Implementation of ALU in ModelSim

Joseph Nguyen
San Jose State University
College of Science, Computer Science
joseph.q.nguyen@sjsu.edu

***Abstract*—this report explores the implementation of Arithmetic and Logic Unit using an ALU module. The implementation will be written in Verilog language using ModelSim simulator.**

I. Introduction

The objectives of this project are to learn how to download and install ModelSim and to implement and test an ALU module using Verilog language. Additionally, we will also learn how to simulate and observe signal wave forms from the test bench.

II. Getting Started

A. Installing ModelSim

ModelSim is a Window-only simulator tool by MentorGraphics that can be installed for free at https://www.mentor.com/company/higher_ed/modelsim-student-edition. Click on the "Download Student Edition" to start downloading the simulator (you might have to create a free account). Once the file is done downloading, run it and follow the directions for installation. Once you have finish installation, you must now request for a license in order to use the program. A browser window should open up prompting you fill in some information. Once you are done, submit the form and a file should be sent to the email address you provided. If you for some reason did not receive the file, check your spam folder, otherwise you must redo the installation process. Once you have received the file "student_license.dat" save it to the installation directory for ModelSim PE Student Edition, for instance "C:\Modeltech_pe_edu_10.4a." This directory should include the sub-directory "win32pe_edu." Do not edit "student_license.dat" or else the license will not work. You should now be able to run ModelSim.

B. Setting up the Project

Once you have ModelSim installed, download and unzip the given zip file "proj_1.zip" from the following canvas link, https://sjsu.instructure.com/courses/1265088/assignments/4778137?module_item_id=9675706. After opening the zip file, there should be the following v files:

1) alu.v
2) prj_01_tb.v
3) prj_definition.v

Launch ModelSim, once the program is opened, go to the top left and click on File, then click New and then click Project. A window should pop up for you to name the project and also choose its directory. Choose an appropriate name and directory and then click OK.

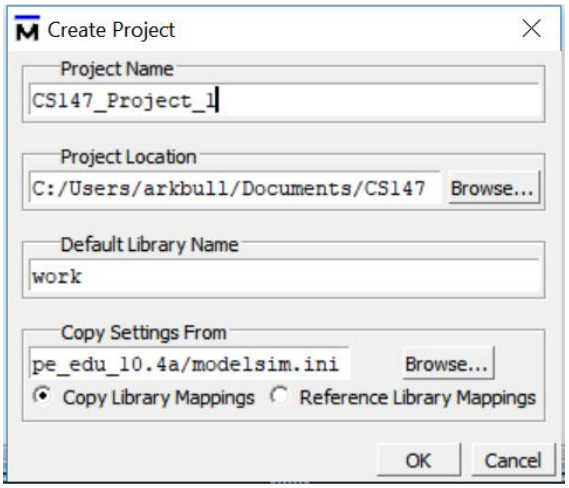

Fig. 1. Create Project window in ModelSim

Once you click OK, click on Add Existing File and locate where you downloaded the three source files.

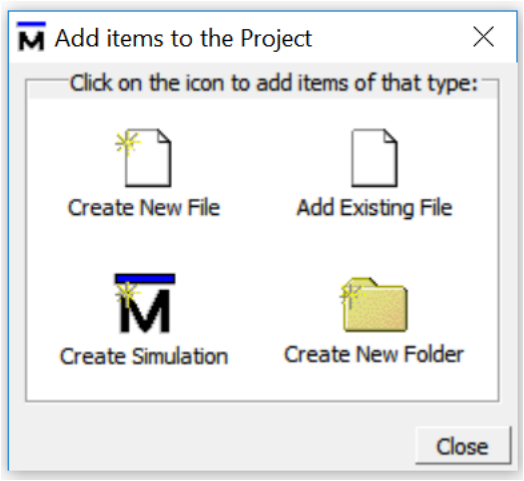

Fig. 2. Add items to the Project window

You will then select the all three of the files and then open them up.

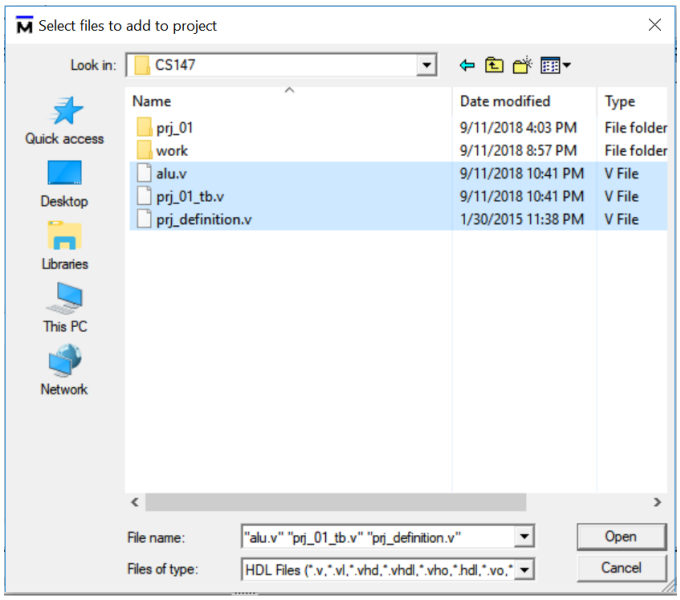

Fig. 3. Selecting files window

Once you have in finished loading the files, make sure that you are on the Project tab. There should be blue question marks for the three source files. This is because they have not been compiled yet. To compile it, go to the top left to where it says Compile, click on that and then click on Compile All. Now the three files should have a green check mark for the status.

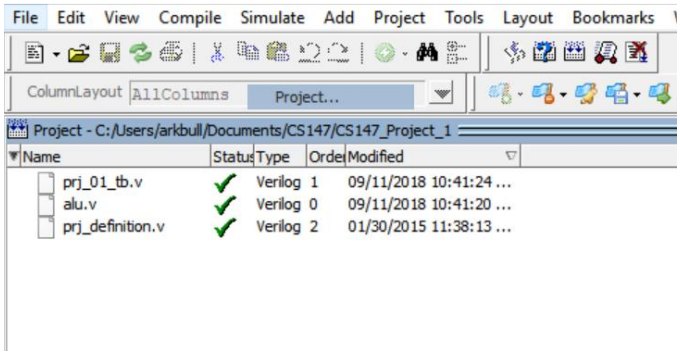

Fig. 4. Status of the compiled files

To run a simulation, click on the Libary tab next to the Project tab. Then click on the plus sign of "work." You should see alu and prj_01_tb and possibly some other files if you previously imported other code. The file that we want to click on is prj_01_tb.

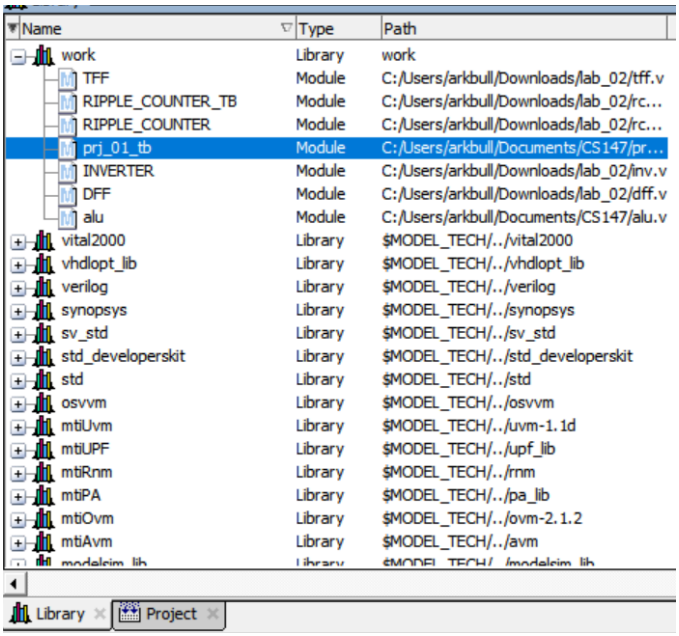

Fig. 5. Options to choose from before running a simulation

Once you double click prj_01_tb, the layout of the program should change a little bit. You should see an Objects tab on the right and in this tab you will see six items, select all of these items and then right click and select Add Wave.

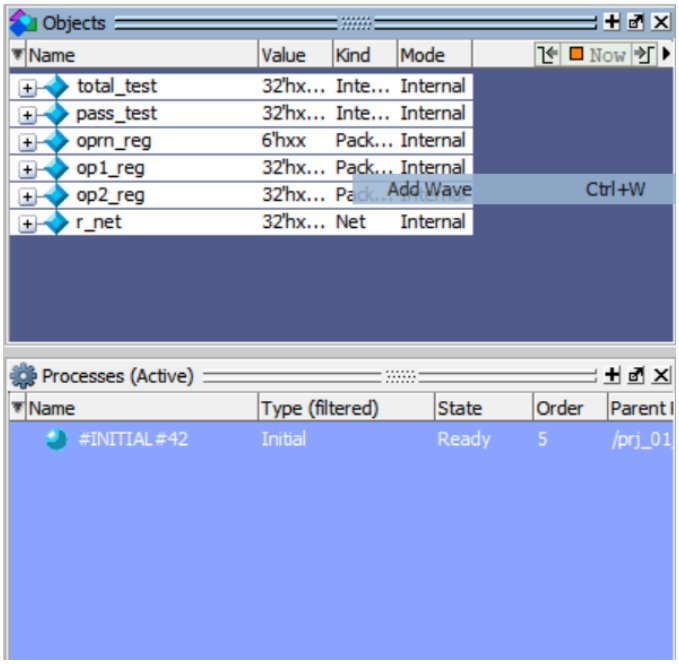

Fig. 6. Objects tab with highlighted selections

Once you are at the wave window, you should see some icons at the top. Click on the one that says Run –all in order to run the simulation. Additionally, it would be better to view the numbers in decimal, so highlight all six of the items in the wave window, right click and click on Radix, then click on

decimal. We will go back to analyzing the waves later on. Now we ready to edit the code.

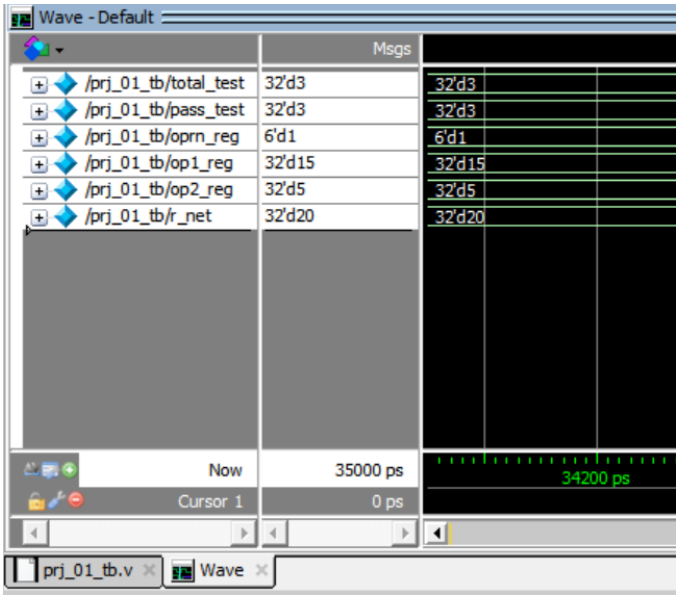

Fig. 7. Wave tab with Radix of numbers changed to decimal

III. Requirments of ALU

The goal of the program is to create and test an ALU module that would take three inputs and give an output. The results will be printed out and also shown in a wave format.

A. *module alu(result, op1, op2, oprn)*

1) *Inputs:*
   a. op1 – The first operand
   b. op2 – The second operand
   c. oprn – The operation symbol (0x1 – 0x9)
2) *Return values:*
   a. result – the result of the operation of the two op1 and op2 based on the operation code oprn. The valid operation codes are as followed:
      i. add (0x1)
      ii. sub (0x2)
      iii. mul(0x3)
      iv. shift_right (0x4)
      v. shift_left (0x5)
      vi. bitwise AND (0x6)
      vii. bitwise OR (0x7)
      viii. bitwise NOR (0x8)
      ix. set less than (0x9)

B. *module prj_01_tb*

3) *Purpose:*
   a. *This module is responsible for creating the test cases for each operation*

C. *function test_golden*

4) *Inputs:*
   a. *op1 – the first operand*
   b. *op2 – the second operand*
   c. *oprn – the operation code*
   d. *res – the result from the alu module*
5) *Purpose*
   a. *This function will take the test cases from prj_01_tb and calculate a result using alu, and then it will compare the result with an expected value "golden." It will keep track of test cases that pass and fail and print them out on the transcript.*

IV. Implementation of ALU

To implement these modules and function, we will be using Verilog language. The files that we will be editing are "alu.v" and "prj_01_tb.v." "prj_definition.v" defines the constants that we will be using and it has already been implemented so we do not have to edit it.

A. *Implementing model alu*

In "alu.v," most of the code has already been done for us. All we need to do is add the code for the rest of the cases. As explained earlier, this module takes two integer operands and based on the operation code that is provided, it will do an operation between the two numbers. For instance, if the numbers are 5 and 2 and the operation code is 0x3 (0x means that the number is in hexadecimal), then 5 * 3 = 15. The way we implement the cases is by using a switch statement as shown.

```
// Whenever op1, op2 or oprn changes do something
always @ (op1 or op2 or oprn)
begin
    case (oprn)
        `ALU_OPRN_WIDTH'h01 : result = op1 + op2; // addition
        `ALU_OPRN_WIDTH'h02 : result = op1 - op2; // subtraction
        `ALU_OPRN_WIDTH'h03 : result = op1 * op2; // multiplication
        `ALU_OPRN_WIDTH'h04 : result = op1 >> op2; // shift right
        `ALU_OPRN_WIDTH'h05 : result = op1 << op2; // shift left
        `ALU_OPRN_WIDTH'h06 : result = op1 & op2; // bit-wise AND
        `ALU_OPRN_WIDTH'h07 : result = op1 | op2; // bit-wise OR
        `ALU_OPRN_WIDTH'h08 : result = ~(op1 | op2); // bit-wise NOR
        `ALU_OPRN_WIDTH'h09 : result = op1 < op2 ? op1 : op2; // set less than
        default: result = `DATA_WIDTH'hxxxxxxxx;

    endcase
end
```

Fig. 8. Switch statement with different cases for alu module

B. *Implementing prj_01_tb*

To implement our test cases we will add a test case for each operation in prj_01_tb. Most of the code has been done for us already all we need to do is add in the rest by switching the numbers and operation code.

```
// test 15 + 3 = 18
#5  op1_reg=15;
    op2_reg=3;
    oprn_reg=`ALU_OPRN_WIDTH'h01;
#5  test_and_count(total_test, pass_test,
                   test_golden(op1_reg,op2_reg,oprn_reg,r_net));
// test 15 - 5 = 10
#5  op1_reg=15;
    op2_reg=5;
    oprn_reg=`ALU_OPRN_WIDTH'h02;
#5  test_and_count(total_test, pass_test,
                   test_golden(op1_reg,op2_reg,oprn_reg,r_net));
// test 15 * 2 = 30
#5  op1_reg=15;
    op2_reg=2;
    oprn_reg=`ALU_OPRN_WIDTH'h03;
#5  test_and_count(total_test, pass_test,
                   test_golden(op1_reg,op2_reg,oprn_reg,r_net));

// test 8 >> 2 = 2
#5  op1_reg=8;
    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));

// test 1 << 2 = 4
#5  op1_reg=1;
    op2_reg=2;
    oprn_reg=`ALU_OPRN_WIDTH'h05;
#5  test_and_count(total_test, pass_test,
                   test_golden(op1_reg,op2_reg,oprn_reg,r_net));

// test 10 & 7 = 2
#5  op1_reg=10;
    op2_reg=7;
    oprn_reg=`ALU_OPRN_WIDTH'h06;
#5  test_and_count(total_test, pass_test,
                   test_golden(op1_reg,op2_reg,oprn_reg,r_net));
// test 10 | 7 = 15
#5  op1_reg=10;
    op2_reg=7;
    oprn_reg=`ALU_OPRN_WIDTH'h07;
#5  test_and_count(total_test, pass_test,
                   test_golden(op1_reg,op2_reg,oprn_reg,r_net));

// test 10 ~| 7 = 4294967280
#5  op1_reg=10;
    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));

// test 100 slt 101 = 100
#5  op1_reg=100;
    op2_reg=101;
    oprn_reg=`ALU_OPRN_WIDTH'h09;
#5  test_and_count(total_test, pass_test,
                   test_golden(op1_reg,op2_reg,oprn_reg,r_net));
```

Fig. 9. The test cases for each operation in prj_01_tb

C. *Implementing test_golden*

In order to test our results and print them out, we need to use write statements in a switch statement. Most of the code has already been done for us at the bottom of prj_01_tb, all we need to do is put different cases for each operation.

```
case(oprn)
    `ALU_OPRN_WIDTH'h01 : begin $write("+ "); golden = op1 + op2; end
    `ALU_OPRN_WIDTH'h02 : 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("slt "); golden = op1 < op2 ? op1 : op2; end

    default: begin $write("? "); golden = `DATA_WIDTH'hx; end
endcase
```

Fig. 10. The switch statement responsible for print the different cases

Now you have completed implementing the ALU. Now what we need to do first is save the code. If we do not save the code and compile, it will not actually compile the edited files that you have made. We can save the files by clicking the icon on the top left or by pressing Ctrl + s. Once the files are saved go to the top left and click on Compile and then Compile All. If there were no errors, we are now ready to test and simulate the ALU.

V. Test Strategy and Implementation of ALU

To test and simulate the ALU, we will do the same procedure earlier in order to see the wave signals. Once you have the wave window open, you can use the scroll wheel at the bottom to see the different inputs and outputs of the program based on the amount of picoseconds after the program was simulated.

A. *No input – For the first 5000 ps, there are no inputs, hence there is not output which is why the r_net value is x. An x value indicants an uknown value.*

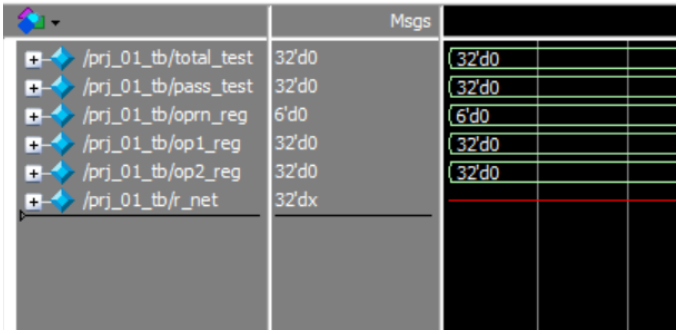

Fig. 11. The wave signals of the ALU simulation with no input

B. *Addition – next we have the inputs for addition, ALU takes 15 and 3 and operation code 0x1 so it gives an addition output of 18.*

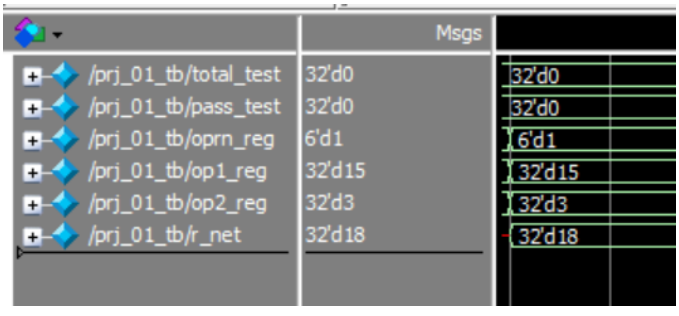

Fig. 12. The wave signals of the ALU simulation with addtion

C. *Subtraction – next we have the inputs for subtraction, ALU takes 15 and 5 and operation code 0x2 so it gives an subtraction output of 10.*

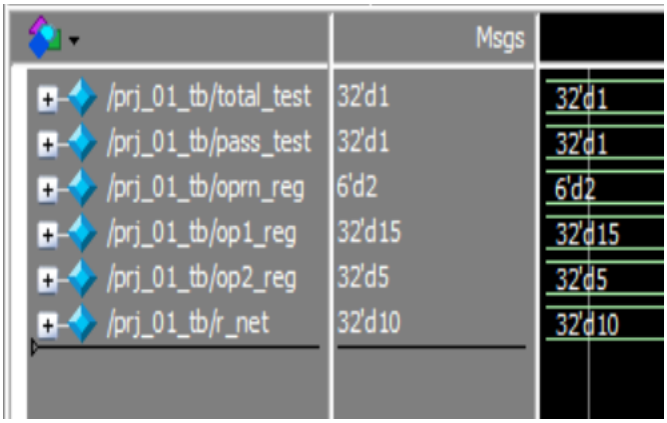

Fig. 13. The wave signals of the ALU simulation with subtraction

D. *Multiplication – next we have the inputs for multiplication, ALU takes 15 and 2 and operation code 0x3 so it gives an multiplication output of 30.*

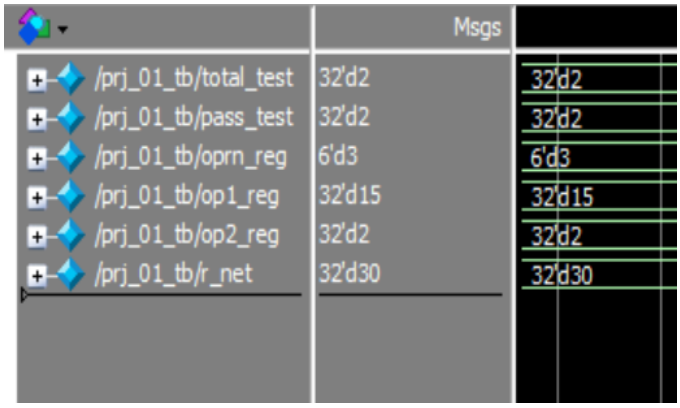

Fig. 14. The wave signals of the ALU simulation with multiplication

E. *Right Bit Shift – next we have the inputs for right shift, ALU takes 8 (1000) and 2 and operation code 0x4 so it shifts 8 (1000) to the right by 2 and gives 2 (0010).*

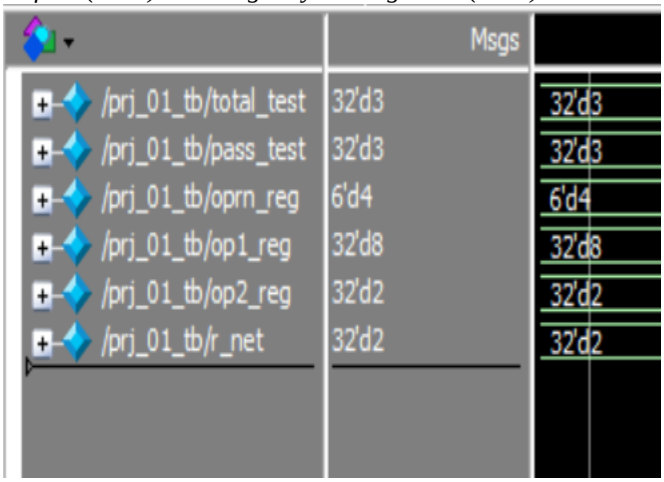

Fig. 15. The wave signals of the ALU simulation with right shift

F. *Left Bit Shift – next we have the inputs for left shift, ALU takes 1 (0001) and 2 and operation code 0x5 so it shifts 1 (0001) to the left by 2 and gives 4 (0100).*

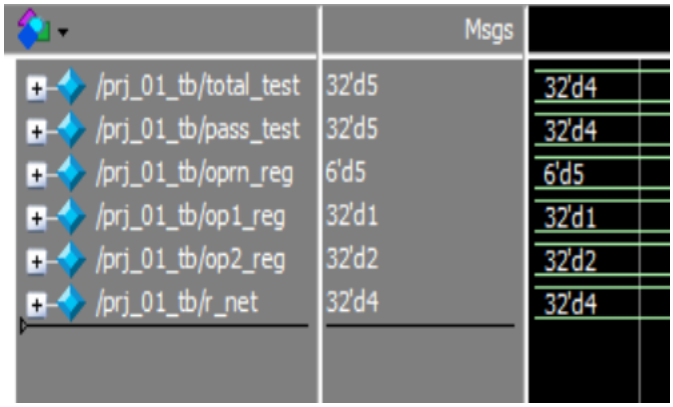

Fig. 16. The wave signals of the ALU simulation with left shift

G. *Bitwise AND – next we have the inputs for AND, ALU takes 10 (1010) and 7 (0111) and operation code 0x6 so it does an AND operation between 10 and 7 and gives 2 (0010).*

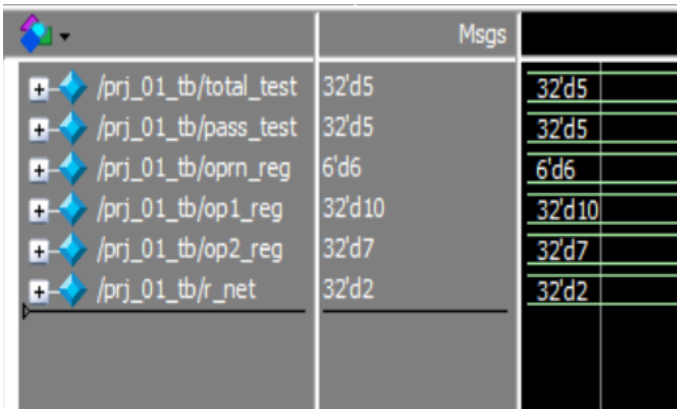

Fig. 17. The wave signals of the ALU simulation with bitwise AND

H. *Bitwise OR – next we have the inputs for OR, ALU takes 10 (1010) and 7 (0111) and operation code 0x7 so it does an OR operation between 10 and 7 and gives 15 (1111).*

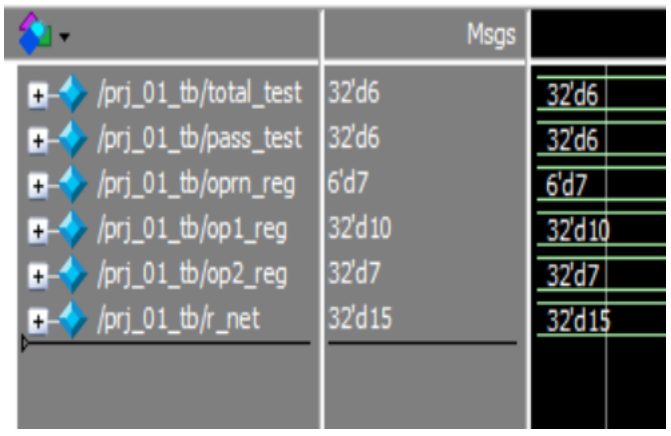

Fig. 18. The wave signals of the ALU simulation with bitwise OR

I. *Bitwise NOR – next we have the inputs for NOR, ALU takes 10 (1010) and 7 (0111) and operation code 0x8 so it does a NOR operation between 10 and 7 and gives 4294967280. Notice how there is a negative in front of the 32, this means that the number can either be interpreted as signed or unsigned. In this case, the number in the wave is signed so it is -16; however, in the transcript window, the number is unsigned so it gives the first number.*

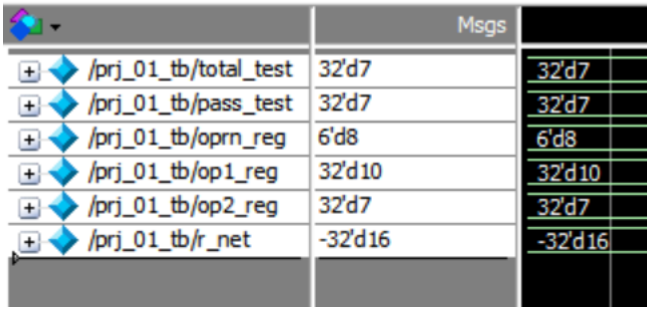

Fig. 19. The wave signals of the ALU simulation with bitwise NOR

J. *Set Less Than – lastly, we have the inputs for slt, ALU takes 100 and 101 and operation code 0x9 so it returns the smaller number which is 100.*

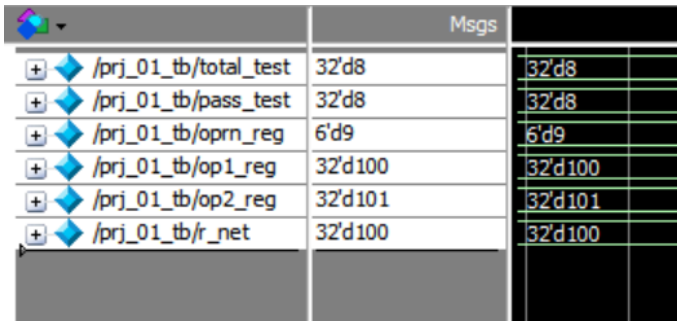

Fig. 20. The wave signals of the ALU simulation with set less than

K. *All operations – If we zoom out, we can show all of the operations at once. Notice how in between operations, the program checks if the test cases pass or fail and increments accordingly. The whole simulation lasted 95000 ps.*

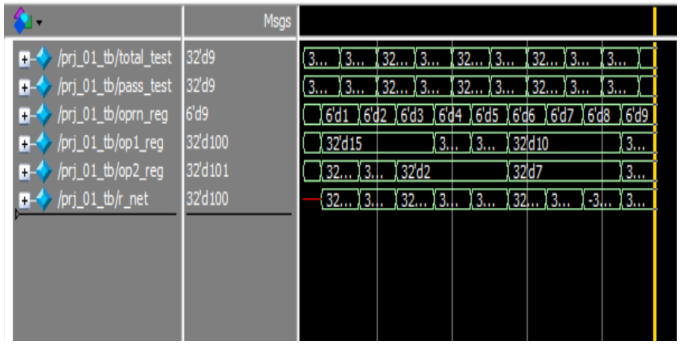

Fig. 21. The wave signals of the ALU simulation with all operations

L. *Transcript – Since the program was written to also write statements, the outputs are shown on a trasnript window on the bottom. You can either scroll up to see the outputs or dock the transcript window on a separate window.*

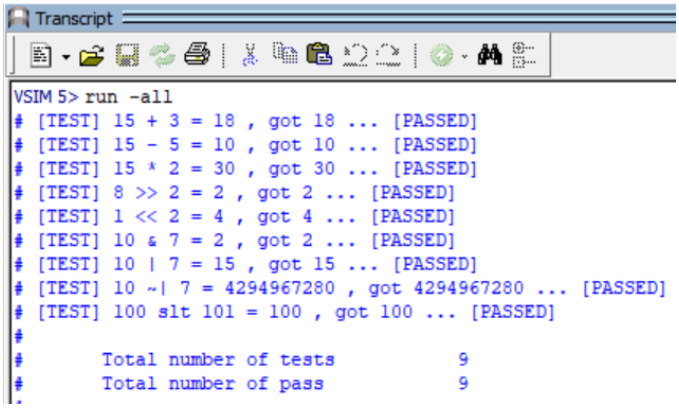

Fig. 22. The transcript of the ALU simulation

VI. CONCLUSION

This project has taught me how to download and install ModelSim and also a little bit of coding with Verilog language. I now have a better insight as to how the ALU works. I also learned a lot from analyzing the wave signals. I now understand how the computer takes a program and spends a certain amount of time processing each operation. Even though operations can be very fast, they are not instant. It was also very frustrating to deal with a different language as well as a different IDE/simulator. One thing that I was stuck on for a very long time was that after editing a code, you must save it in order to compile it. I was trying to run my code but all of the edits I made did not seem to really matter. Now that I know this, I can write programs much faster.