A logo for a university

Description automatically generated

**END SEMESTER ASSESSMENT (ESA) B.TECH. (CSE)**

**III SEMESTER**

**UE23CS251A – DIGITAL DESIGN & COMPUTER ORGANIZATION LABORATORY**

**PROJECT REPORT**

**ON**

“DESIGN AND IMPLEMENTATION OF A 16 BIT ALU USING 4X1 MUX THAT PERFORMS SHIFT AND ROTATE OPERATIONS”

SUBMITTED BY

**NAME SRN**

**1)M NIRANJAN PES2UG23CS308**

**2)KEERTHAN P.V PES2UG23CS272**

**3)MAHESH I.A PES2UG23CS316**

**4)MALLIKARJUN K PES2UG23CS321**

**TEAM NO. 9**

**AUGUST – DECEMBER 2024**

**DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING**

**ELECTRONIC CITY CAMPUS,**

**BENGALURU – 560100, KARNATAKA, INDIA**

|  |  |  |
| --- | --- | --- |
| TABLE OF CONTENTS | | |
| Sl.No | TOPIC | PAGE No |
|  | ABSTRACT OF THE PROJECT | 3-6 |
|  | CIRCUIT DIAGRAM | 7 |
|  | MAIN VERILOG CODE | 8-10 |
|  | TEST BENCH FILE | 10-13 |
|  | SCREEN SHOTS OF THE OUTPUT | 14-15 |

**ABSTRACT OF THE PROJECT:**

This paper presents the design of a 16-bit Arithmetic Logic Unit (ALU) using 4-to-1 multiplexers to perform bitwise shift and rotate operations. The ALU is capable of executing four fundamental operations on a 16-bit input data set: shift left, shift right, rotate left, and rotate right. Each operation is implemented using combinational logic and selectively routed through multiplexers to produce the desired result based on control signals. By utilizing multiplexers for operation selection, this ALU design achieves a simplified, efficient logic structure that minimizes hardware complexity while ensuring reliable operation.

The design incorporates a multiplexer for each bit in the 16-bit data path, allowing for dynamic selection of the desired operation through control signals. Specifically, two control bits are used to direct the multiplexer output, enabling the selection between shifting and rotating functionalities. The shift operations fill vacant positions with zeros, while the rotate operations wrap the displaced bits to the opposite end of the data, maintaining all original data values. This design approach demonstrates a methodical yet modular technique to implement fundamental ALU operations, suitable for various applications requiring compact and efficient digital processing units.

**Understanding 4-to-1 Multiplexer Configuration**

Each **4-to-1 multiplexer** has:

* **4 data inputs** (let’s label them as D0, D1, D2, D3).
* **2 select lines** (S1 and S0) to choose which input to output.
* **1 output**.

Based on the values of S1 and S0:

* If S1 S0 = 00, the multiplexer outputs D0.
* If S1 S0 = 01, the multiplexer outputs D1.
* If S1 S0 = 10, the multiplexer outputs D2.
* If S1 S0 = 11, the multiplexer outputs D3.

In our design:

* **D0** will be connected to the **Shift Left** output.
* **D1** will be connected to the **Shift Right** output.
* **D2** will be connected to the **Rotate Left** output.
* **D3** will be connected to the **Rotate Right** output.

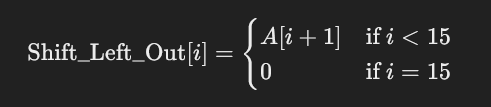
**Designing the Operations for Each Bit**

For each of the 16 bits, we need to define the following operations.

* 1. **Shift Left**

For a 1-bit **logical left shift**:

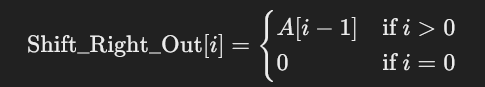
* Each bit is shifted left, and the **rightmost bit** (LSB) is set to **0**.
* For each bit i:



* 1. **Shift Right**

For a 1-bit **logical right shift**:

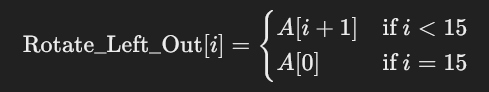
* Each bit is shifted right, and the **leftmost bit** (MSB) is set to **0**.
* For each bit i:



* 1. **Rotate Left**

For a 1-bit **rotate left**:

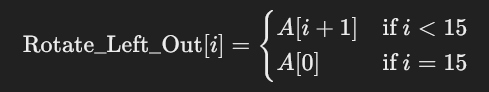
* Each bit is shifted left, with the **MSB wrapped around** to the LSB.
* For each bit i:



* 1. **Rotate Right**

For a 1-bit **rotate right**:

* Each bit is shifted right, with the **LSB wrapped around** to the MSB.
* For each bit i:



**Using 4-to-1 Multiplexers to Select Operations**

For each bit position iii (from 0 to 15), we’ll use a 4-to-1 multiplexer to choose between the four operations.

Let’s set:

* **D0** = Shift Left output for bit iii.
* **D1** = Shift Right output for bit iii.
* **D2** = Rotate Left output for bit iii.
* **D3** = Rotate Right output for bit iii.

Each multiplexer’s **S1** and **S0** lines will be controlled by two control signals C1 and C0:

* C1C0=00C1C0 = 00C1C0=00 for **Shift Left**.
* C1C0=01C1C0 = 01C1C0=01 for **Shift Right**.
* C1C0=10C1C0 = 10C1C0=10 for **Rotate Left**.
* C1C0=11C1C0 = 11C1C0=11 for **Rotate Right**

Breakup of the wires and ports used in the module and testbench:

**1. ALU Module**

**Ports:**

* **Inputs:**
  + input [15:0] A: This is a 16-bit input operand for the ALU. It represents the data on which the ALU will perform operations.
  + input [1:0] ALU\_Sel: This is a 2-bit selection input used to choose which operation the ALU will perform. Each value of this input corresponds to a different operation:
    - 00: Shift Left
    - 01: Shift Right
    - 10: Rotate Left
    - 11: Rotate Right
* **Output:**
  + output reg [15:0] ALU\_Out: This is a 16-bit output that holds the result of the ALU operations. It's declared as a reg because it is assigned in an always block.

**Internal Wires:**

* **Operation Results:**
  + wire [15:0] shift\_left\_result: This wire stores the result of the logical shift left operation.
  + wire [15:0] shift\_right\_result: This wire stores the result of the logical shift right operation.
  + wire [15:0] rotate\_left\_result: This wire stores the result of the rotate left operation.
  + wire [15:0] rotate\_right\_result: This wire stores the result of the rotate right operation.

**2. Testbench Module (tb\_ALU)**

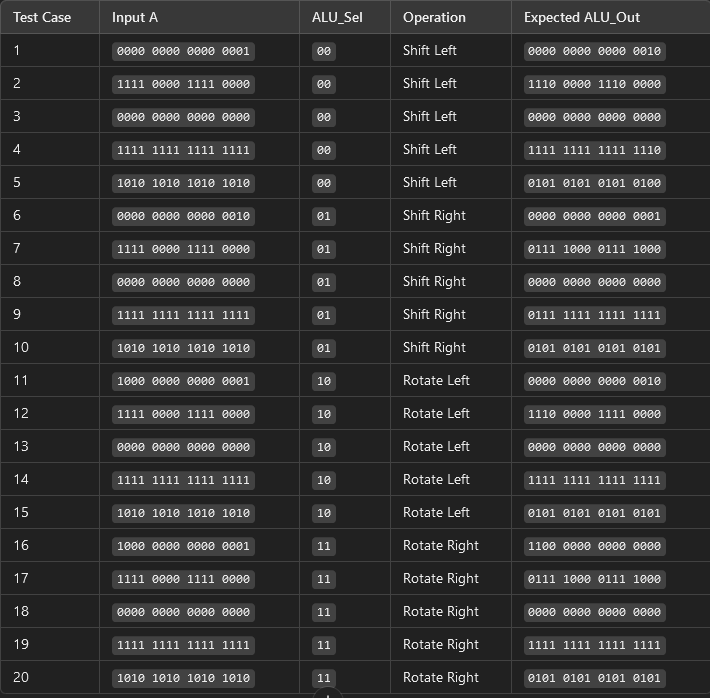
**Ports:**

* **Inputs:**
  + reg [15:0] A: This is a 16-bit register used to provide input values to the ALU during the simulation. It's declared as a reg because its value is driven within the testbench.
  + reg [1:0] ALU\_Sel: This is a 2-bit register that selects the operation for the ALU, similar to the input in the ALU module.
* **Output:**
  + wire [15:0] ALU\_Out: This is a 16-bit wire that captures the output from the ALU module during simulation.

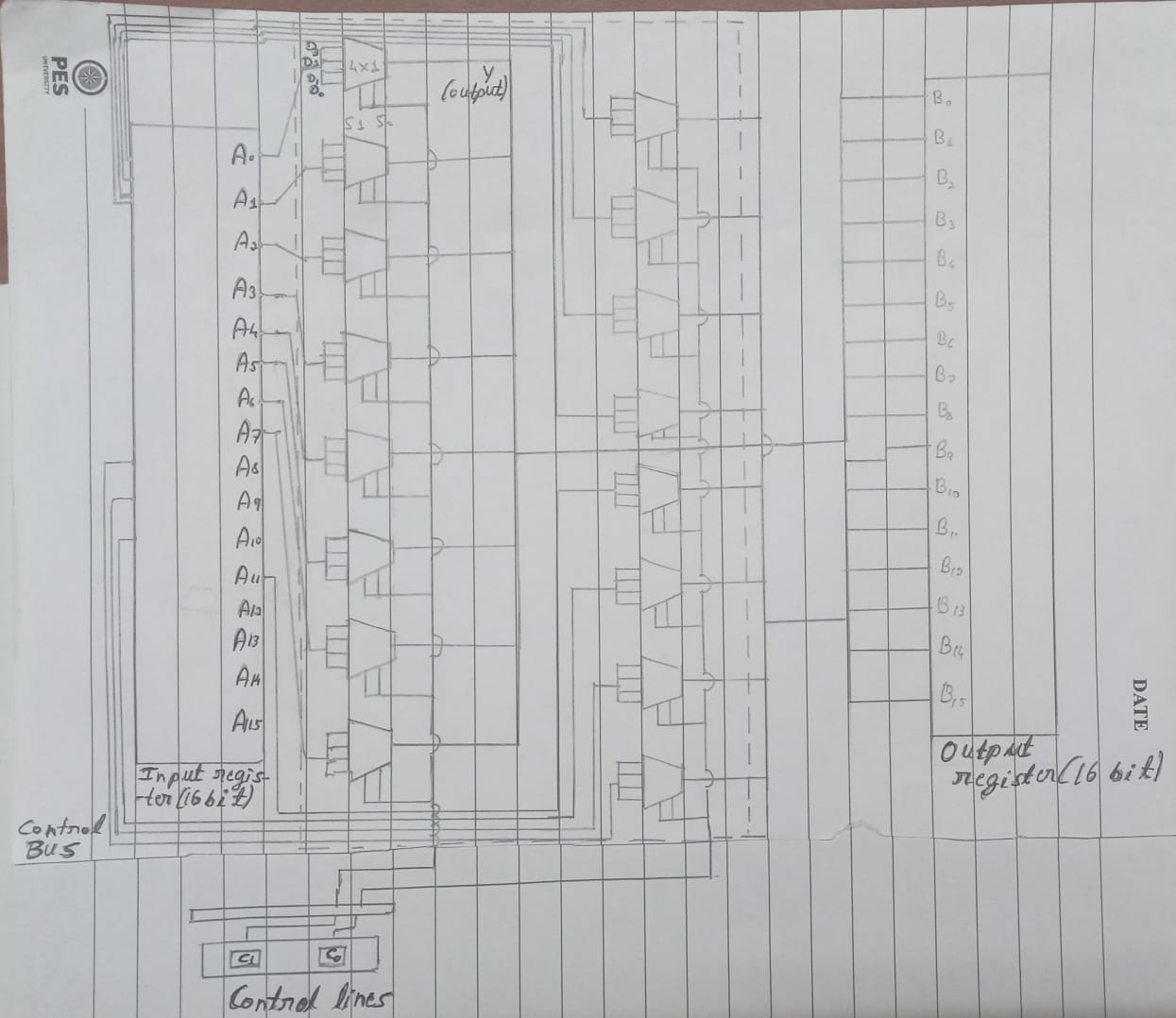
**Additional Simulation Features:**

* **Waveform Dumping:**
  + $dumpfile("alu\_waveform.vcd"): This command sets the name of the VCD (Value Change Dump) file that will be generated during simulation. It allows for visualization of the signals in simulation tools like GTKWave.
  + $dumpvars(0, tb\_ALU): This command instructs the simulator to dump all the variables in the tb\_ALU module to the VCD file. The argument 0 indicates that it should dump all variables from the top level of the testbench.

**Expected truth table for few of the testcases:**



**CIRCUIT DIAGRAM:**



**MAIN VERILOG CODE:(16bitALMUX.V)**

**module ALU (**

**input [15:0] A,**

**input [1:0] ALU\_Sel,**

**output [15:0] ALU\_Out**

**);**

**wire [15:0] shift\_left\_result;**

**wire [15:0] shift\_right\_result;**

**wire [15:0] rotate\_left\_result;**

**wire [15:0] rotate\_right\_result;**

**// Instantiate submodules for each operation**

**Shift\_Left shift\_left\_inst (.A(A), .Result(shift\_left\_result));**

**Shift\_Right shift\_right\_inst (.A(A), .Result(shift\_right\_result));**

**Rotate\_Left rotate\_left\_inst (.A(A), .Result(rotate\_left\_result));**

**Rotate\_Right rotate\_right\_inst (.A(A), .Result(rotate\_right\_result));**

**// Instantiate the 4x1 multiplexer to select the operation based on ALU\_Sel**

**Mux4x1 mux\_inst (**

**.in0(shift\_left\_result),**

**.in1(shift\_right\_result),**

**.in2(rotate\_left\_result),**

**.in3(rotate\_right\_result),**

**.sel(ALU\_Sel),**

**.out(ALU\_Out)**

**);**

**endmodule**

**module Shift\_Left (**

**input [15:0] A,**

**output [15:0] Result**

**);**

**assign Result = A << 1;**

**endmodule**

**module Shift\_Right (**

**input [15:0] A,**

**output [15:0] Result**

**);**

**assign Result = A >> 1;**

**endmodule**

**module Rotate\_Left (**

**input [15:0] A,**

**output [15:0] Result**

**);**

**assign Result = {A[14:0], A[15]};**

**endmodule**

**module Rotate\_Right (**

**input [15:0] A,**

**output [15:0] Result**

**);**

**assign Result = {A[0], A[15:1]};**

**endmodule**

**// 4x1 multiplexer to select one of the four operations based on ALU\_Sel**

**module Mux4x1 (**

**input [15:0] in0,**

**input [15:0] in1,**

**input [15:0] in2,**

**input [15:0] in3,**

**input [1:0] sel,**

**output reg [15:0] out**

**);**

**always @(\*) begin**

**case (sel)**

**2'b00: out = in0;**

**2'b01: out = in1;**

**2'b10: out = in2;**

**2'b11: out = in3;**

**default: out = 16'b0;**

**endcase**

**end**

**endmodule**

**TEST BENCH FILE:** (tb\_ALUMUX.v)

`timescale 1ns / 1ps

module tb\_ALUMUX;

// Testbench signals

reg [15:0] A; // 16-bit input A

reg [1:0] ALU\_Sel; // 2-bit select signal

wire [15:0] ALU\_Out; // 16-bit ALU output

// Instantiate the ALU module

ALU uut (

.A(A),

.ALU\_Sel(ALU\_Sel),

.ALU\_Out(ALU\_Out)

);

// Test procedure

initial begin

// Initialize waveform dump for simulation analysis

$dumpfile("16bitALMUX.vcd"); // Output VCD file

$dumpvars(0, tb\_ALUMUX); // Dump all variables in the tb\_ALU module

// Test Case Group 1: Shift Left

A = 16'b0000000000000001; // Simple shift of LSB

ALU\_Sel = 2'b00; // Select Shift Left operation

#10;

$display("Shift Left: A = %b, ALU\_Out = %b", A, ALU\_Out);

A = 16'b1111000011110000; // Pattern test

#10;

$display("Shift Left: A = %b, ALU\_Out = %b", A, ALU\_Out);

A = 16'b0000000000000000; // Edge case: All 0s

#10;

$display("Shift Left: A = %b, ALU\_Out = %b", A, ALU\_Out);

A = 16'b1111111111111111; // Edge case: All 1s

#10;

$display("Shift Left: A = %b, ALU\_Out = %b", A, ALU\_Out);

A = 16'b1010101010101010; // Alternating bits

#10;

$display("Shift Left: A = %b, ALU\_Out = %b", A, ALU\_Out);

// Test Case Group 2: Shift Right

A = 16'b0000000000000010; // Simple shift of second bit

ALU\_Sel = 2'b01; // Select Shift Right operation

#10;

$display("Shift Right: A = %b, ALU\_Out = %b", A, ALU\_Out);

A = 16'b1111000011110000; // Pattern test

#10;

$display("Shift Right: A = %b, ALU\_Out = %b", A, ALU\_Out);

A = 16'b0000000000000000; // Edge case: All 0s

#10;

$display("Shift Right: A = %b, ALU\_Out = %b", A, ALU\_Out);

A = 16'b1111111111111111; // Edge case: All 1s

#10;

$display("Shift Right: A = %b, ALU\_Out = %b", A, ALU\_Out);

A = 16'b1010101010101010; // Alternating bits

#10;

$display("Shift Right: A = %b, ALU\_Out = %b", A, ALU\_Out);

// Test Case Group 3: Rotate Left

A = 16'b1000000000000001; // Check MSB to LSB wrap-around

ALU\_Sel = 2'b10; // Select Rotate Left operation

#10;

$display("Rotate Left: A = %b, ALU\_Out = %b", A, ALU\_Out);

A = 16'b1111000011110000; // Pattern test

#10;

$display("Rotate Left: A = %b, ALU\_Out = %b", A, ALU\_Out);

A = 16'b0000000000000000; // Edge case: All 0s

#10;

$display("Rotate Left: A = %b, ALU\_Out = %b", A, ALU\_Out);

A = 16'b1111111111111111; // Edge case: All 1s

#10;

$display("Rotate Left: A = %b, ALU\_Out = %b", A, ALU\_Out);

A = 16'b1010101010101010; // Alternating bits

#10;

$display("Rotate Left: A = %b, ALU\_Out = %b", A, ALU\_Out);

// Test Case Group 4: Rotate Right

A = 16'b1000000000000001; // Check LSB to MSB wrap-around

ALU\_Sel = 2'b11; // Select Rotate Right operation

#10;

$display("Rotate Right: A = %b, ALU\_Out = %b", A, ALU\_Out);

A = 16'b1111000011110000; // Pattern test

#10;

$display("Rotate Right: A = %b, ALU\_Out = %b", A, ALU\_Out);

A = 16'b0000000000000000; // Edge case: All 0s

#10;

$display("Rotate Right: A = %b, ALU\_Out = %b", A, ALU\_Out);

A = 16'b1111111111111111; // Edge case: All 1s

#10;

$display("Rotate Right: A = %b, ALU\_Out = %b", A, ALU\_Out);

A = 16'b1010101010101010; // Alternating bits

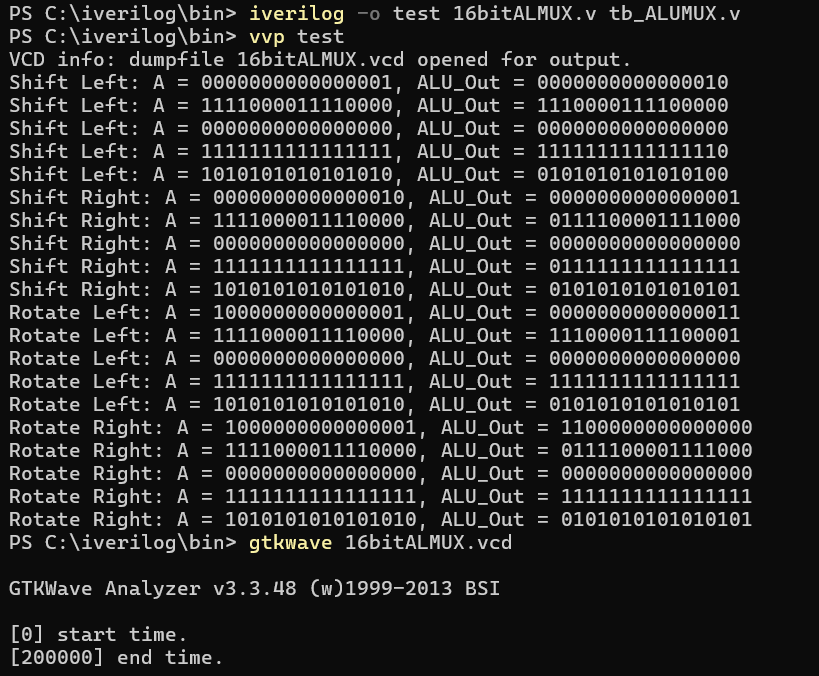
#10;

$display("Rotate Right: A = %b, ALU\_Out = %b", A, ALU\_Out);

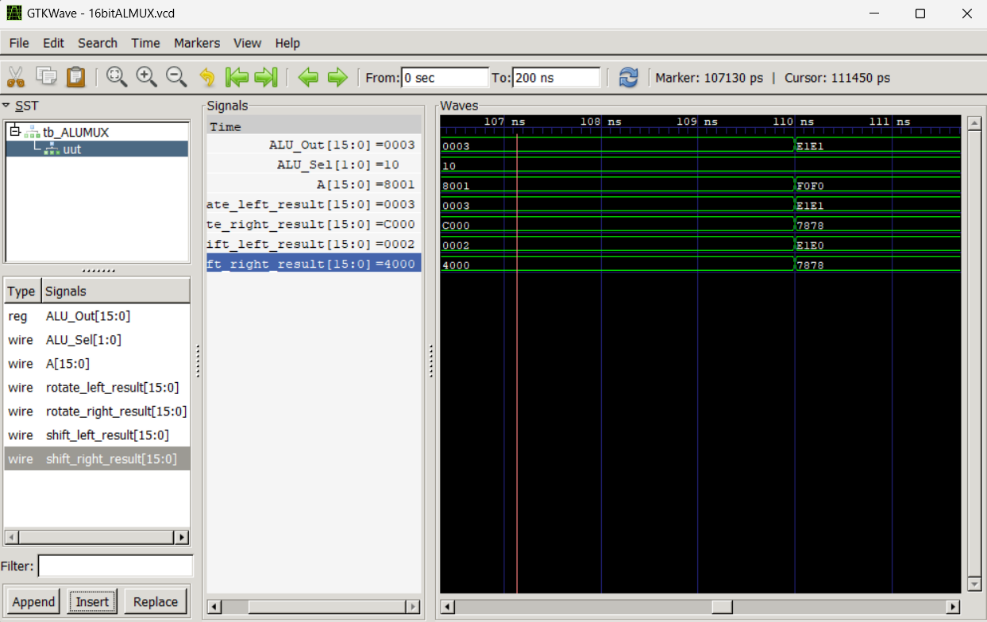
end

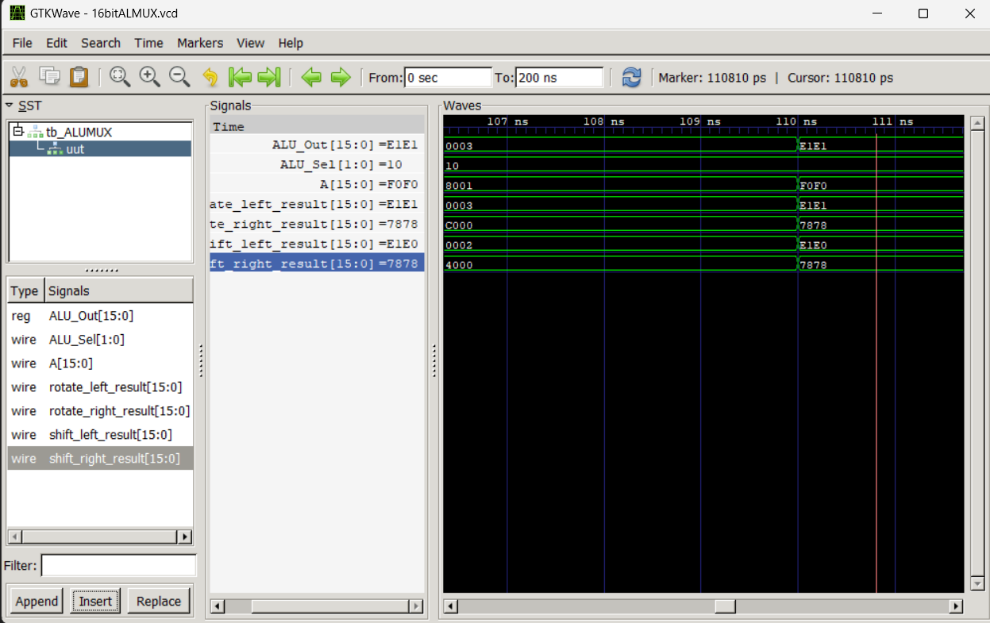
endmodule

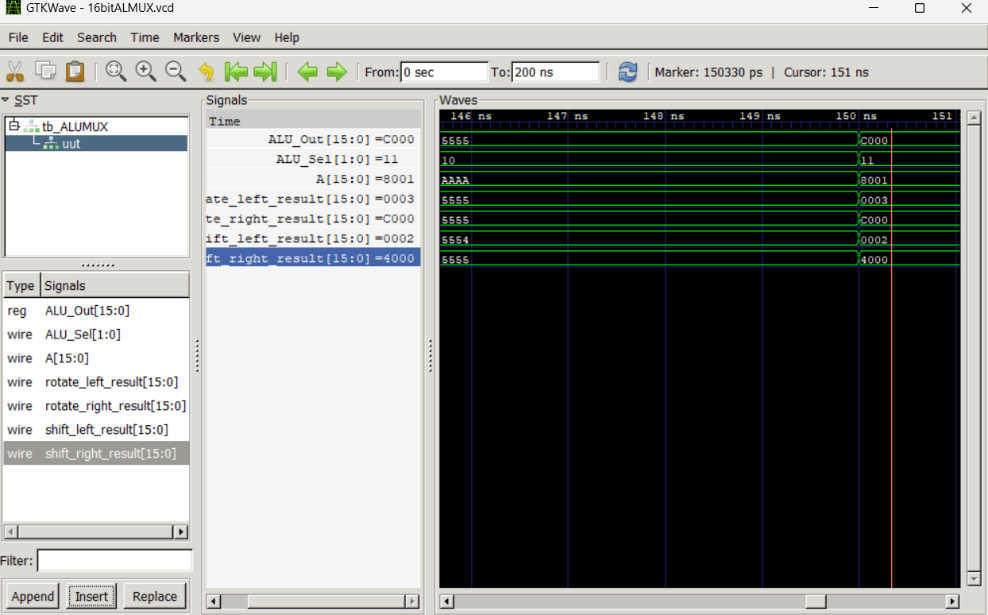
**SCREEN SHOT OF THE OUTPUT:**

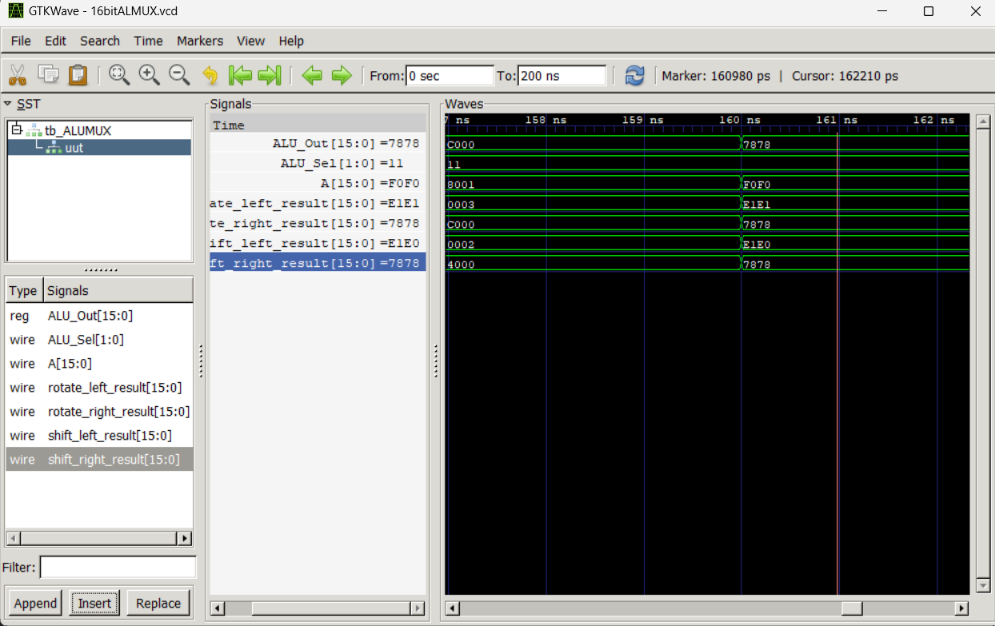
****

**GTKWAVE FORM OUTPUTS:**

****

****

****

****