Arithmetic and Logic Unit(ALU) Simulator

Luan Nguyen

Department of Computer Science, San Jose State University, San Jose, USA

luan.nguyen03@sjsu.edu

***Abstract*—This report paper explains the development of the Arithmetic and Logic Unit(ALU) using the simulation software, Modelsim.**

I. Introduction

The ALU is a fundamental hardware component of the Central Processing Unit(CPU). It is responsible for performing simple arithmetic and logic operations. This report will focus on simulating the functionality of the ALU using the Modelsim software. This report will include the following:

1) Steps on how to install the Modelsim software.
2) Steps to set up the project environment.
3) Requirements of the ALU.
4) Design and implementation of the ALU.
5) Test strategy and test implementation.
6) Outputs in waveform and text formats.
7) Conclusion/Closure.

II. Modelsim Installation

For this specific report, the simulation software that will be used to develop this ALU is ModelSim-Intel® FPGAs Standard Edition Software Version 20.1.1. To install this software, follow these steps:

1)Open this link or paste in the internet browser: https://www.intel.com/content/www/us/en/software-kit/750666/modelsim-intel-fpgas-standard-edition-software-version-20-1-1.html. This will direct to the exact software package and version.

2)This software is compatible with both Windows and Linux. Select the operating system and download the appropriate file into a temporary directory.

3) Open the download file and follow the installation steps. The download file can be deleted after the process is complete.

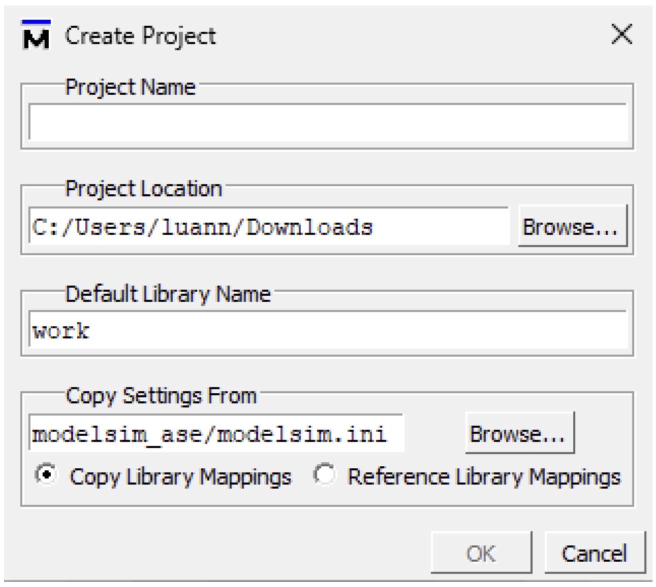
This software package does require a license for full functionality, but the trial/evaluation mode provides enough functionality to complete this ALU simulator and is legal to use for personal and education purposes. For MacOS users, there are two alternatives to install this software:

- Use a cloud-based virtual Windows/Linux machine provided by Microsoft Azure. Verified students can get up to $100 credit when creating a free account. Use this link: https://azure.microsoft.com/en-us/free/students.
- Use a local virtual Windows/Linux machine using UTM virtualization software. Use this link: https://mac.getutm.app/. This software is free to use.

III. Setting up project environment

A. Create and Set Up

Once the application is opened, navigate to “File” located at the top left, then “New”, finally “Project”; the “Create Project” window from Figure 3.1 should appear. Start by filling out the project name and project location. Finally, confirm with the “OK” button.



Project Location: C:/Users/luann/Downloads

Default Library Name: work

Copy Settings From: modelsim_ase/modelsim.ini

[✓ Copy Library Mappings] Reference Library Mappings

Figure 3.1. Create Project Window

Once the project is created, the window from Figure 3.2 should appear.

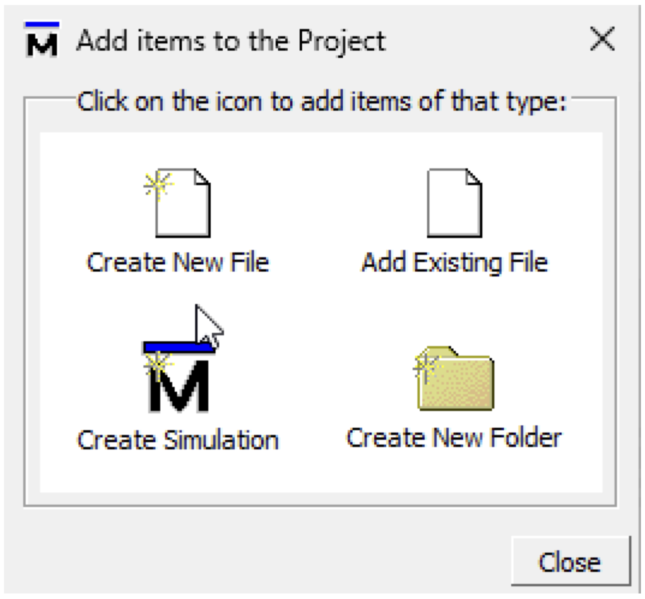


Figure 3.2. Add Items Window

Before adding items to this project, head over to the project assignment page of this course and download the zip file and extract them out into a discoverable directory. These files are:

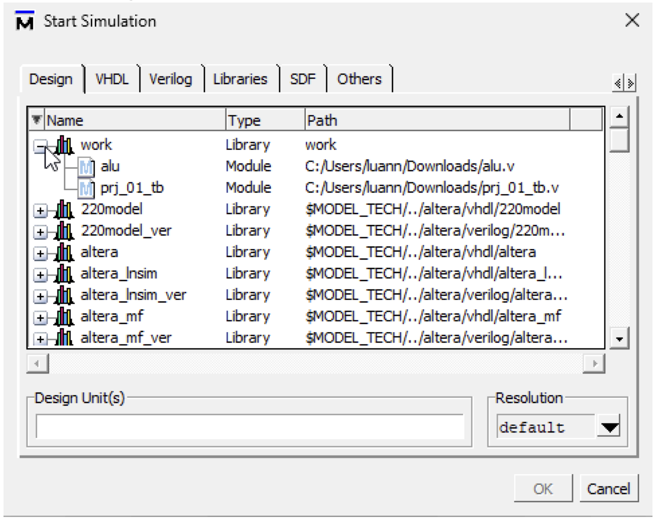
- alu.v: where the alu operations are implemented.
- prj_01_tb.v: where test cases are implemented.
- prj_definition.v: where size limits for data and operands are initialized and do not need to be modified.

Then, navigate to "Add Existing File" from the image above and add said files.

B. *Compile and Simulate*

Before simulating a program, it is important to save and compile the code beforehand. After modifying any file, navigate to "File" and "Save", or Ctrl + S, then navigate "Compile" and "Compile All".

To simulate a program, click the "Simulate" Button on the navigation bar and "Start Simulation".


Start Simulation

Design | VHDL | Verilog | Libraries | SDF | Others

Name	Type	Path
work	Library	work
alu	Module	C:/Users/luann/Downloads/alu.v
prj_01_tb	Module	C:/Users/luann/Downloads/prj_01_tb.v
220model	Library	$MODEL_TECH/../altera/vhdl/220model
220model_ver	Library	$MODEL_TECH/../altera/verilog/220m...
altera	Library	$MODEL_TECH/../altera/vhdl/altera
altera_lnsim	Library	$MODEL_TECH/../altera/vhdl/altera_l...
altera_lnsim_ver	Library	$MODEL_TECH/../altera/verilog/altera...
altera_mf	Library	$MODEL_TECH/../altera/vhdl/altera_mf
altera_mf_ver	Library	$MODEL_TECH/../altera/verilog/altera...

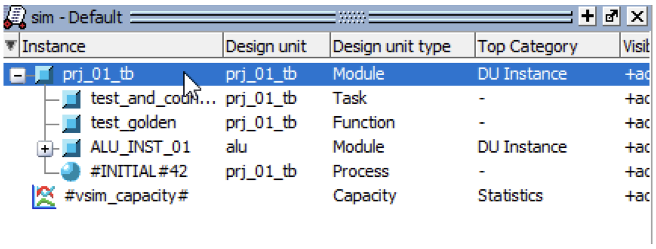
Design Unit(s)

Resolution: default

OK Cancel

Figure 3.3. Start Simulation Window

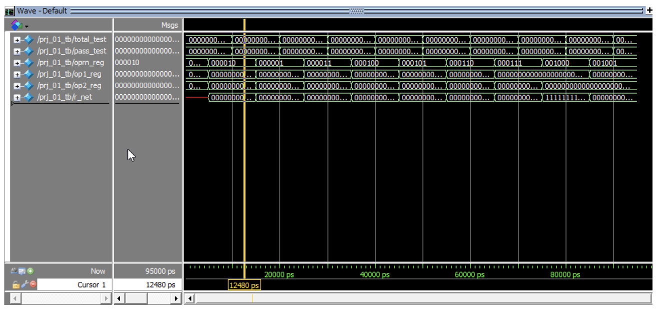
Once opened, navigate to the "+" icon next to "work" and select "prj_01_tb", which is the file containing test cases, then confirm with "OK".


sim - Default

Instance	Design unit	Design unit type	Top Category	Visit
prj_01_tb	prj_01_tb	Module	DU Instance	+ac
test_and_coun...	prj_01_tb	Task	-	+ac
test_golden	prj_01_tb	Function	-	+ac
ALU_INST_01	alu	Module	DU Instance	+ac
#INITIAL#42	prj_01_tb	Process	-	+ac
#vsim_capacity#		Capacity	Statistics	+ac

3.4. Simulation Window

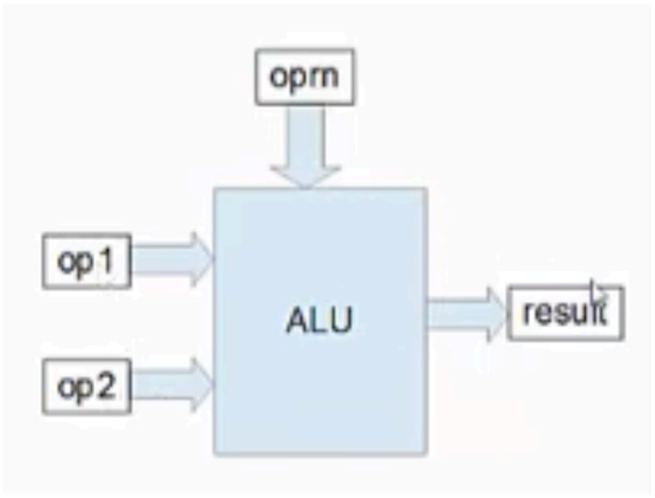
To add a wave, navigate to the simulation tab above and right click on "prj_01_tb", then"Add Wave". After clicking the "Zoom Full" magnifying glass icon, the default waveform window should look similar to the image in Figure 3.5.



3.5. Waveform Window

IV. Requirements of The ALU

As explained briefly in the introduction, the ALU is a crucial hardware component of the CPU that handles all the basic arithmetic and logic operations. The image in Figure 4.1 is a simplified diagram of the ALU:



4.1. Simplified ALU Diagram

The ALU takes in operand 1, operand 2, and an operator and inputs and produces the result as the output. For this specific implementation of the ALU, the following operations and their corresponding hex code are supported:

- Addition(0x1)
- Subtraction(0x2)
- Multiplication(0x3)
- Shift Right Logical(0x4)
- Shift Left Logical(0x5)
- And Logical(0x6)
- Or Logical(0x7)
- Nor Logical(0x8)
- Set Less Than(0x9)

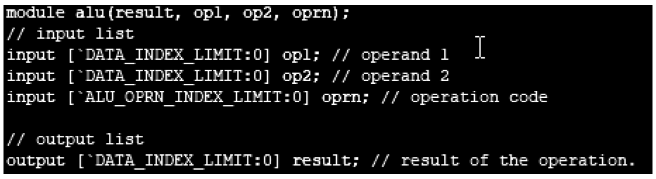
Other requirements include the data size of each input and output in bits. Operand 1, Operand 2, and result can hold a 32-bit value, while operator can hold a 6-bit value as specified in the prj_definition.v file.

In addition to the Modelsim application, basic understanding of Hardware Description Language(HDL) is required, specifically Verilog(VHDL). Verilog is a HDL used to mode, design, and simulate digital circuits.

V. Design and Implementation of The ALU

A. Design Specifications

The design of the ALU is simple. Start by declaring the ALU as a module, which is a fundamental building block of Verilog design. Modules can represent anything from a simple to complex digital circuit system. The sample code provided by alu.v has already specified this design as shown below:



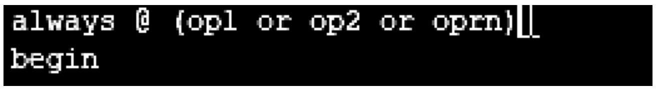
```
module alu(result, op1, op2, oprn);
// input list
input [`DATA_INDEX_LIMIT:0] op1; // operand 1
input [`DATA_INDEX_LIMIT:0] op2; // operand 2
input [`ALU_OPRN_INDEX_LIMIT:0] oprn; // operation code

// output list
output [`DATA_INDEX_LIMIT:0] result; // result of the operation.
```

Figure 5.1. Input and Output Ports

Notice that there are 3 input ports and 1 output port. Op1 and Op2 are represented in 32 bits, while the oprn is represented in 6 bits. The result port is also represented in 32 bits, where the result of op1 (oprn) op2 is stored.

B. Implementation of the ALU

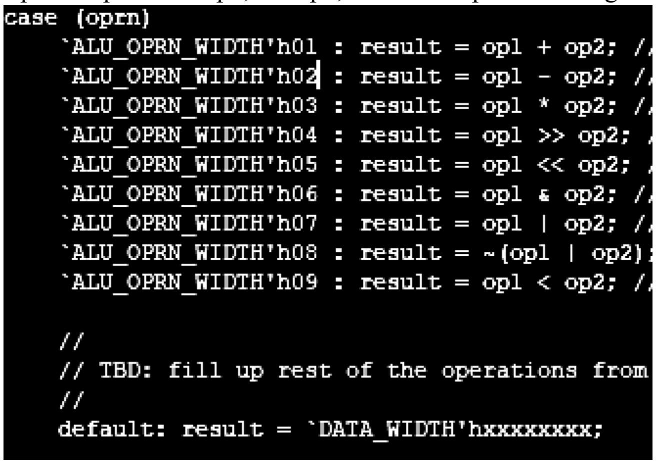
To implement this ALU, start by declaring an *always* code block in the alu.v file. This code block is event-driven, meaning its execution is triggered whenever the certain conditions are met, similar to a while loop in high level programming languages. The specific implementation is shown below:



```
always @ (op1 or op2 or oprn)
begin
```

Figure 5.2. Always Code Block

The code within this block will execute whenever the input ports op1, op2, or oprn changes.



```
case (oprn)
    `ALU_OPRN_WIDTH'h01 : result = op1 + op2; /
    `ALU_OPRN_WIDTH'h02 : result = op1 - op2; /
    `ALU_OPRN_WIDTH'h03 : result = op1 * op2; /
    `ALU_OPRN_WIDTH'h04 : result = op1 >> op2;
    `ALU_OPRN_WIDTH'h05 : result = op1 << op2;
    `ALU_OPRN_WIDTH'h06 : result = op1 & op2; /
    `ALU_OPRN_WIDTH'h07 : result = op1 | op2; /
    `ALU_OPRN_WIDTH'h08 : result = ~(op1 | op2)
    `ALU_OPRN_WIDTH'h09 : result = op1 < op2; /

    //
    // TBD: fill up rest of the operations from
    //
    default: result = `DATA_WIDTH'hxxxxxxxx;
```

Figure 5.3. Case Code Block

Within the code block, declare a case statement, similar to a switch or if statements. This is where different operations are implemented. The detailed explanation of each operation are as follows:

1) Addition: Takes in 2 operands and adds them together. The result is stored in the result port. The operation code for addition is 6'h01 in hexadecimal as specified in the requirements.
2) Subtraction: Takes in 2 operands and subtracts op1 from op2. The result is stored in the result port. The operation code for subtraction is 6'h02 in hexadecimal as specified in the requirements.
3) Multiplication: Takes in 2 operands and multiplies them. The result is stored in the result port. The operation code for multiplication is 6'h03 in hexadecimal as specified in the requirements.
4) Shift Right Logical: Takes in 2 operands and shifts operand 1 to the right by operand 2. The result is stored in the result port. The operation code for shift right logical is 6'h04 in hexadecimal as specified in the requirements. In Verilog, the operation keyword for shift right logical is denoted as "A >> B" where A in binary format will be shifted right B times. Rightmost bits that are shifted out of bound are discarded. While the leftmost bits are filled with zeros.
5) Shift Left Logical: Takes in 2 operands and shifts operand 1 to the right by operand 2. The result is stored in the result port. The operation code for shift left logical is 6'h05 in hexadecimal as specified in the requirements. In Verilog, the operation keyword for shift left logical is denoted as "A << B" where A in binary format will be shifted left B times. Leftmost bits that are shifted out of bound are discarded. While the rightmost bits are filled with zeros.
6) And Logical: Takes in 2 operands and performs an and bitwise operation on them. The result is then stored in the result port. In Verilog, the operation keyword for and is "&".
7) Or Logical: Takes in 2 operands and performs an or bitwise operation on them. The result is then stored in the result port. In Verilog, the operation keyword for or is "|".
8) Nor Logical: Takes in 2 operands and performs a nor bitwise operation on them. The result is then stored in the result port. In Verilog, the operation keyword for or is "~(op 1 | op2)", or in plain text, not of or operation.
9) Set Less Than: Take in 2 operands and compare them. If op1 is less than op2, the operation returns 1, and 0 otherwise. The returned value is stored in the result port. The operation keyword for set less than is "<" in Verilog.

For unidentified operations, the default result is x. The exact Verilog implementations are shown in Figure 5.3.

VI. Test Strategy and Test Implementation

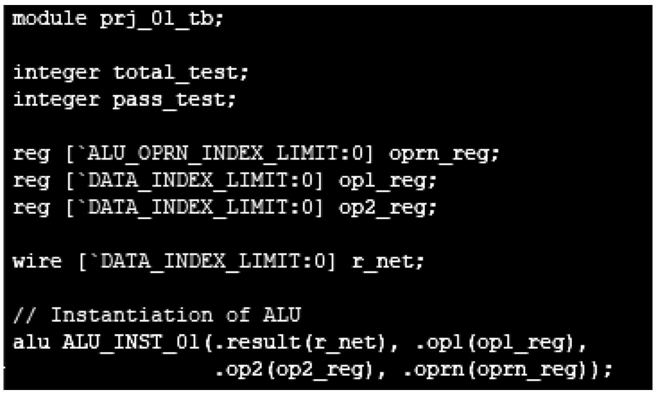
After completing the ALU implementation in alu.v file. It is important to test these operations to ensure that they are functioning properly. These tests will be written in prj_01_tb.v.

A. Test Strategy

To test the ALU, a test bench file is required. This is what prj_01_tb.v is for. This is where test cases will be run, the test bench will run their own operations then compare its result with the ALU operations. Each operation code will be tested once, totalling 9 test cases.

B. Test Implementation

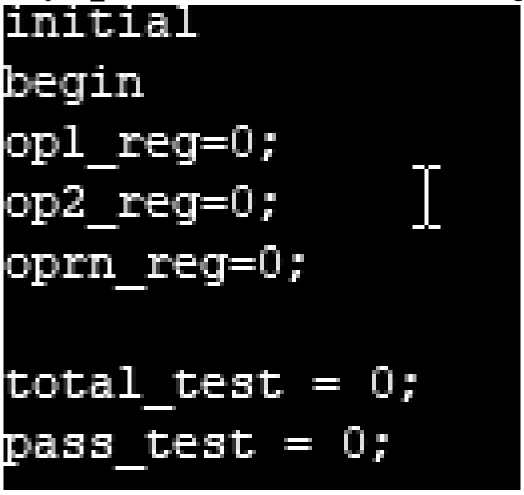
Similar to the ALU implementation, the test bench is also represented as a module, with 4 variables. Then, instantiate the ALU with the appropriate parameters as shown in Figure 6.1.



```
module prj_01_tb;

integer total_test;
integer pass_test;

reg [`ALU_OPRN_INDEX_LIMIT:0] oprn_reg;
reg [`DATA_INDEX_LIMIT:0] op1_reg;
reg [`DATA_INDEX_LIMIT:0] op2_reg;

wire [`DATA_INDEX_LIMIT:0] r_net;

// Instantiation of ALU
alu ALU_INST_01(.result(r_net), .op1(op1_reg),
                .op2(op2_reg), .oprn(oprn_reg));
```

Figure 6.1. Variables and ALU Instantiation

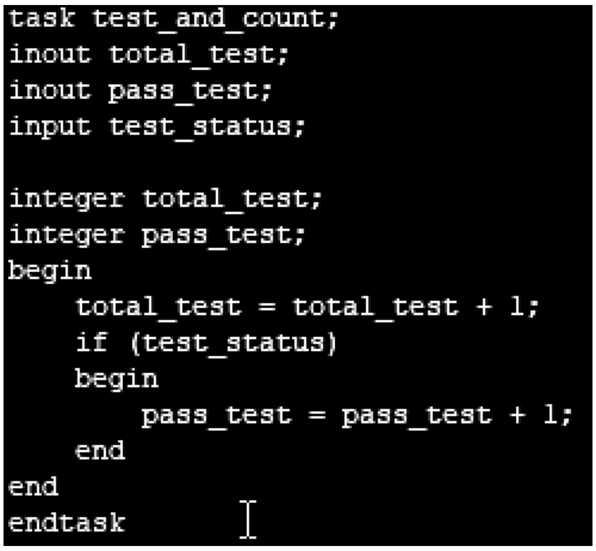
To code the test cases, start with an initial code block that will run once at the beginning of the program. Op1_reg, op2_reg, and oprn_reg are the 3 variables where test cases inputs are stored. Since there are multiple tests, total_test and pass_test are also introduced as shown in Figure 6.2.



```
initial
begin
op1_reg=0;
op2_reg=0;
oprn_reg=0;

total_test = 0;
pass_test = 0;
```

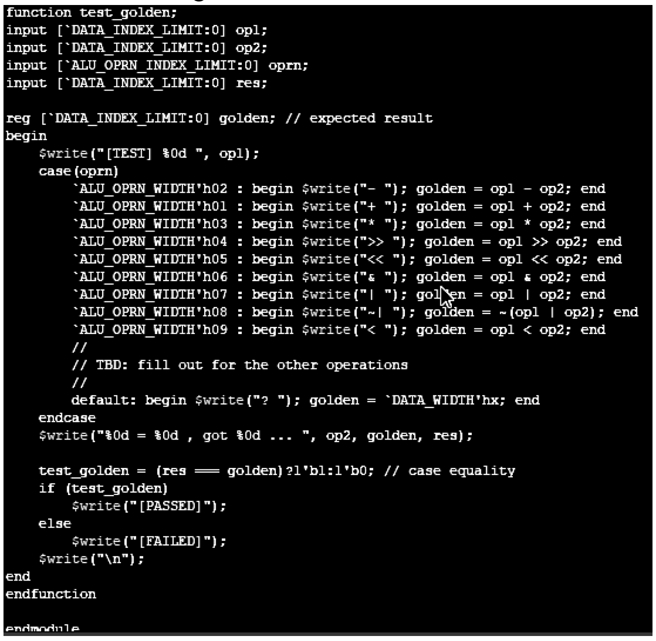
Figure 6.2. Initial Code Block

Before going into the test cases, it is important to mention a helper task and a helper function. The task code block shown in Figure 6.3. This is where the total_test and pass_test are processed. test_golden function is used to calculate the expected result and compare it with the result from the ALU, then output the operation in text and determine whether it passes or fails.



```
task test_and_count;
inout total_test;
inout pass_test;
input test_status;

integer total_test;
integer pass_test;
begin
    total_test = total_test + 1;
    if (test_status)
    begin
        pass_test = pass_test + 1;
    end
end
endtask
```

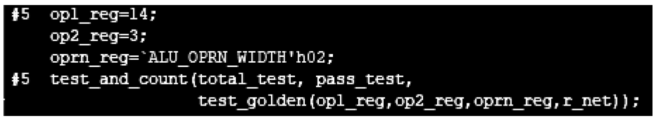
Figure 6.3. Task Code Block



```
function test_golden;
input [`DATA_INDEX_LIMIT:0] op1;
input [`DATA_INDEX_LIMIT:0] op2;
input [`ALU_OPRN_INDEX_LIMIT:0] oprn;
input [`DATA_INDEX_LIMIT:0] res;

reg [`DATA_INDEX_LIMIT:0] golden; // expected result
begin
    $write("[TEST] %0d ", op1);
    case(oprn)
        `ALU_OPRN_WIDTH'h02 : begin $write("- "); golden = op1 - op2; end
        `ALU_OPRN_WIDTH'h01 : begin $write("+ "); golden = op1 + op2; end
        `ALU_OPRN_WIDTH'h03 : begin $write("* "); golden = op1 * op2; end
        `ALU_OPRN_WIDTH'h04 : begin $write(">> "); golden = op1 >> op2; end
        `ALU_OPRN_WIDTH'h05 : begin $write("<< "); golden = op1 << op2; end
        `ALU_OPRN_WIDTH'h06 : begin $write("& "); golden = op1 & op2; end
        `ALU_OPRN_WIDTH'h07 : begin $write("| "); golden = op1 | op2; end
        `ALU_OPRN_WIDTH'h08 : begin $write("~| "); golden = ~(op1 | op2); end
        `ALU_OPRN_WIDTH'h09 : begin $write("< "); golden = op1 < op2; end
        //
        // TBD: fill out for the other operations
        //
        default: begin $write("? "); golden = `DATA_WIDTH'hx; end
    endcase
    $write("%0d = %0d , got %0d ... ", op2, golden, res);

    test_golden = (res === golden)?1'b1:1'b0; // case equality
    if (test_golden)
        $write("[PASSED]");
    else
        $write("[FAILED]");
    $write("\n");
end
endfunction

endmodule
```

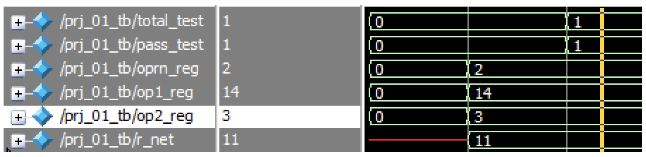
Figure 6.4. test_golden Function

1) Test Case for Subtraction:



```
#5  op1_reg=14;
    op2_reg=3;
    oprn_reg=`ALU_OPRN_WIDTH'h02;
#5  test_and_count(total_test, pass_test,
                   test_golden(op1_reg,op2_reg,oprn_reg,r_net));
```

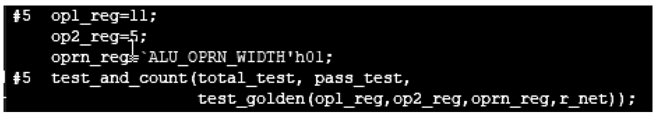
Waveform:



```
/prj_01_tb/total_test   1    0         1
/prj_01_tb/pass_test    1    0         1
/prj_01_tb/oprn_reg     2    0    2
/prj_01_tb/op1_reg      14   0    14
/prj_01_tb/op2_reg      3    0    3
/prj_01_tb/r_net        11        11
```

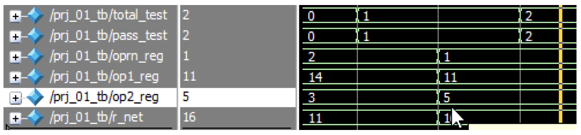
Explanation: 14 - 3 = 11, total_test++, pass_test++ since the result is expected.

2) Test Case for Addition:



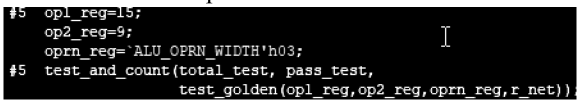
```
#5  op1_reg=11;
    op2_reg=5;
    oprn_reg=`ALU_OPRN_WIDTH'h01;
#5  test_and_count(total_test, pass_test,
                   test_golden(op1_reg,op2_reg,oprn_reg,r_net));
```

Waveform:



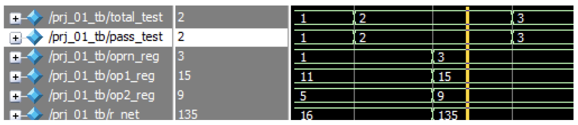
Explanation: 11 + 5 = 16, total_test++, pass_test++ since the result is expected.

3) Test Case for Multiplication:



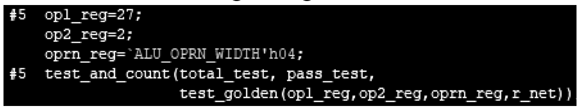
```
#5  op1_reg=15;
    op2_reg=9;
    oprn_reg=`ALU_OPRN_WIDTH'h03;
#5  test_and_count(total_test, pass_test,
                   test_golden(op1_reg,op2_reg,oprn_reg,r_net))
```

Waveform:



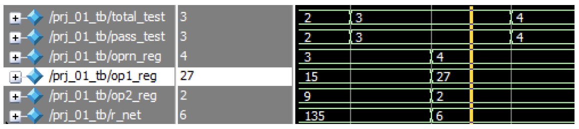
Explanation: 15 * 9 = 135.

4) Test Case for Shift Right Logical:



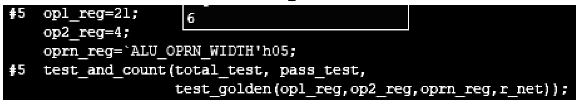
```
#5  op1_reg=27;
    op2_reg=2;
    oprn_reg=`ALU_OPRN_WIDTH'h04;
#5  test_and_count(total_test, pass_test,
                   test_golden(op1_reg,op2_reg,oprn_reg,r_net))
```

Waveform:



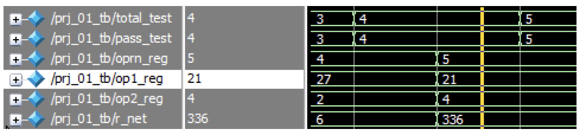
Explanation: 27 >> 2 = 6

5) Test Case for Shift Left Logical:



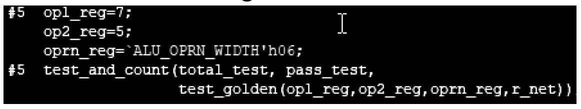
```
#5  op1_reg=21;
    op2_reg=4;
    oprn_reg=`ALU_OPRN_WIDTH'h05;
#5  test_and_count(total_test, pass_test,
                   test_golden(op1_reg,op2_reg,oprn_reg,r_net));
```

Waveform:



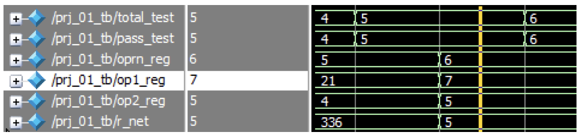
Explanation: 21 << 4 = 336

6) Test Case for AND Logical:



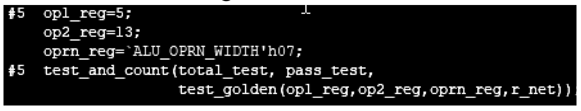
```
#5  op1_reg=7;
    op2_reg=5;
    oprn_reg=`ALU_OPRN_WIDTH'h06;
#5  test_and_count(total_test, pass_test,
                   test_golden(op1_reg,op2_reg,oprn_reg,r_net))
```

Waveform:



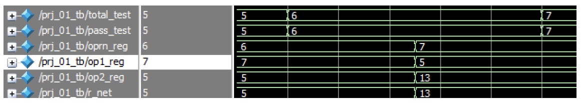
Explanation: 7 & 5 = 5

7) Test Case for OR Logical:



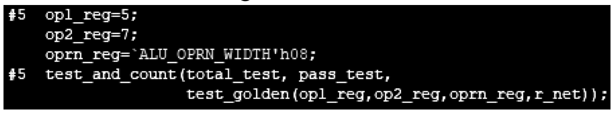
```
#5  op1_reg=5;
    op2_reg=13;
    oprn_reg=`ALU_OPRN_WIDTH'h07;
#5  test_and_count(total_test, pass_test,
                   test_golden(op1_reg,op2_reg,oprn_reg,r_net))
```

Waveform:



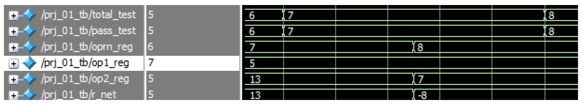
Explanation: 5 | 13 = 13

8) Test Case for NOR Logical:



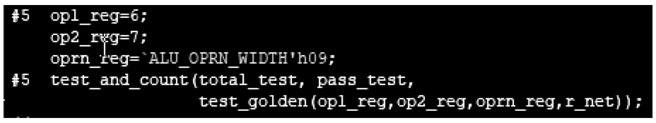
```
#5  op1_reg=5;
    op2_reg=7;
    oprn_reg=`ALU_OPRN_WIDTH'h08;
#5  test_and_count(total_test, pass_test,
                   test_golden(op1_reg,op2_reg,oprn_reg,r_net));
```

Waveform:



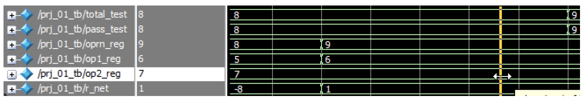
Explanation: ~(5 | 7) = -8. In unsigned representation, the number will be translated as 4294967288.

9) Test Case for Set Less Than:



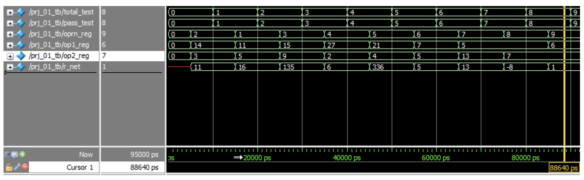
```
#5  op1_reg=6;
    op2_reg=7;
    oprn_reg=`ALU_OPRN_WIDTH'h09;
#5  test_and_count(total_test, pass_test,
                   test_golden(op1_reg,op2_reg,oprn_reg,r_net));
```

Waveform:

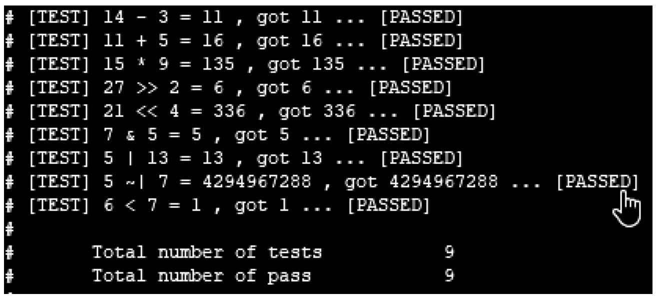


Explanation: 6 < 7 = 1 since the expression is true.

Output of all test cases in waveform:



Output of all test cases in text:



```
# [TEST] 14 - 3 = 11 , got 11 ... [PASSED]
# [TEST] 11 + 5 = 16 , got 16 ... [PASSED]
# [TEST] 15 * 9 = 135 , got 135 ... [PASSED]
# [TEST] 27 >> 2 = 6 , got 6 ... [PASSED]
# [TEST] 21 << 4 = 336 , got 336 ... [PASSED]
# [TEST] 7 & 5 = 5 , got 5 ... [PASSED]
# [TEST] 5 | 13 = 13 , got 13 ... [PASSED]
# [TEST] 5 ~| 7 = 4294967288 , got 4294967288 ... [PASSED]
# [TEST] 6 < 7 = 1 , got 1 ... [PASSED]
#
#       Total number of tests          9
#       Total number of pass           9
```

VII. Conclusion

From this project, I learned how to set up a Windows virtual machine on my MacOS to download Modelsim. I've also learned the basic functionality of ModelSim and syntax of Verilog HDL. I was able to create a simulation of the ALU using what I've learned, as well as creating test cases and reading waveforms.