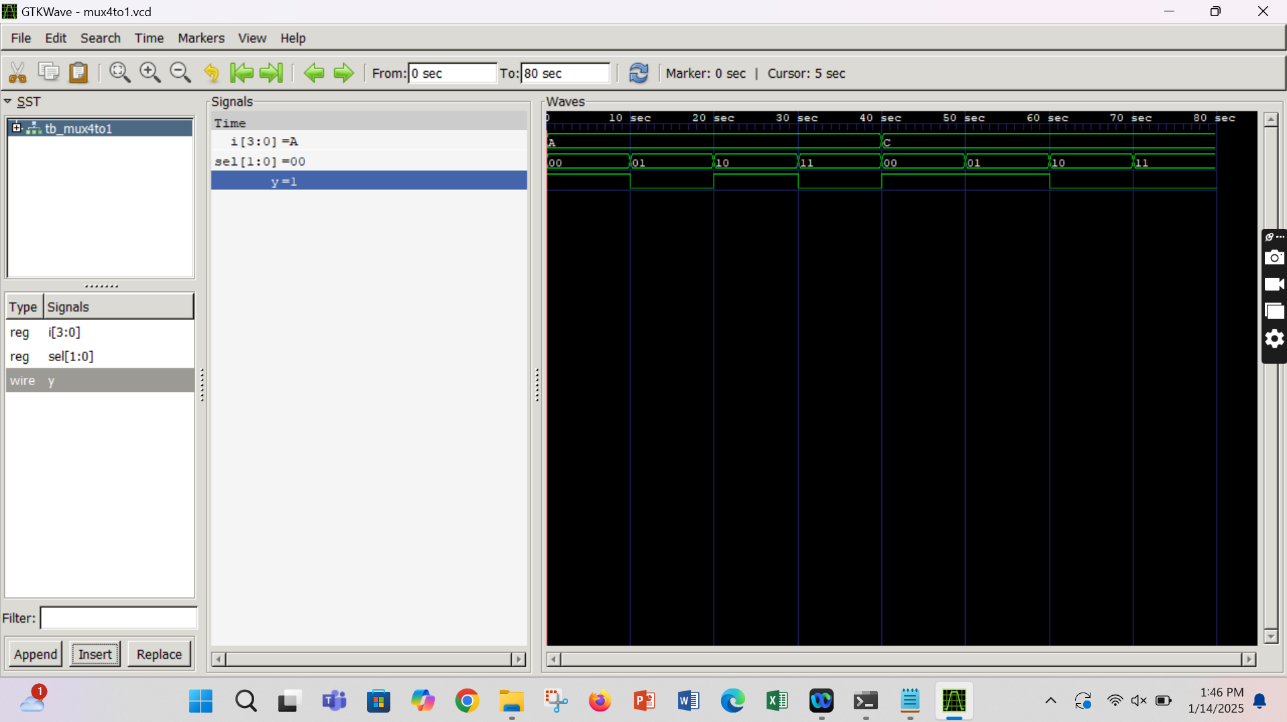
end

endmodule

1. **Command Prompt:**



1. **GTKWave Simulation:**

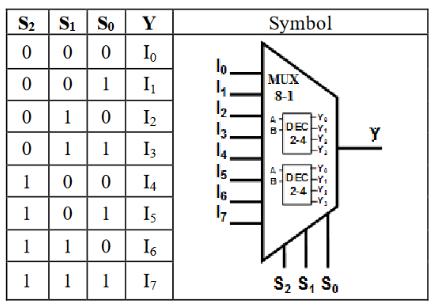


* 1. ***8-1 MUX:***

1. **Introduction:**

An **8-to-1 Multiplexer (MUX)** is a combinational circuit that selects one out of eight input signals and forwards it to a single output based on the values of three selection lines (**S2, S1, and S0**). It is an essential component in digital systems, often used in data routing, communication networks, and ALU (Arithmetic Logic Unit) operations.

1. **Circuit Diagram:**

****

1. **Code:**

module mux8to1(

input [7:0] in;

input [2:0] sel;

output reg out

);

always @(\*) begin

case(sel)

3'b000: out=in[0];

3'b001: out=in[1];

3'b010: out=in[2];

3'b011: out=in[3];

3'b100: out=in[4];

3'b101: out=in[5];

3'b110: out=in[6];

3'b111: out=in[7];

default: out= 1'b0;

endcase

end

endmodule

1. **TestBench Code:**

module tb\_mux8to1;

reg [7:0] in;

reg [0:2] sel;

wire out;

mux8to1 uut(

.in(in),

.sel(sel),

.out(out)

);

initial begin

in=8'b10101010;

sel=3'b000;

$dumpfile("mux8to1.vcd");

$dumpvars(0,tb\_mux8to1);

$display("Time\t\tin\t\tsel\tout");

$monitor("%g\t%b\t%b\t%b",$time,in,sel,out);

#10 sel=3'b000;

#10 sel=3'b001;

#10 sel=3'b011;

#10 sel=3'b100;

#10 sel=3'b101;

#10 sel=3'b110;

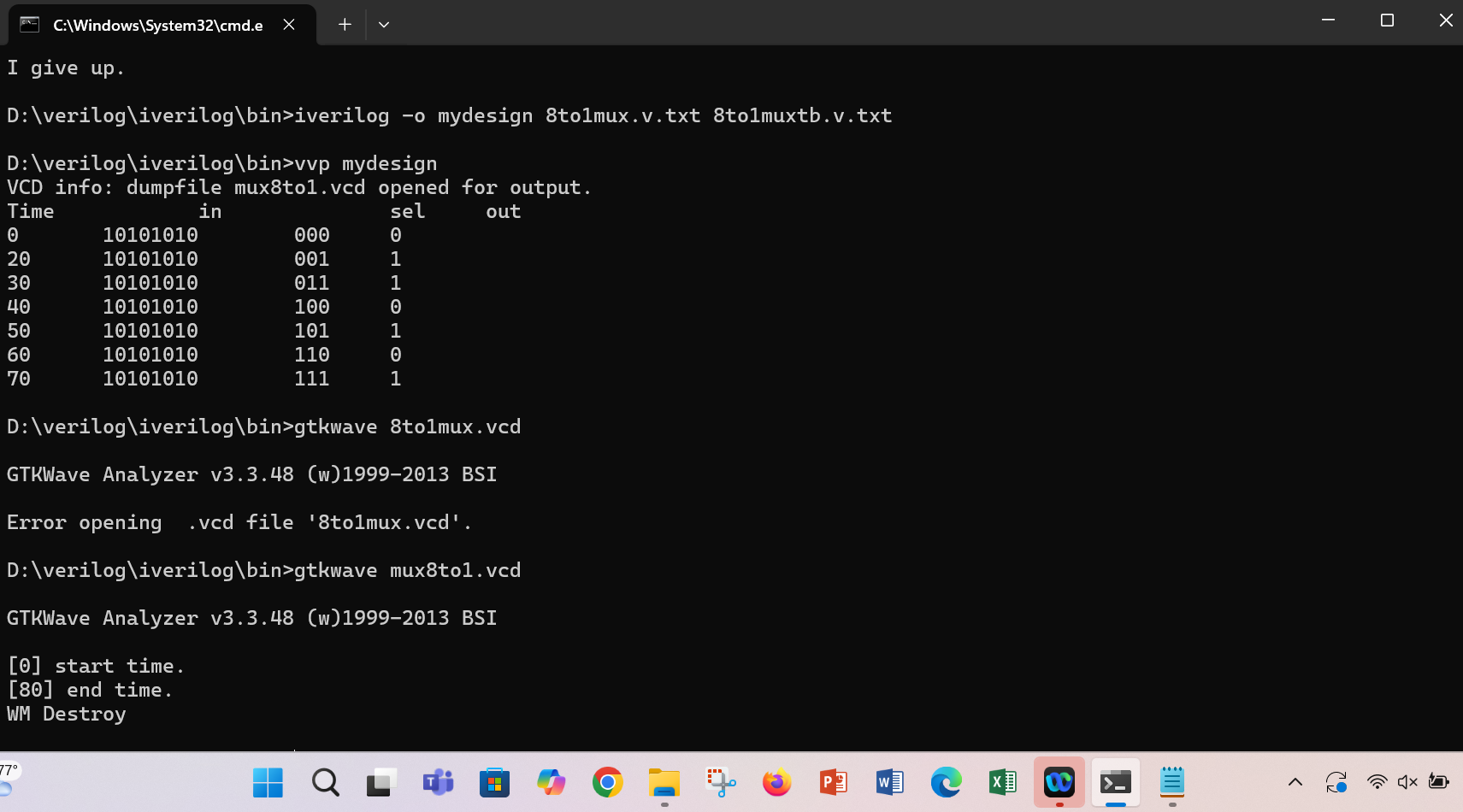
#10 sel=3'b111;

#10 $finish;

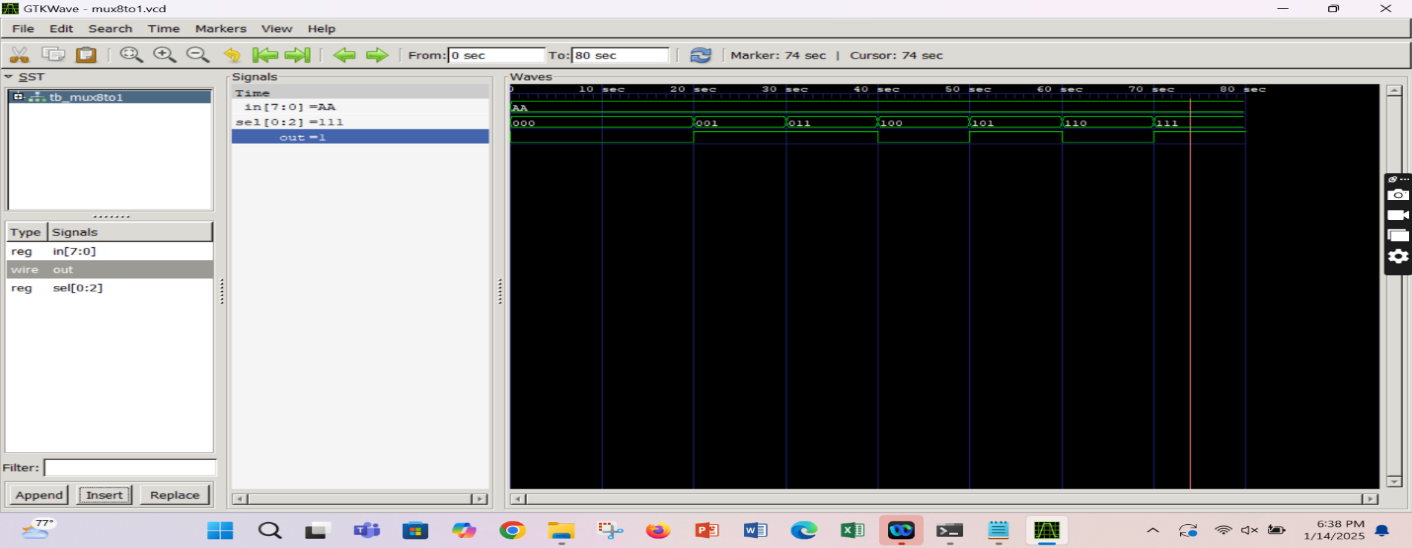
end

endmodule

1. **Command Prompt:**

****

1. **GTKWave Output:**

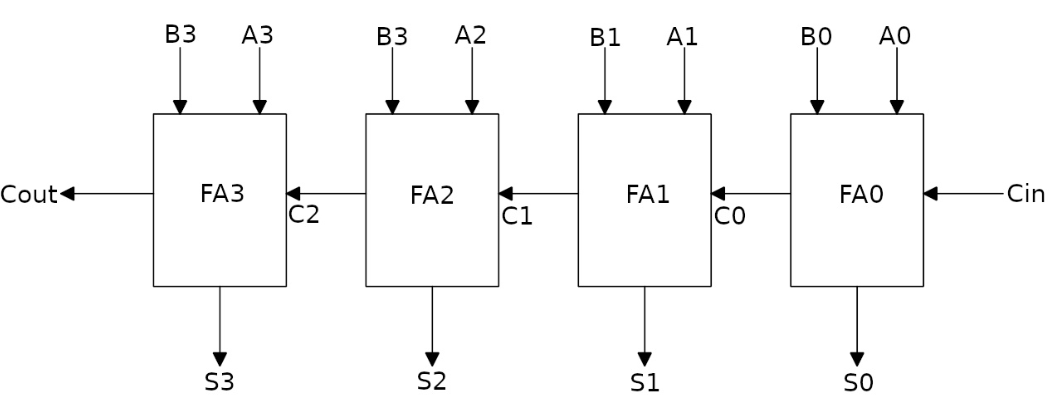
****

* 1. ***4-Bit ripple carry Adder***

1. **Introduction**

A **4-bit Ripple Carry Adder (RCA)** is a combinational circuit that performs binary addition on two 4-bit numbers. It consists of four **full adders** connected in series, where the carry output of each adder is passed to the next stage as a carry-in. This design is called "ripple carry" because the carry signal propagates sequentially through all the full adders, which may cause a delay in large bit-width adders.

1. **Circuit diagram**

****

1. **Code**

module ripple\_carry\_adder(

input [3:0] A,

input [3:0] B,

input Cin,

output [3:0] S,

output Cout

);

wire c1,c2,c3;

full\_adder FA0 (.A(A[0]),.B(B[0]),.Cin(Cin),.S(S[0]),.Cout(c1));

full\_adder FA0 (.A(A[1]),.B(B[1]),.Cin(c1),.S(S[1]),.Cout(c2));

full\_adder FA0 (.A(A[2]),.B(B[2]),.Cin(c2),.S(S[2]),.Cout(c3));

full\_adder FA0 (.A(A[3]),.B(B[3]),.Cin(c3),.S(S[3]),.Cout(Cout));

endmodule

module full\_adder(

input A,B,Cin,

output S,Cout

);

assign Sum=A^B^Cin;

assign Cout= (A & B) | (B & Cin) | (A & Cin);

endmodule

1. **Test Bench code**

module ripple\_carry\_adder\_tb;

reg [3:0] A;

reg [3:0] B;

reg Cin;

wire [3:0] Sum;

wire Cout;

ripple\_carry\_addr uut(

.A(A),

.B(B),

.Cin(Cin),

.Sum(Sum),

.Cout(Cout)

);

initial begin

$dumpfile("ripple\_carry\_adder\_tb.vcd");

$dumpvars(0,ripple\_carry\_adder\_tb);

$display("A B Cin | Sum Cout");

$monitor("%b %b %b | %b %b",A,B,Cin,Sum,Cout);

A=4'b0000;B=4'b0000;Cin=0;#10;

A=4'b0001;B=4'b0001;Cin=0;#10;

A=4'b0110;B=4'b0101;Cin=0;#10;

A=4'b0111;B=4'b0111;Cin=0;#10;

A=4'b1111;B=4'b1111;Cin=0;#10;

A=4'b1010;B=4'b0101;Cin=1;#10;

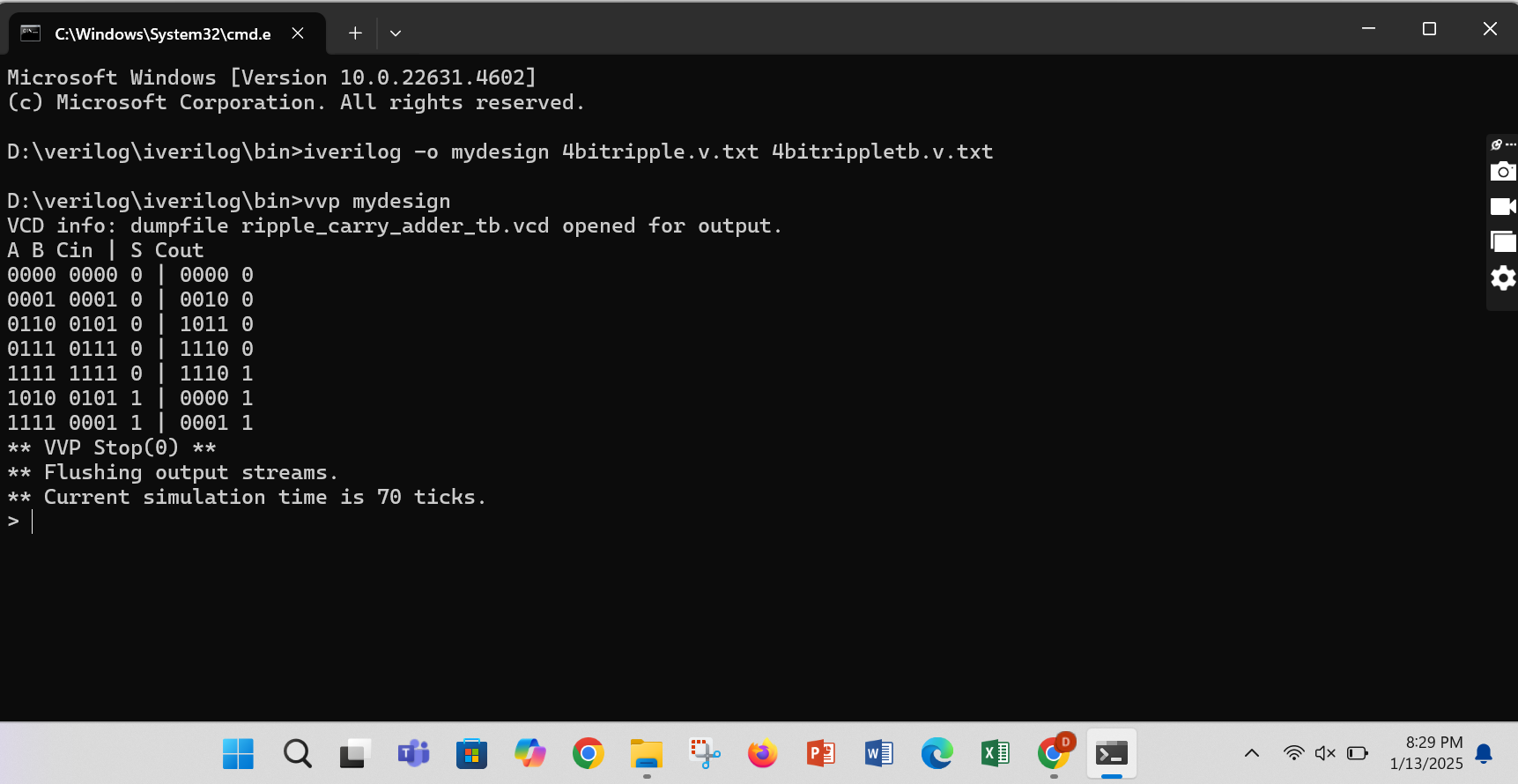
A=4'b1111;B=4'b0001;Cin=1;#10;

$stop;

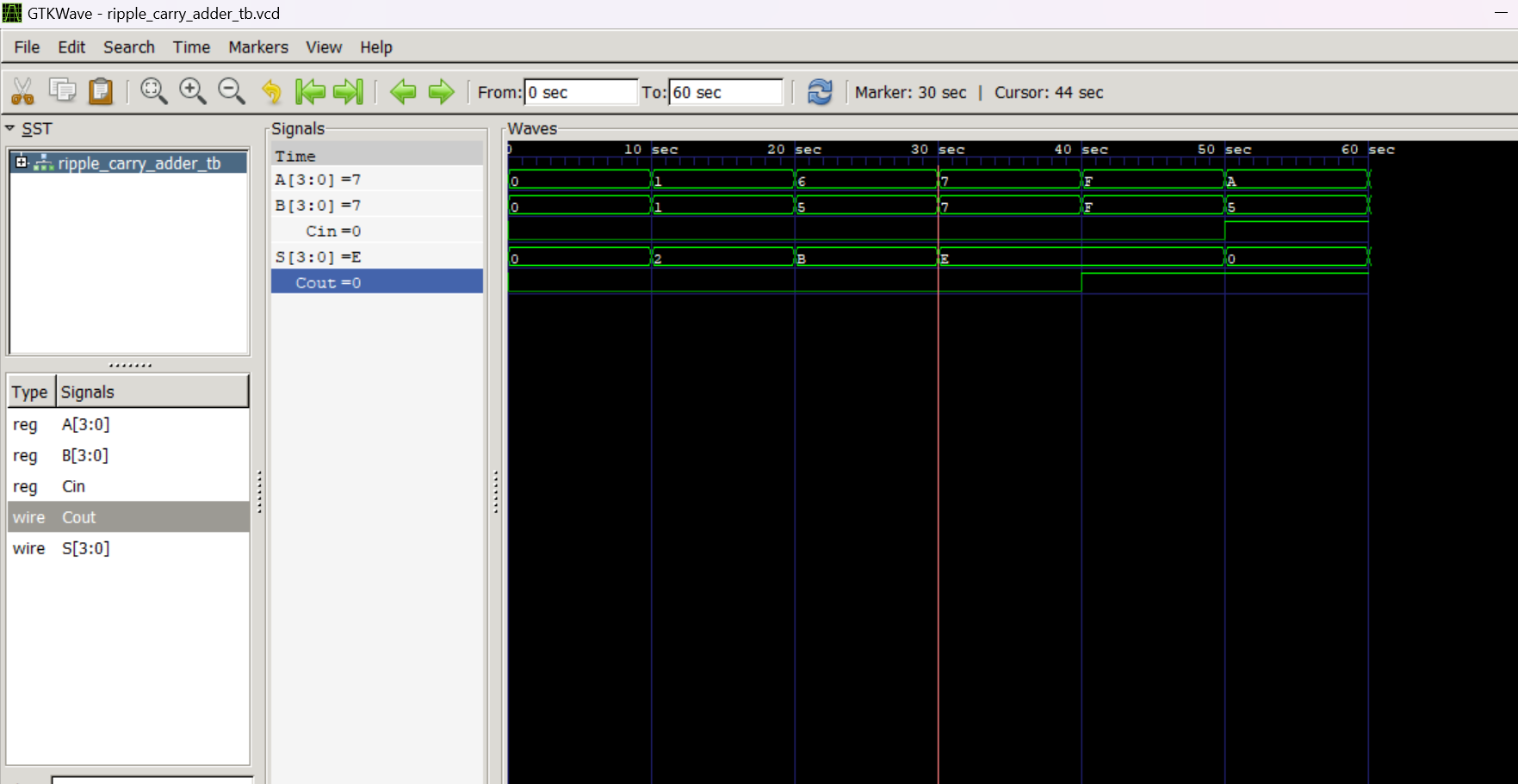
end

endmodule

1. **Command prompt**

****

1. **GTK Wave simulation**



* 1. ***Seven Segment:***

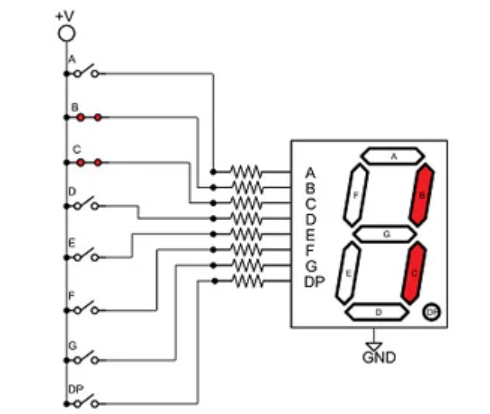
**a.Introduction:**

This Verilog module is designed to drive a **seven segment display** using a 4-bit binary input. The input (binary\_in) represents decimal values from **0 to 9**, and the module outputs a corresponding 7-bit signal (seg\_out) that controls the individual segments (**a to g**) of the display.

The logic uses a case statement within an always block to match each binary input with its appropriate segment combination. Each 7-bit output pattern is tailored for a **common cathode seven segment display**, where a logic **0** turns ON the segment and **1** turns it OFF.

This driver is commonly used in digital clocks, counters, and embedded systems where numeric values need to be displayed clearly using LEDs. The code also includes a default case to handle invalid inputs by turning off all segments, ensuring robust behavior.

**b. Circuit Diagram:**

****

**c. Code**

module seven\_segemnt\_driver(

input wire [3:0] binary\_in,

output reg [6:0] seg\_out

);

always @(\*) begin

case(binary\_in)

4'b0000:seg\_out = 7'b0000001;

4'b0001:seg\_out = 7'b1001111;

4'b0010:seg\_out = 7'b0010010;

4'b0011:seg\_out = 7'b0000110;

4'b0100:seg\_out = 7'b1001100;

4'b0101:seg\_out = 7'b0100100;

4'b0110:seg\_out = 7'b0100000;

4'b0111:seg\_out = 7'b0001111;

4'b1000:seg\_out = 7'b0000000;

4'b1001:seg\_out = 7'b0000100;

default:seg\_out = 7'b1111111;

endcase

end

endmodule

**d. TestBench code:**

`timescale 1ns/1ps

module tb\_seven\_segment\_driver;

reg [3:0] binary\_in;

wire [6:0] seg\_out;

seven\_segment\_driver uut(

.binary\_in(binary\_in),

.seg\_out(seg\_out)

);

initial begin

$dumpfile("seven\_segemnt\_driver.vcd");

$dumpvars(0,tb\_seven\_segment\_driver);

binary\_in=4'b0000;#10;

binary\_in=4'b0001;#10;

binary\_in=4'b0010;#10;

binary\_in=4'b0011;#10;

binary\_in=4'b0100;#10;

binary\_in=4'b0101;#10;

binary\_in=4'b0110;#10;

binary\_in=4'b0111;#10;

binary\_in=4'b1000;#10;

binary\_in=4'b1001;#10;

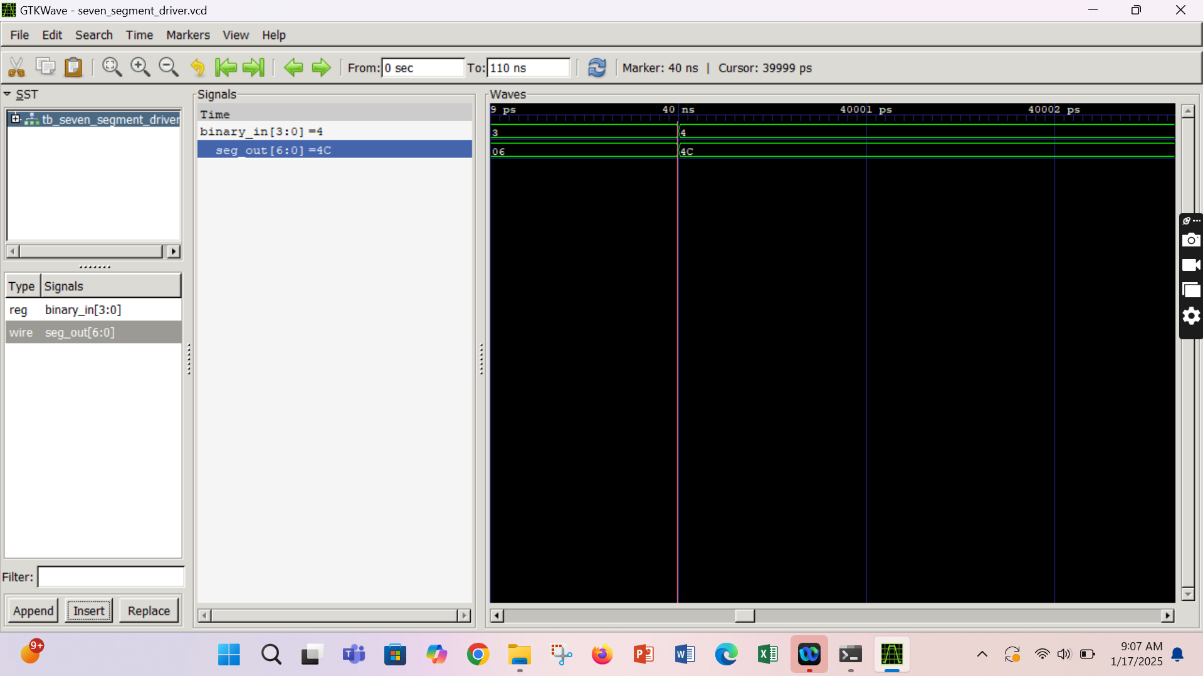
binary\_in=4'b1111;#10;

$finish;

end

endmodule

**e.GTKWave output:**

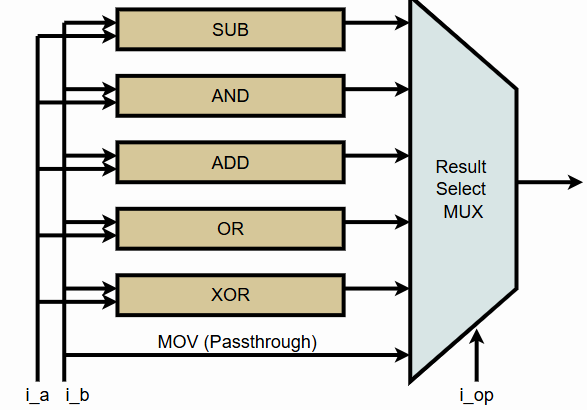


* 1. ***ALU:***

1. **Introduction:**

An Arithmetic Logic Unit (ALU) is a fundamental component of digital systems, particularly within the CPU, responsible for executing arithmetic and logical operations. This project presents a simple ALU circuit capable of performing five essential operations: addition, subtraction, bitwise AND, OR, and XOR. The ALU takes two binary inputs and uses combinational logic—built from full adders, logic gates, and multiplexers—to carry out the selected function based on control signals or opcodes. By integrating these operations into a single circuit, the ALU efficiently demonstrates how basic computational tasks are executed at the hardware level, offering a clear understanding of the processing core in modern computing devices.

1. **Circuit Diagram:**

****

1. **Code:**

module ALU(

input [3:0] a,

input [3:0] b,

input [2:0] sel,

output reg [3:0] result,

output reg zero\_flag

);

always @(\*) begin

case(sel)

3'b000: result=a+b;

3'b001: result=a-b;

3'b010: result=a&b;

3'b011: result=a|b;

3'b100: result=a^b;

default: result=4'b0000;

endcase

zero\_flag=(result==4'b0000);

end

endmodule

1. **TestBench code:**

module ALU\_tb;

reg [3:0] a;

reg [3:0] b;

reg [2:0] sel;

wire [3:0] result;

wire zero\_flag;

ALU uut(

.a(a),

.b(b),

.sel(sel),

.result(result),

.zero\_flag(zero\_flag)

);

initial begin

a=4'b0011; b=4'b0001; sel=3'b000; #10;

$display("ADD: a=%b,b=%b,sel=%b ->result=%b,zero\_flag=%b",a,b,sel,result,zero\_flag);

a=4'b0100; b=4'b0011; sel=3'b001; #10;

$display("SUB: a=%b,b=%b,sel=%b ->result=%b,zero\_flag=%b",a,b,sel,result,zero\_flag);

a=4'b1100; b=4'b1010; sel=3'b010; #10;

$display("AND: a=%b,b=%b,sel=%b ->result=%b,zero\_flag=%b",a,b,sel,result,zero\_flag);

a=4'b1100; b=4'b1010; sel=3'b011; #10;

$display("OR: a=%b,b=%b,sel=%b ->result=%b,zero\_flag=%b",a,b,sel,result,zero\_flag);

a=4'b1100; b=4'b1010; sel=3'b100; #10;

$display("XOR: a=%b,b=%b,sel=%b ->result=%b,zero\_flag=%b",a,b,sel,result,zero\_flag);

a=4'1111; b=4'b0000; sel=3'b111; #10;

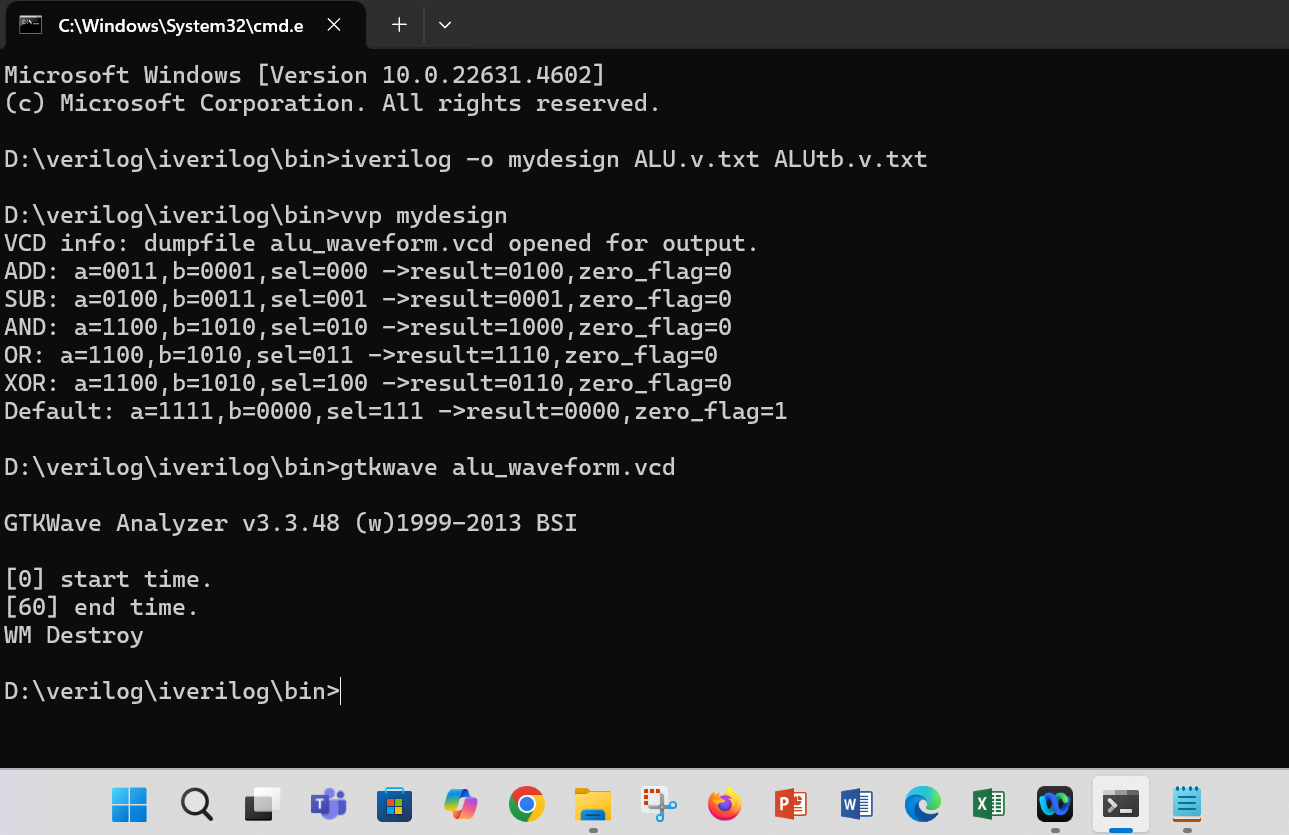
$display("Default: a=%b,b=%b,sel=%b ->result=%b,zero\_flag=%b",a,b,sel,result,zero\_flag);

$finish;

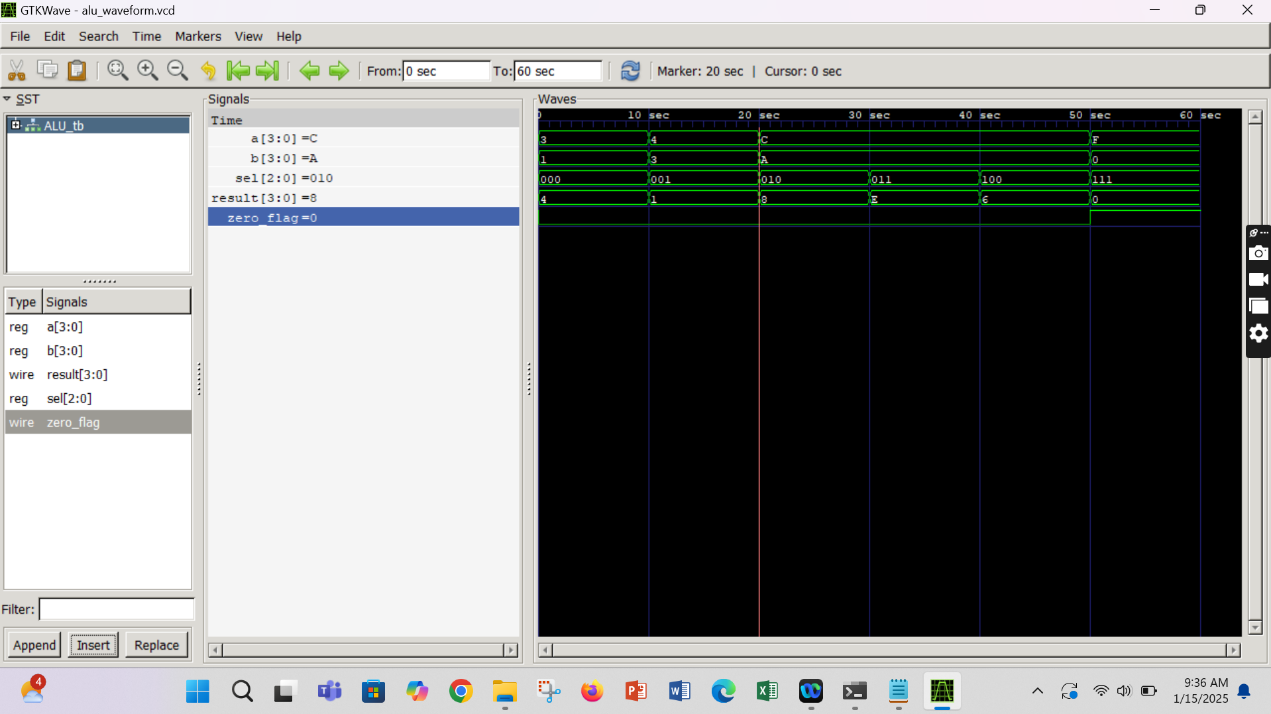
end

endmodule

1. **Command Prompt:**

****

1. **GTKWave Output:**

****

* 1. ***Traffic Control system***

1. ***Introduction:***

The **Traffic Light Controller** is a crucial component in urban traffic management systems, designed to regulate the flow of vehicles and ensure safety at intersections. This project implements a digital traffic light controller using Verilog HDL, simulating the real-world operation of red, green, and yellow signals. The system operates on a finite state machine (FSM) model, cycling through traffic light states based on a timer. It uses three defined states—**red**, **green**, and **yellow**—each assigned specific time durations (10, 7, and 3 clock cycles, respectively). The controller responds to a clock signal and reset input, and outputs a 3-bit signal indicating the active light. This design is efficient, hardware-friendly, and demonstrates fundamental FSM concepts used in real-time embedded control applications.

1. ***Code***

module traffic\_light\_controller(

input wire clk,

input wire reset,

output reg [2:0] light

);

typedef enum reg[1:0]{

red=2'b00,

green=2'b01,

yellow=2'b10

}state\_t;

state\_t current\_state,next\_state;

reg [3:0] timer;

localparam red\_time=4'd10;

localparam green\_time=4'd7;

localparam yellow\_time=4'd3;

always @(posedge clk or posedge reset) begin

if (reset) begin

current\_state<=red;

timer<=0;

end else begin

if(timer==0) begin

current\_state<=next\_state;

case(next\_state)

red:timer<=red\_time;

green:timer<=green\_time;

yellow:timer<=yellow\_time;

endcase

end else begin

timer<=timer-1;

end

end

end

always @(\*) begin

case(current\_state)

red:next\_state=green;

green:next\_state=yellow;

yellow:next\_state=red;

default:next\_state=red;

endcase

end

always @(\*) begin

case(current\_state)

red:light=3'b100;

green:light=3'b001;

yellow:light=3'b010;

default:light=3'b100;

endcase

end

endmodule

1. ***TestBench code***

module traffictb;

reg clk;

reg reset;

wire [2:0] light;

// Instantiate the traffic light controller

traffic\_light\_controller uut (

.clk(clk),

.reset(reset),

.light(light)

);

// Clock generation

initial begin

clk = 0;

forever #5 clk = ~clk; // 10ns clock period

end

// Test sequence

initial begin

$dumpfile("traffic.vcd"); // Dump waveform

$dumpvars(0, traffictb);

reset = 1; #10; // Assert reset

reset = 0; #100; // Deassert reset, observe the behavior

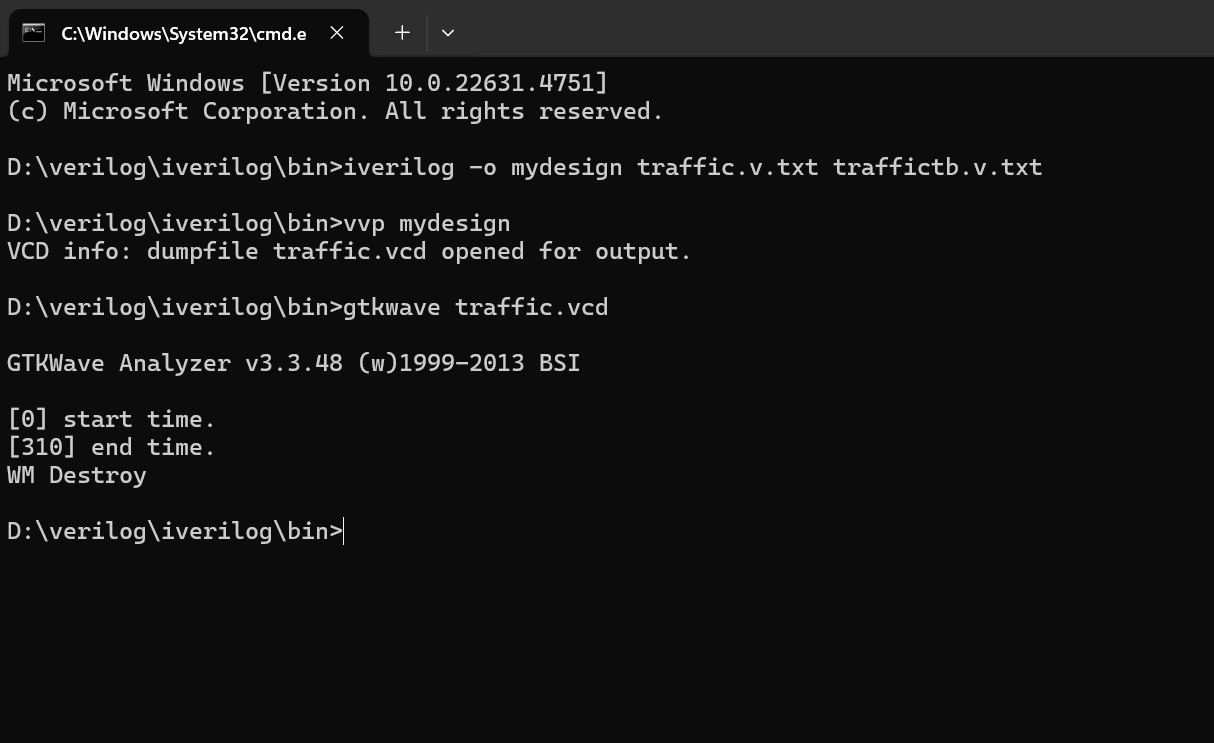
#200; // Let the simulation run for a while

$finish; // End simulation

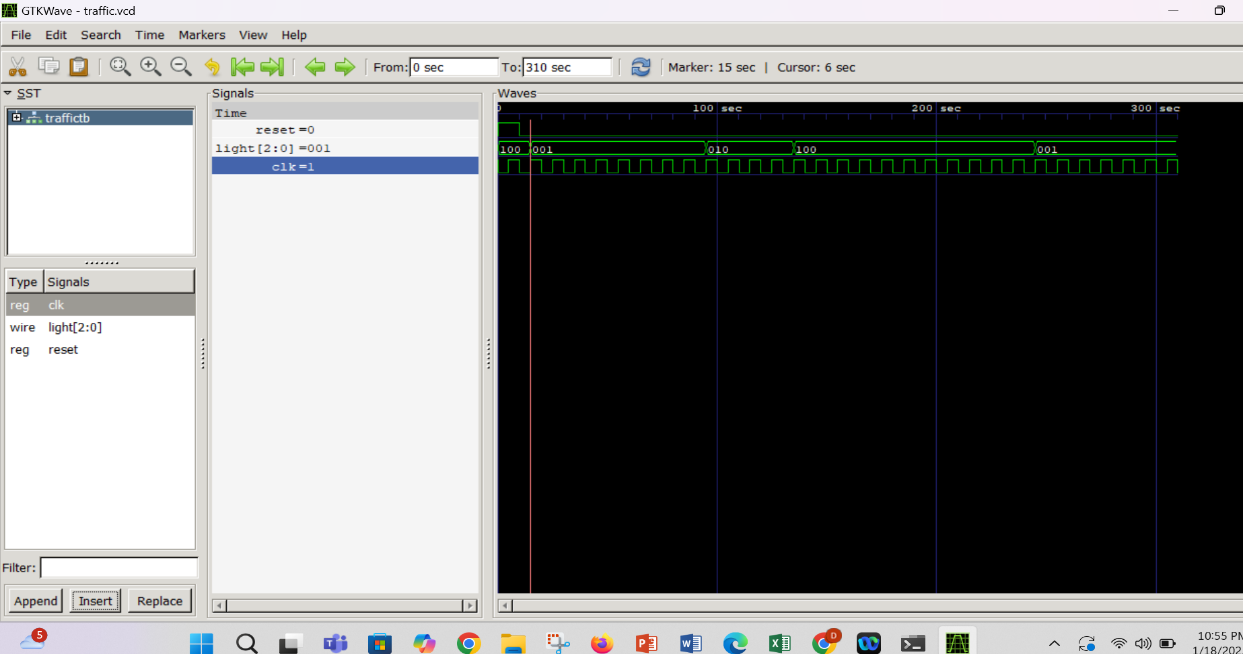
end

endmodule

1. **Command Prompt Output:**



1. **GTKWave Output:**

****

1. **CONCLUSION:**

This compilation of foundational digital design projects marks the beginning of my journey into the world of **Verilog HDL** and **analog circuit design**. Through hands-on implementation of basic modules like the **Half Adder**, **Full Adder**, **D Flip-Flop**, **Multiplexers**, and more complex systems such as the **4-bit Ripple Carry Adder**, **Seven Segment Display**, **ALU**, and **Traffic Control System**, I have gained valuable insights into the structure and behavior of digital logic systems. Each module has strengthened my understanding of data flow, control logic, and hardware-level operation. This journey has not only enhanced my technical skills but also sparked a deeper interest in the hardware-software interface. As I continue to explore advanced digital systems and embedded design, this foundational work stands as a critical first step toward mastering the world of digital electronics and system design.