Design and Simulation of Arithmetic and Logic Unit (ALU)

Alisha Chowhan
Department of Computer Science
San Jose State University
Email: alisha.chowhan@sjsu.edu

***Abstract*—The main objective of this project is to design and test a 32-bit Arithmetic and Logic Unit (ALU) using the ModelSim simulation tool. The following will be discussed in this report:**

1. Process of installing ModelSim and setting it up
2. Implementing the Arithmetic and Logic Unit (ALU) using the Verilog HDL
3. Testing functionality of the ALU module
4. Simulating and analyzing the waves from the ALU testbench

General Information: The tool used for simulation and testing is ModelSim.

I. STEPS TO INSTALL MODELSIM

1. There are many different simulation tools available, like Altera, Quartus, and ModelSim. Professor Patra provided a direction for the `ModelSimSetup-18.1.0.625-windows.exe` download option in an email, however the general steps would be to search for 'intel modelsim download'.
2. Visit www.intel.com and download ModelSim-Intel® FPGA Standard Edition, Version 18.1, or other versions, depending on your device. (Figure I.1)

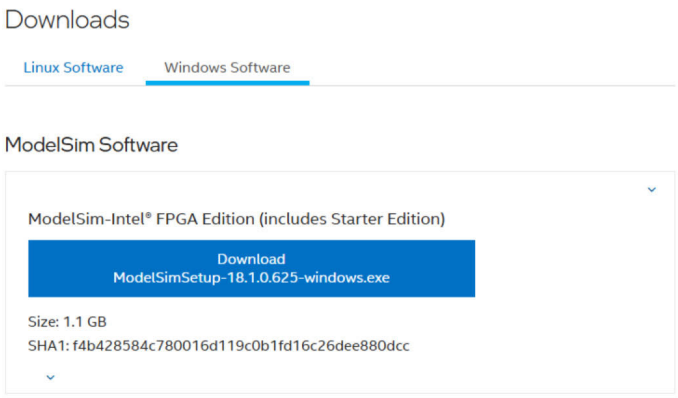


Fig. I.1. ModelSim Download Page

3. After downloading, unzip the file. Next, pick ModelSim - Intel FPGA Starter Edition (License is not required). The Student Version is no longer available.
4. Press `I accept the agreement`.

II. STEPS OF SIMULATION PROJECT CREATION

This section will cover creating the ALU Verilog Project using ModelSim to set up the environment.

Follow these steps below to run the ALU Verilog code and its associated testbench:

1. Download the zip file for this project and extract it into a new folder. The three files are alu.v, prj_01_tb.v, and prj_definitions.v.
   - alu.v has all the ALU module code (edits will be made in this file).
   - prj_01_tb.v has all the testbench code for the ALU verification (edits will be made in this file).
   - prj_definitions.v has all the macro definitions (no edits will be made in this file).
2. Open ModelSim and then navigate to the top left corner of the screen. Click File, then New, and select Project.
3. Enter a project name (ex. CS147MiniProj_ALU) and click OK. Shown in Figure II.1.

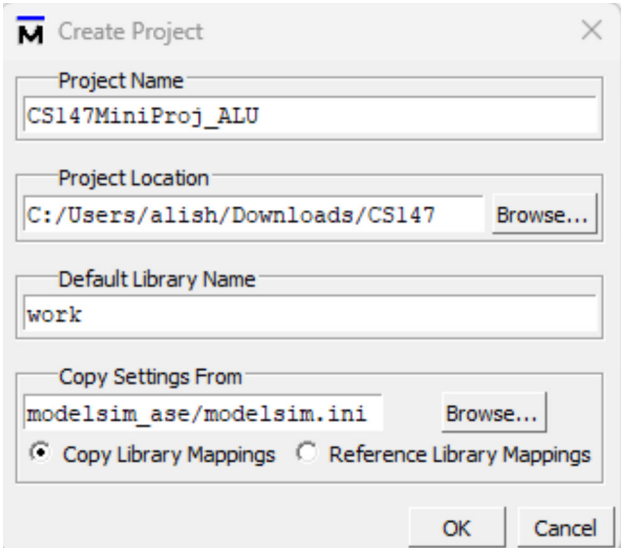


Fig. II.1. Creating a Project

4. Once the project is named, a window will pop up prompting to select what files need to be added. Select 'Add Existing File' as shown in Figure II.2. And then click 'Browse' for

the files in the directory as shown in Figure II.3. Select the three Verilog files as shown in Figure II.4.

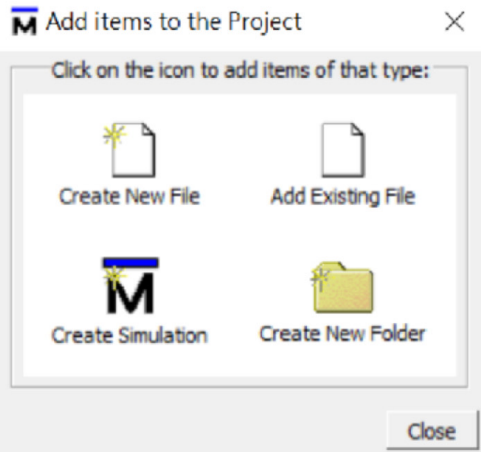


Fig. II.2. Add Existing File

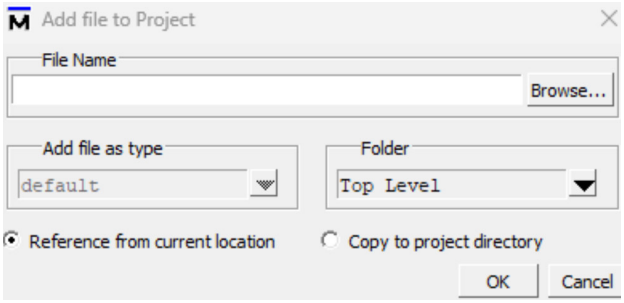


Fig. II.3. Add File to Project

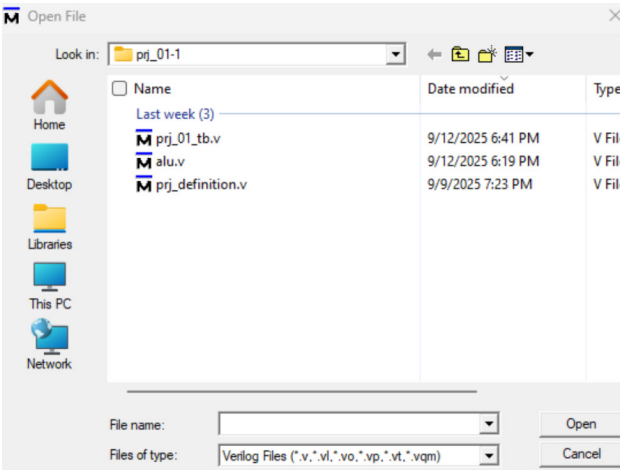


Fig. II.4. Select files to add into Project

5. Once the three files are added to the project click ‘Compile‘ (in red) after selecting each file one at a time. If successful, green check marks for each file will appear as in Figure II.5.

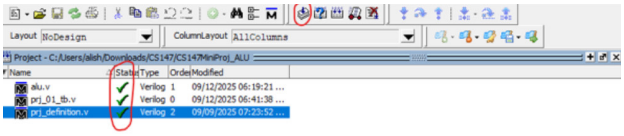


Fig. II.5. Compilation of Files Check

6. Next, click on the ‘Library’ tab located on the bottom left of the screen. Click ‘work’ and then navigate to ‘prj_01_tb.v’ and double click as shown in Figure II.6 below, and then a ‘sim-Default’ window will be shown like in Figure II.7 below.

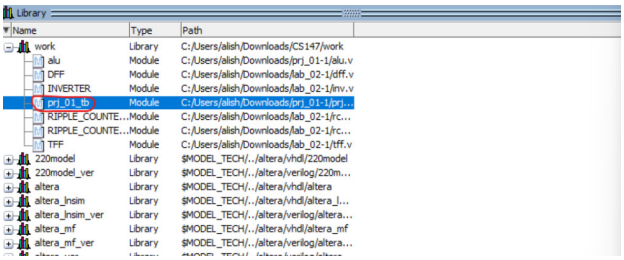


Fig. II.6. Code run Testbench

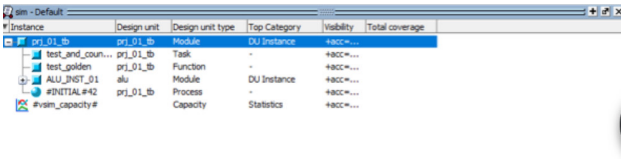


Fig. II.7. sim-Default Window

7. Click ‘ALU_INST_01’, click ‘View’ on the top left bar, and then ‘Objects’ from the top bar, which will open a window displaying op1,op2,oprn, and result. Select all the options, right-click, and select the ‘Add Wave as shown in Figure II.8 below.

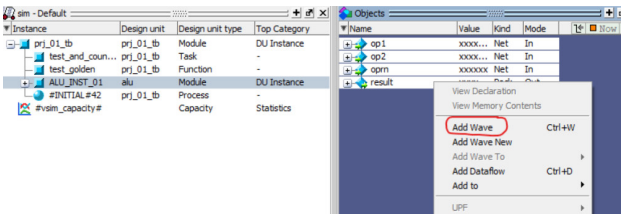


Fig. II.8. Object Window

8. Once the Wave windows are open. If another format is needed, select all the signals, right click, click ‘Radix,’ and the options to change to various formats like decimal, binary, or octal as shown, like in Figure II.9 below. Finally, press ‘Run All’(top right) to see the waveforms.

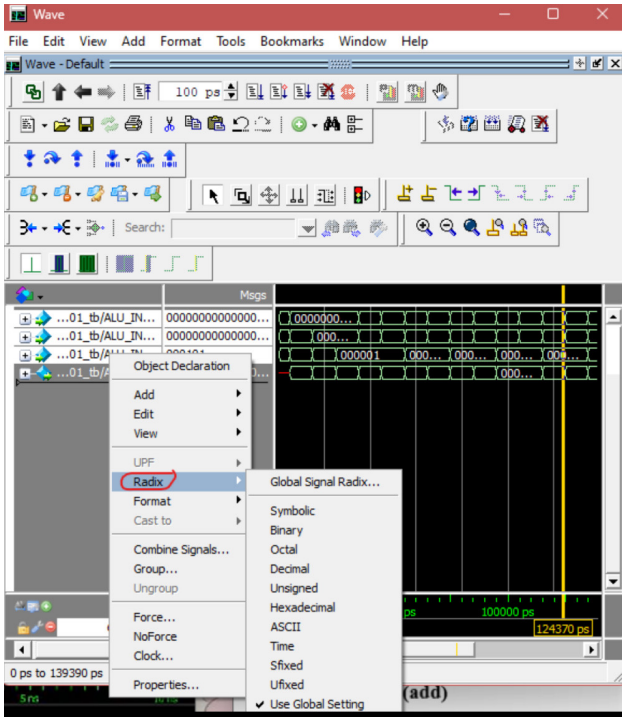


Fig. II.9. Waveform

III. REQUIREMENTS OF ALU

The ALU is a key digital component built from combinational logic. It carries out arithmetic operations like addition, subtraction, multiplication, and logical operations such as AND, OR, and NOR. Alongside this, there are also shift operations, including left and right, and comparisons such as 'set less than'. The ALU is a core building block within a computer's CPU due to these functions.

The CS 147DV instruction set[1] specifies the following arithmetic and logical operations that the ALU has to support. These operations that must be implemented are:

Name	Mnemonic	Format	Operation
Addition	`add`	R	R[rd] = R[rs] + R[rt]
Subtraction	`sub`	R	R[rd] = R[rs] - R[rt]
Multiplication	`mul`	R	R[rd] = R[rs] * R[rt]
Logical AND	`and`	R	R[rd] = R[rs] & R[rt]
Logical OR	`or`	R	R[rd] = R[rs] | R[rt]
Logical NOR	`nor`	R	R[rd] = ~(R[rs] | R[rt])
Set less than	`slt`	R	R[rd] = (R[rs] < R[rt])?1:0
Shift left logical	`sll`	R	R[rd] = R[rs] << shamt
Shift right logical	`srl`	R	R[rd] = R[rs] >> shamt

The CS147DV instruction set includes a 6-bit OpCode and funct that specifies which operation needs to be executed. For instance, if any opcode value were to be 0x01, which is combined with a specific function value, it would indicate an addition instruction. In the ALU design for this project, the control signal named '`oprn`' is used to select the operation to be executed. The list of operations are:

NAME	OPERATION CODE
add	1
sub	2
mul	3
srl	4
sll	5
and	6
or	7
nor	8
slt	9

IV. DESIGN AND IMPLEMENTATION OF ALU

The Arithmetic Logic Unit (ALU) was created in Verilog to handle a range of arithmetic and logical instructions. The design accepts two 32-bit inputs (op1 and op2) and a 6-bit control signal (oprn), which specifies the operation. Based on the opcode, the ALU produces a 32-bit output (result).

A. Design Strategy

The ALU is implemented as a Verilog module with defined inputs for operands, the control signal, and one output for the result. A visual representation is shown in Figure IV.1.

The input design is:

- **op1** – 32-bit operand
- **op2** – 32-bit operand
- **oprn** – 6-bit opcode selecting which operation is performed

The output is:

- **result** – 32-bit signal holding the operation outcome

```
module ALU(result, op1, op2, oprn);
    input [31:0] op1, op2;      //operands
    input [5:0] oprn;           //operation code
    output [31:0] result;       //output of ALU
    //case statement for operations implemented here
endmodule
```

Fig. IV.1. Design Strategy Visual

B. Implementation Details

In the implementation, all arithmetic and logical operations are handled inside a `case` block, which checks the value of `oprn`. For example, if `oprn = 6'h01`, the ALU performs addition (`op1 + op2`).

A mathematical understanding and visualization of how the ALU supports these essential operations is shown as(screenshots of code snippets):

Addition: The addition of op1 and op2 is performed when the oprn code is h01.

```
`ALU_OPRN_WIDTH'h01 : result = op1 + op2; //addition
```

Subtraction: Subtraction is signaled when oprn code is h02.

```
`ALU_OPRN_WIDTH'h02 : result = op1 - op2; //subtraction
```

Multiplication: Performed when oprn code is h03.

```
`ALU_OPRN_WIDTH'h03 : result = op1 * op2; //multiplication
```

Logical Shift Right: Performed when oprn code is h04.

```
`ALU_OPRN_WIDTH'h04 : result = op1 >> op2; //shift right
```

Logical Shift Left: Performed when oprn code is h05.

```
`ALU_OPRN_WIDTH'h05 : result = op1 << op2; //shift left
```

Bitwise Operations: AND, OR, NOR happen when oprn is h06, h07, h08, respectively.

```
`ALU_OPRN_WIDTH'h06 : result = op1 & op2; //and
`ALU_OPRN_WIDTH'h07 : result = op1 | op2; //or
`ALU_OPRN_WIDTH'h08 : result = ~(op1 | op2); //nor
```

Set Less Than (SLT): This operation compares op1 and op2, returning 1 if op1 is less than op2 and otherwise returns 0. Triggered by opcode h09.

```
`ALU_OPRN_WIDTH'h09 : result = (op1 < op2) ? 1:0; //set less than
```

V. TEST STRATEGY AND IMPLEMENTATION

To make sure the ALU functions properly, a testbench is built to run through different arithmetic and logical operations. These tests would include addition, subtraction, multiplication, shifts, and bitwise functions. For each case, the values used are `op1`(input 1), `op2` (input 2), and `oprn`(type of operation) to set up the operation. The output is then compared to the 'expected result' or a "golden result". Using the `test_and_count` to keep track of how many tests were run and how many actually passed. The outputs are shown both in the simulation console and the waveforms confirm the timing and correctness of the ALU behavior. The console correctness is displayed below in Figure V.1.

```
# [TEST] 15 add(+) 3 = 18 , got 18 ... [PASSED]
# [TEST] 15 sub(-) 5 = 10 , got 10 ... [PASSED]
# [TEST] 15 add(+) 5 = 20 , got 20 ... [PASSED]
# [TEST] 7 add(+) 9 = 16 , got 16 ... [PASSED]
# [TEST] 250 add(+) 50 = 300 , got 300 ... [PASSED]
# [TEST] 40 sub(-) 12 = 28 , got 28 ... [PASSED]
# [TEST] 500 sub(-) 275 = 225 , got 225 ... [PASSED]
# [TEST] 6 mul(*) 8 = 48 , got 48 ... [PASSED]
# [TEST] 12 mul(*) 7 = 84 , got 84 ... [PASSED]
# [TEST] 64 LRS(>>) 3 = 8 , got 8 ... [PASSED]
# [TEST] 128 LRS(>>) 4 = 8 , got 8 ... [PASSED]
# [TEST] 3 SHL(<<) 3 = 24 , got 24 ... [PASSED]
# [TEST] 9 SHL(<<) 2 = 36 , got 36 ... [PASSED]
# [TEST] 25 AND(&) 14 = 8 , got 8 ... [PASSED]
# [TEST] 60 AND(&) 19 = 16 , got 16 ... [PASSED]
# [TEST] 25 OR(|) 14 = 31 , got 31 ... [PASSED]
# [TEST] 60 OR(|) 19 = 63 , got 63 ... [PASSED]
# [TEST] 20 NOR 5 = 4294967274 , got 4294967274 ... [PASSED]
# [TEST] 30 NOR 12 = 4294967265 , got 4294967265 ... [PASSED]
# [TEST] 8 SLT(<) 15 = 1 , got 1 ... [PASSED]
# [TEST] 22 SLT(<) 11 = 0 , got 0 ... [PASSED]
#
#       Total number of tests          21
#       Total number of pass           21
```

Fig. V.1. Console Test Cases Results

Note: Each operation was tested with two or more different cases, and all results appear in the console output in Figure 5.1. For simplicity, only one test case per operation is shown in the waveform figures below.

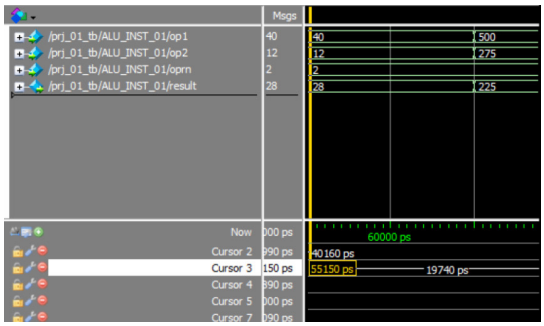
A. Analyzing Waves from ALU

The waveforms from the ALU testbench show how the inputs are taken in, including op1, op2, and oprn, to produce the correct result for each of the nine operations. Below are all the different test cases to confirm that the ALU performs as expected for the arithmetic and logic functions.

1. **ADDITION:** $15 + 3 = 18$

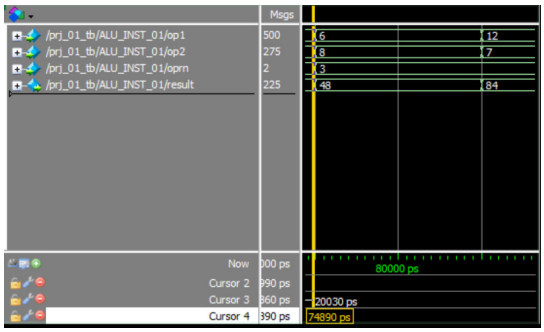
In the waveform, the input operands are op1=15, op2=3, and oprn set to 01, which is addition. At 5050 ps, the result is shown, which is 18. This is the correct output, hence confirming that the ALU properly handles addition.

2. **SUBTRACTION:** $40 - 12 = 28$



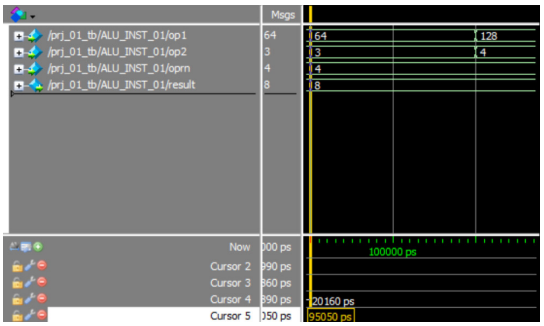
In the waveform, the input operands are op1=40, op2=12, and oprn set to 02, which is subtraction. Resulting in 28, which is correct, hence confirming the ALU handles subtraction.

3. **MULTIPLY:** $6 * 8 = 48$



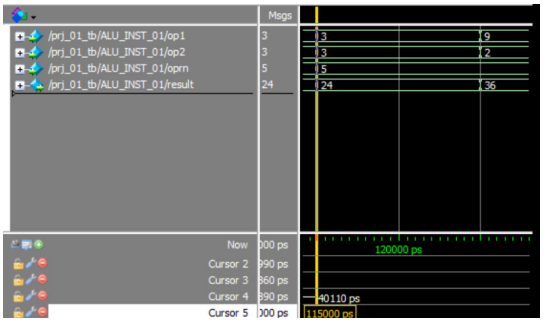
In the waveform, the input operands are op1=6, op2=8, and oprn set to 03, which is multiplication. Resulting in 48, which is correct, hence confirming the ALU handles multiplication.

4. **LOGICAL SHIFT RIGHT:** $64 \gg 3 = 8$



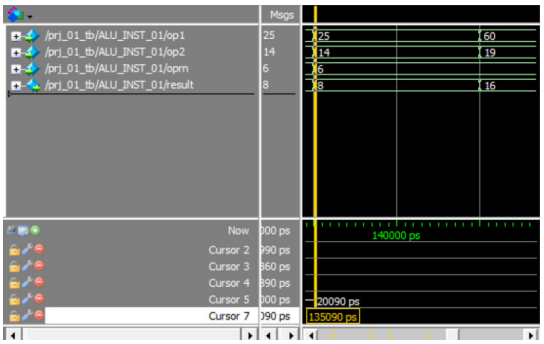
In the waveform, the input operands are op1=64, op2=3, and oprn set to 04, which is logical shift right. Resulting in 8, which is correct, hence confirming the ALU handles logical shift right.

5. **LOGICAL SHIFT LEFT:** $3 \ll 3 = 24$



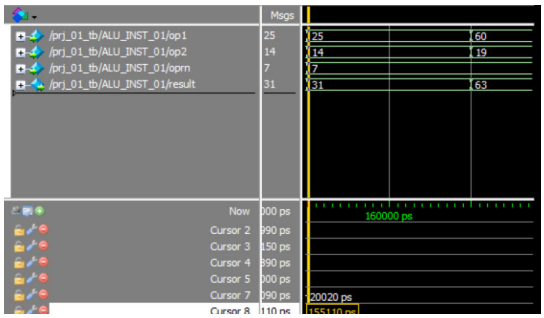
In the waveform, the input operands are op1=3, op2=3, and oprn set to 05, which is logical shift left. Resulting in 24, which is correct, hence confirming the ALU handles logical shift left.

6. **BITWISE AND:** $25 \& 14 = 8$



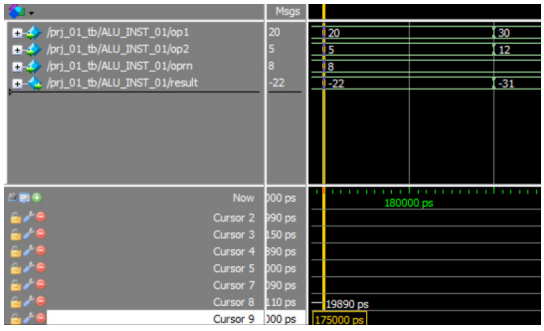
In the waveform, the input operands are op1=25, op2=14, and oprn set to 06, which is bitwise AND. Resulting in 8, which is correct, hence confirming the ALU handles bitwise AND.

7. **BITWISE OR:** $25 \mid 14 = 31$



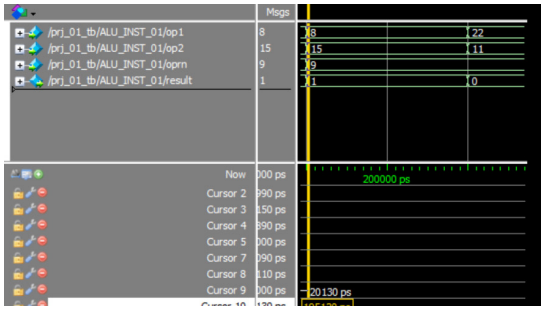
In the waveform, the input operands are op1=25, op2=14, and oprn set to 07, which is bitwise OR. Resulting in 31, which is correct, hence confirming the ALU handles bitwise OR.

8. **BITWISE NOR:** 20 NOR 5 $= -22$ (signed decimal) $= 4294967274$ (unsigned decimal)



In the waveform, the input operands are op1=20, op2=5, and oprn set to 08, which is bitwise NOR. Resulting in -22, which is correct, hence confirming the ALU handles bitwise NOR.

9. **SET LESS THAN:** $8 < 15 = 1$



In the waveform, the input operands are op1=8, op2=15, and oprn set to 09, which is bitwise set less than. Resulting in 1, which is correct, hence confirming the ALU handles set less than.

VI. Conclusion

The simulation confirmed that the ALU works correctly and handles the arithmetic and logical operations: addition, subtraction, multiplication, shifts, and bitwise functions correctly. The testbench also displayed that every operation produced a matching result to the expected output.

In conclusion, this project proved the functionality, reliability, and importance of the ALU for digital systems.

REFERENCES

[1] K. Patra, CS 147, Lecture 01: Computer Systems, Instruction Set, ALU Slides.