ModelSim Installation and Implementing an Arithmetic Logic Unit (ALU) in Verilog using ModelSim

Abdurrahman Mohammad

San Jose State University Computer Science Department

San Jose State University

Santa Clara, United States

abdurrahman.mohammad@sjsu.edu

***Abstract*—This electronic report demonstrates the installation of ModelSim on Windows 10 as well as an implementation of an Arithmetic Logic Unit (ALU) in the Verilog language using ModelSim. This report further demonstrates:**

1. The installation of ModelSim on a Windows 10 machine
2. Construction and implementation of an ALU in Verilog using the application ModelSim.
3. Executing the test bench of this ALU in Verilog with ModelSim.
4. Simulation of the ALU and observation of its waveform output using ALU test bench.

General Information

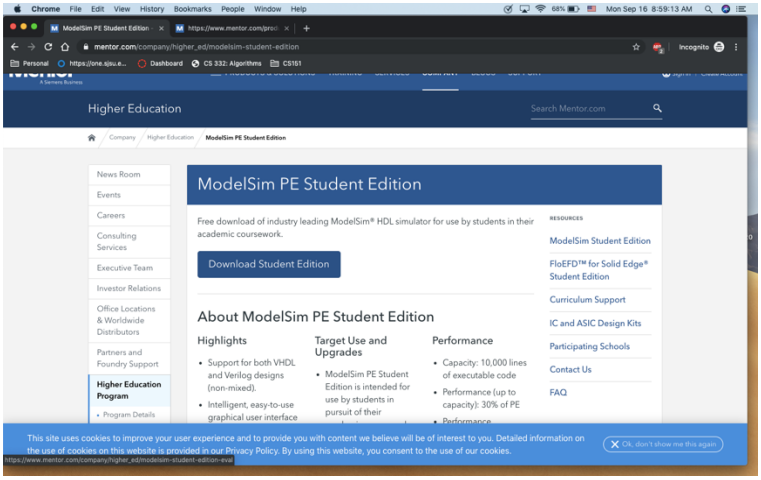
Tools Used:

1. ModelSim by Mentor Graphics
2. Windows 10 by Microsoft

1. ModelSim Installation on Windows 10

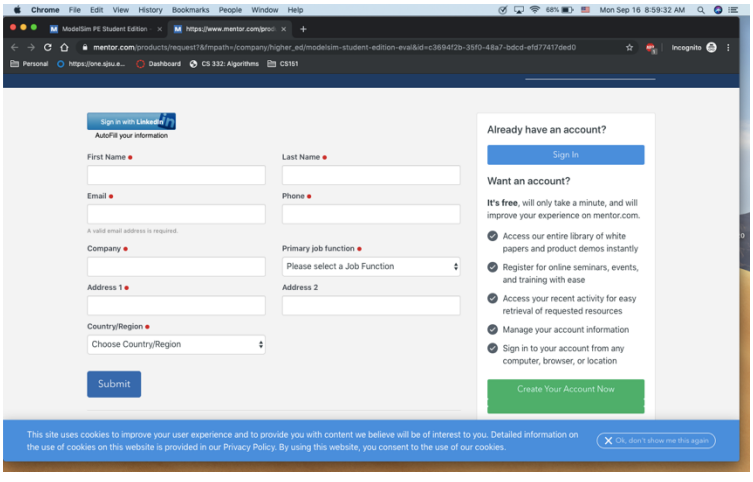
ModelSim is an application that allows users to simulate hardware without physically testing out that hardware in real life. It implements a Hardware Description Language (HDL) called Verilog which is very useful for constructing, implementing, then testing a virtual hardware component such as an ALU. The following are instruction on how to obtain and install ModelSim. As of September 2019, ModelSim has a free student edition.

1. Go to



"https://www.mentor.com/company/higher_ed/modelsim-student-edition"

2. Click "Download Student Edition" which is https://www.mentor.com/products/request?&fmpath=/company/higher_ed/modelsim-student-edition-eval&id=c3694f2b-35f0-48a7-bdcd-efd77417ded0



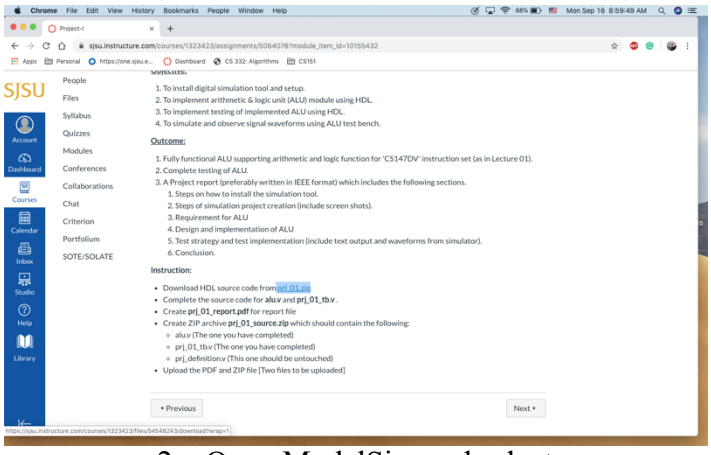
3. Fill out your information and click submit
4. Your browser should then automatically start downloading the application
5. Run the installer and follow the options on screen
6. The first time you open ModelSim, you will be directed to a web form which asks you to input your information for a free license. Do so.
7. You will shortly receive an email with a license. Download it.
8. Visit the directory where you installed ModelSim (such as: c:/modeltech_pe_edu). Find the folder win32pe_edu and drop the student license file into that folder
9. Restart ModelSim and you should receive no errors if this process is completed correctly

2. Construction and implementation of an ALU in Verilog ModelSim

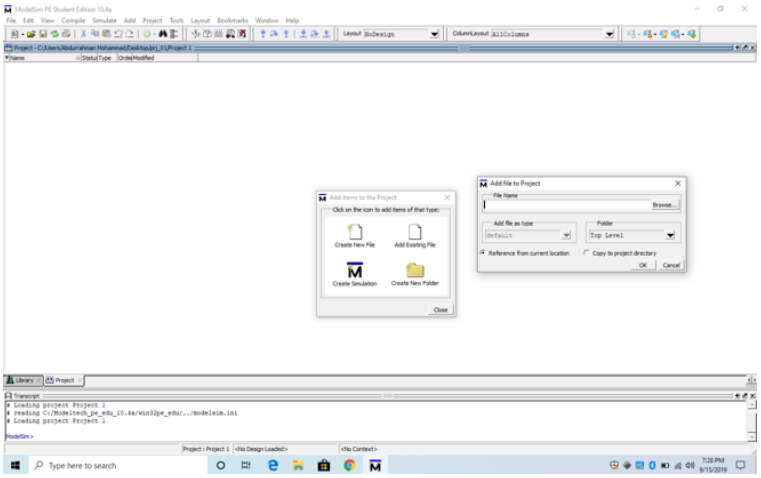
Below are the steps on how to create a project, compile it, and simulate it using

ModelSim. Instructions are specified for Patra's CS 147 class.

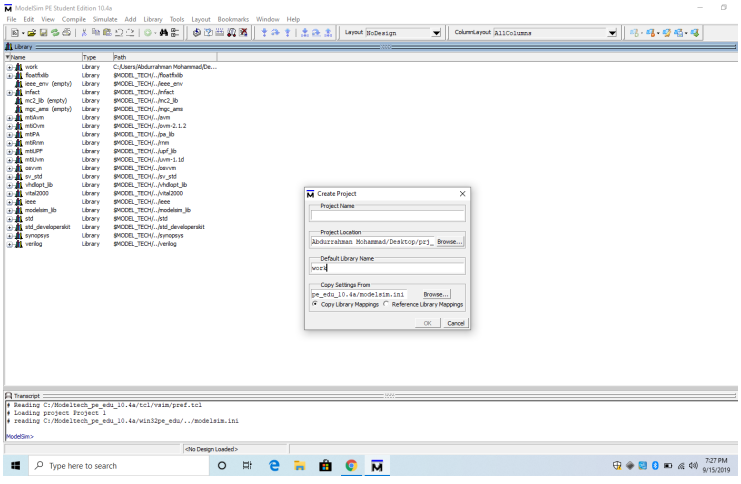
1. Download the "prj_01.zip" zip from canvas and extract it.



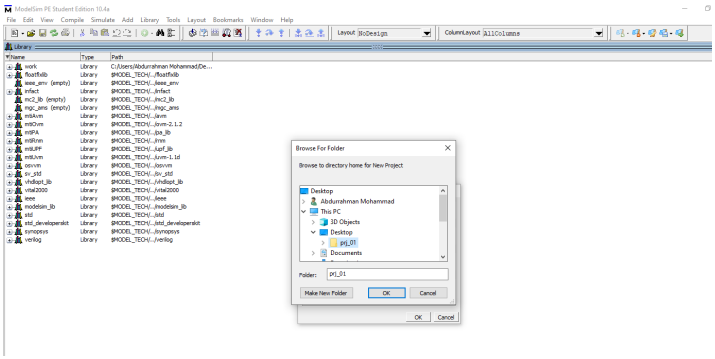
2. Open ModelSim and select File→New→Project



3. Name your project and specify directory

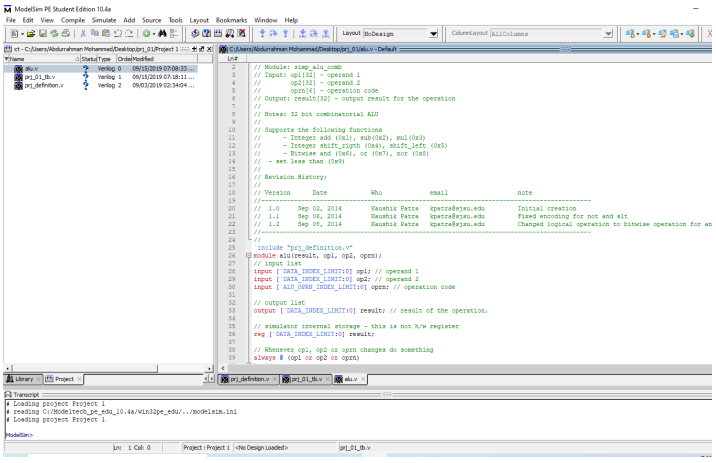


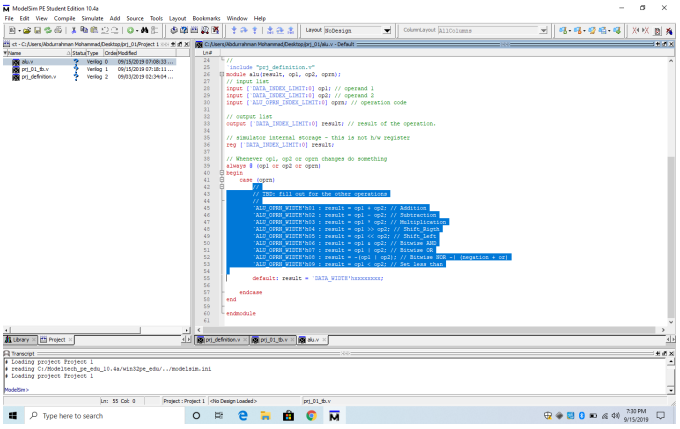
4. A window will pop up asking to create or import files. Select "Add Existing File" and navigate to the directory where you extracted "prj_01.zip". Select all the three files and click "Open". The files should now appear on your ModelSim screen

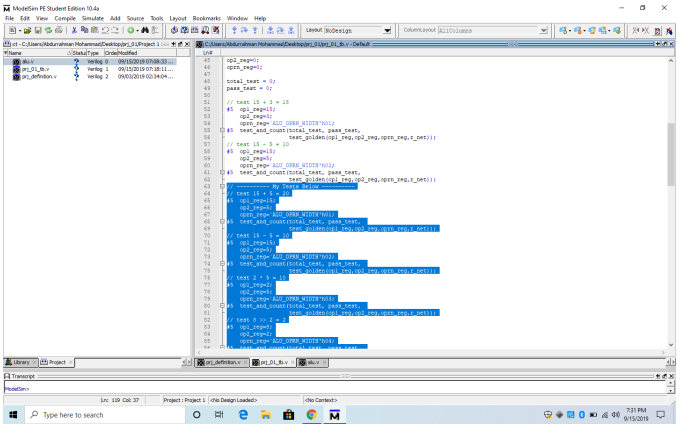


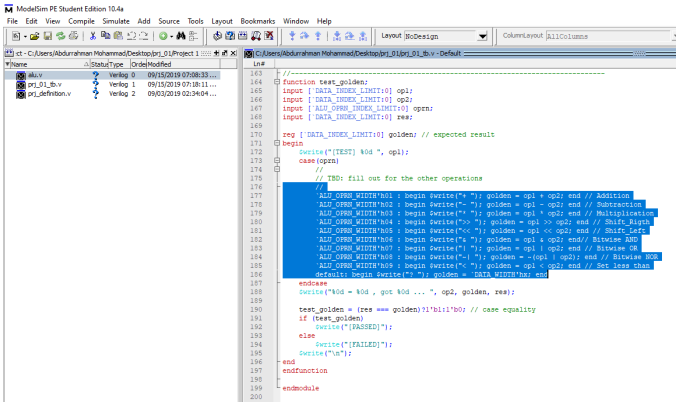
5. Double click the files to edit.

Here are my edits:

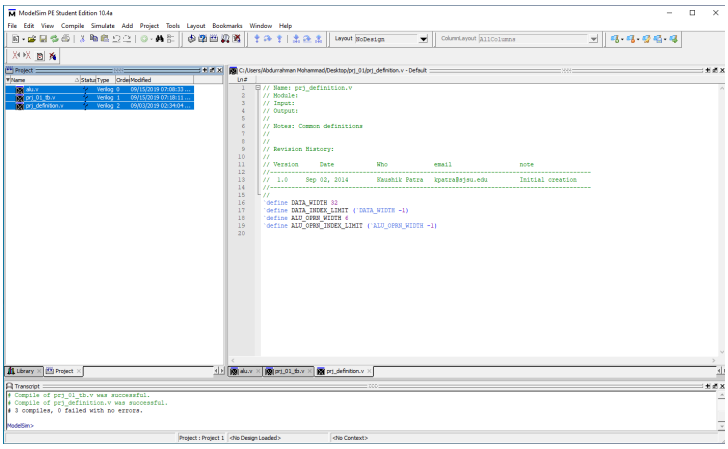


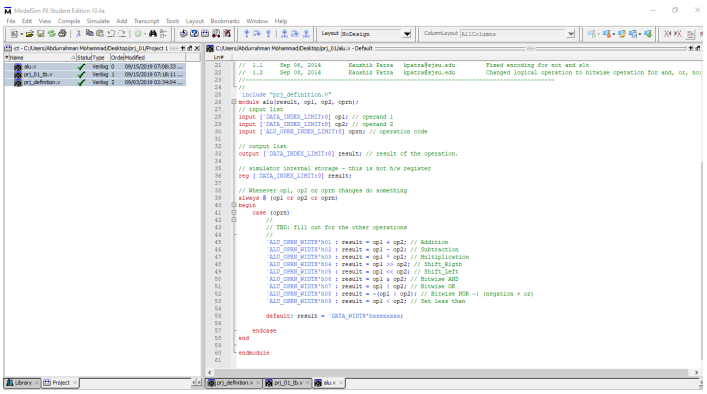




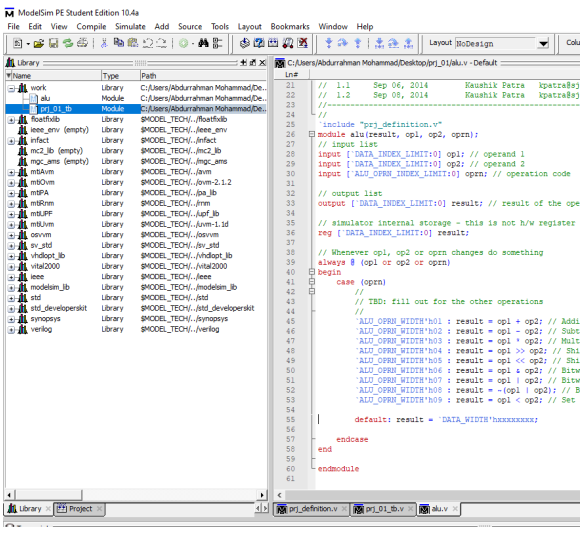


6. After editing, click “Compile all” located in the menu bar under the File, Edit, etc. bar. A checkmark should next to the compiled files and the console should display a success message. If an “X” appears next to your file, there likely is an error in your file and you should fix the error and re-edit the file.

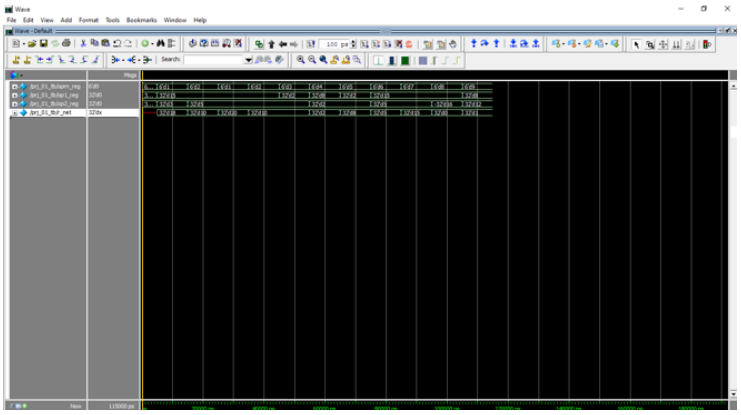




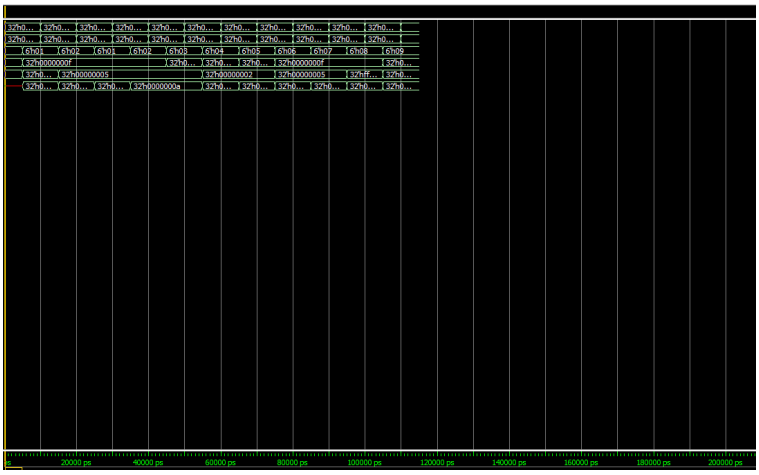
7. After compiling, navigate to the library tab. Select work and double click “prj_01_tb” which is the test bench. Do not click on the ALU, it will not simulate.



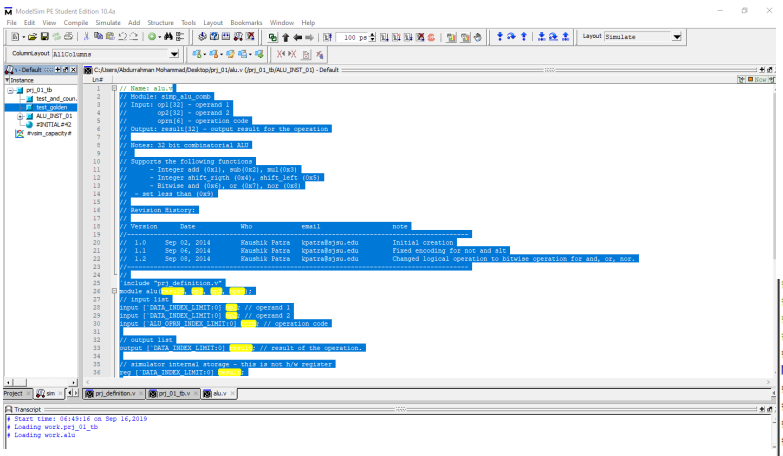
8. A “sim” window will pop up and there will be blue boxes and spheres next to your files. An “object” window may or may not pop up. Right click “prj_01_tb” and select “Add Wave”. A new black and green graph window should pop up. Right click on the objects in the table to the left of the graph and click “Radix” and change the units to “Decimal” or whatever you wish.

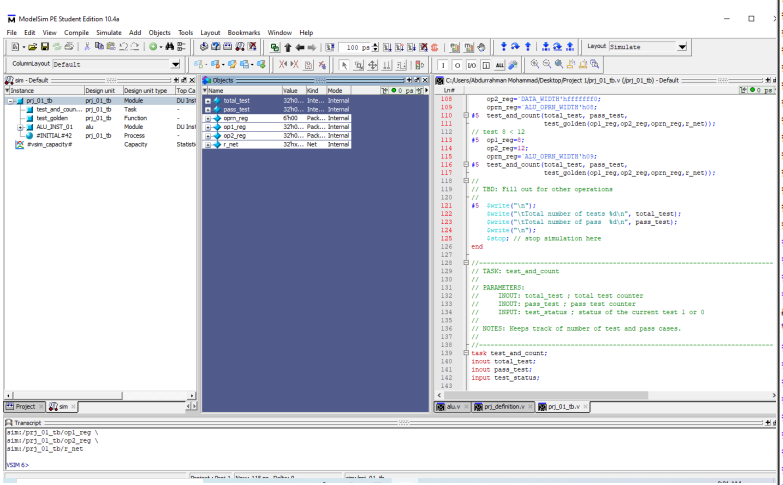


9. Press the blue magnifying glass in the menu bar above the graph and you should obtain your expected graph



Here is the output of the console:

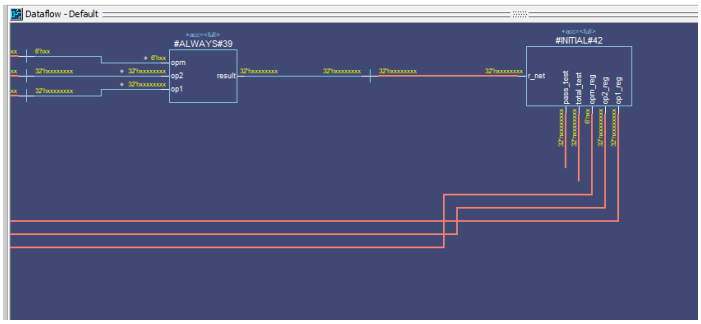




```
# Compile of prj_01_tb.v was successful.
# Compile of prj_definition.v was successful.
# Compile of alu.v was successful.
# Compile of prj_01_tb.v was successful.
# 3 compiles, 0 failed with no errors.
ModelSim> vsim work.prj_01_tb
# //  ModelSim PE Student Edition 10.4a Apr  7 2015
# //
# //  Copyright 1991-2015 Mentor Graphics Corporation
# //  All Rights Reserved.
# //
# //  THIS WORK CONTAINS TRADE SECRET AND PROPRIETARY INFORMATION
# //  WHICH IS THE PROPERTY OF MENTOR GRAPHICS CORPORATION OR ITS
# //  LICENSORS AND IS SUBJECT TO LICENSE TERMS.
# //  THIS DOCUMENT CONTAINS TRADE SECRETS AND COMMERCIAL OR FINANCIAL
# //  INFORMATION THAT ARE PRIVILEGED, CONFIDENTIAL, AND EXEMPT FROM
# //  DISCLOSURE UNDER THE FREEDOM OF INFORMATION ACT, 5 U.S.C. SECTION 552.
# //  FURTHERMORE, THIS INFORMATION IS PROHIBITED FROM DISCLOSURE UNDER
# //  THE TRADE SECRETS ACT, 18 U.S.C. SECTION 1905.
# //
# // NOT FOR CORPORATE OR PRODUCTION USE.
# // THE ModelSim PE Student Edition IS NOT A SUPPORTED PRODUCT.
# // FOR HIGHER EDUCATION PURPOSES ONLY
# //
# vsim -gui
# Start time: 07:48:31 on Sep 16,2019
# Loading work.prj_01_tb
# Loading work.alu
add wave -position insertpoint sim:/prj_01_tb/*
VSIM 3> run -all
# [TEST] 15 + 3 = 18 , got 18 ... [PASSED]
# [TEST] 15 - 5 = 10 , got 10 ... [PASSED]
# [TEST] 15 + 5 = 20 , got 20 ... [PASSED]
# [TEST] 15 - 5 = 10 , got 10 ... [PASSED]
# [TEST] 2 * 5 = 10 , got 10 ... [PASSED]
# [TEST] 8 >> 2 = 2 , got 2 ... [PASSED]
# [TEST] 2 << 2 = 8 , got 8 ... [PASSED]
# [TEST] 15 & 5 = 5 , got 5 ... [PASSED]
# [TEST] 15 | 5 = 15 , got 15 ... [PASSED]
# [TEST] 15 ~| 4294967280 = 0 , got 0 ... [PASSED]
# [TEST] 8 < 12 = 1 , got 1 ... [PASSED]
#
#       Total number of tests          11
#       Total number of pass           11
#
# ** Note: $stop    : C:/Users/Abdurrahman Mohammad/Desktop/Project 1/prj_01_tb.v(125)
#    Time: 115 ns  Iteration: 0  Instance: /prj_01_tb
# Break in Module prj_01_tb at C:/Users/Abdurrahman Mohammad/Desktop/Project 1/prj_01_tb.v line 125

VSIM 4>
```

Here is the data flow model:

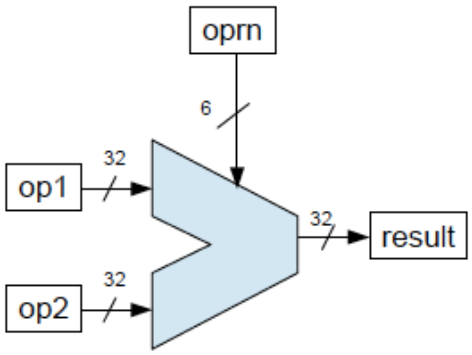


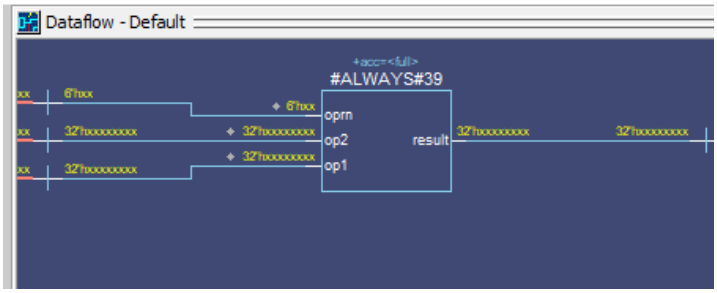
10. After viewing, you may close the application

3. ALU Requirements

The Arithmetic Logic Unit (ALU) is responsible for computing the arithmetic and logical operations for the computer. It has the ability to implement arithmetic such as addition, subtraction, and multiplication. It can perform bitwise operations such as “and”, “or”, and “nor”. In this program, we are simulating our own ALU. In our program, we simulate registers “op1” and “op2” which resemble 32-bit registers. We also define our operations and refer to them with their appropriate operation number which is the 6-bit “oprn” which will be passed by the Control Unit (CU). Our expressions corresponding to “oprn”, or the function corresponding to the “opcode” as stated in class, passed by the CU are evaluated on “op1” and “op2”. The ALU then spits out a 32-bit “result” which is saved in memory.

Following is an ALU diagram and our data flow model. Notice the striking resemblance of what a typical ALU flow looks like and what we have simulated.





4. Design and Implementation of ALU

ModelSim helps simulate our ALU design effectively and accurately, saving us time and money. Our task in ModelSim is to specify the dataflow using Verilog. We do so by defining defining an ALU module in ModelSim as such:

```
`include "prj_definition.v"
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.

// simulator internal storage - this is not h/w register
reg [`DATA_INDEX_LIMIT:0] result;

// Whenever op1, op2 or oprn changes do something
always @ (op1 or op2 or oprn)
begin
    case (oprn)
        //
        // TBD: fill out for the other operations
        //
        `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_Rigth
        `ALU_OPRN_WIDTH'h05 : result = (op1 << op2); // Shift_Left
        `ALU_OPRN_WIDTH'h06 : result = (op1 & op2); // Bitwise AND
        `ALU_OPRN_WIDTH'h07 : result = (op1 | op2); // Bitwise OR
        `ALU_OPRN_WIDTH'h08 : result = ~(op1 | op2); // Bitwise NOR ~| (negation + or)
        `ALU_OPRN_WIDTH'h09 : result = (op1 < op2); // Set less than

        default: result = `DATA_WIDTH'hxxxxxxxx;

    endcase
end

endmodule
```

- module alu(result, op1, op2, oprn); - describes what goes into the ALU which are: a “result” storage/register, “op1” register, “op2” register, and an operation number “opn” which specifies the operation the ALU should do.
- Underneath that is a deifnitoin of inputs which include the size (32-bit specified by constant `DATA_INDEX_LIMIT:0 which means from index 0 to 31) of the registers “op1” and “op2”. It also includes the size of the operatoin number “oprn” which is 6 bits specified by constant `ALU_OPRN_INDEX_LIMIT:0 which means form index 0 to 5.
- Then the output is defined as a 32-bit resgister referenced as “result”
- Underneath that is another “result” but this one simulates internal storage. It is also 32 bit specified by constant `ALU_OPRN_WIDTH.
- Underneath that is a list of 9 operations that the ALU can perform: adition, subtraction, multiplication, integer shift right, integer shift left, bitwise and, bitwise or, bitwise nor, and set less than. Their “opn” or operation code is defined from 01 to 09. The operation code appears after the ‘h. It is 6 bit and is in hexadecimal.

Operation Number	Operation	Equation
01	Addition	result = op1 + op2
02	Subtraction	result = op1 - op2
03	Multiplication	result = op1 * op2
04	Integer shift right	result = op1 >> op2
05	Integer shif left	result = op1 << op2
06	Bitwise AND	result = op1 & op2
07	Bitwise OR	result = op1 | op2
08	Bitwise NOR	result = ~(op1 | op2)
09	Set less than	result = op1 < op2

Above is the definition of our ALU which is incorporated in the ALU module in ModelSim.

After designing our ALU, we will test it out and check if it wroks as we intended. We pass in the 2 operand registers and the operaction code and check its result.

5. Test Strategy and Implementation

To test our ALU, we will define functions in ModelSim that will input values for op1, op2, and opn into the ALU. The ALU will spit back a result which the function in ModelSim will then verify with its own calculations. This will be known as our "test bench." Our functino would say "PASSED" for each operation done correctly by the ALU.

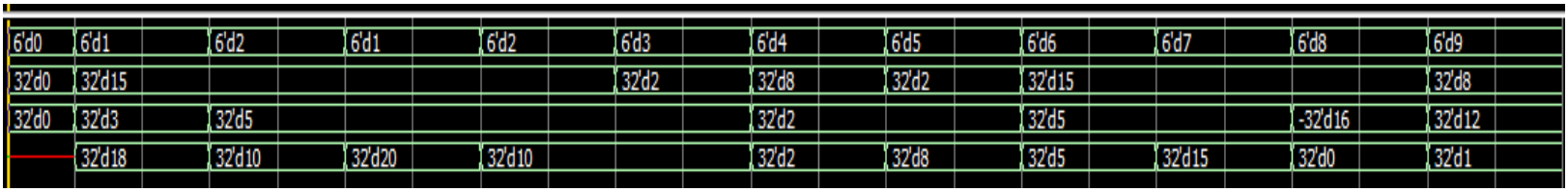
I have added a test case for each operation defined for our ALU by the "opn". The test for each operation is in order of the table mentioned above. You can also verify the operation by its "opn" or operation number mentioned in the 3rd line of each test (it is the number X in `ALU_OPRN_WIDTH'h0**X**).

I have also attached the output from the console verifying the results from the ALU.

```
#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));
// ---------- My Tests Below ----------
// test 15 + 5 = 20
#5  op1_reg=15;
    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));
// 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 2 * 5 = 10
#5  op1_reg=2;
    op2_reg=5;
    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 2 << 2 = 8
#5  op1_reg=2;
    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 15 & 5
#5  op1_reg=15;
    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));
// test 15 | 5
#5  op1_reg=15;
    op2_reg=5;
    oprn_reg=`ALU_OPRN_WIDTH'h07;
#5  test_and_count(total_test, pass_test,
                   test_golden(op1_reg,op2_reg,oprn_reg,r_net));

#5  op1_reg=15;
    op2_reg=`DATA_WIDTH'hfffffff0;
    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 8 < 12
#5  op1_reg=8;
    op2_reg=12;
    oprn_reg=`ALU_OPRN_WIDTH'h09;
#5  test_and_count(total_test, pass_test,
                   test_golden(op1_reg,op2_reg,oprn_reg,r_net));
```

```
add wave -position insertpoint sim:/prj_01_tb/*
VSIM 3> run -all
# [TEST] 15 + 3 = 18 , got 18 ... [PASSED]
# [TEST] 15 - 5 = 10 , got 10 ... [PASSED]
# [TEST] 15 + 5 = 20 , got 20 ... [PASSED]
# [TEST] 15 - 5 = 10 , got 10 ... [PASSED]
# [TEST] 2 * 5 = 10 , got 10 ... [PASSED]
# [TEST] 8 >> 2 = 2 , got 2 ... [PASSED]
# [TEST] 2 << 2 = 8 , got 8 ... [PASSED]
# [TEST] 15 & 5 = 5 , got 5 ... [PASSED]
# [TEST] 15 | 5 = 15 , got 15 ... [PASSED]
# [TEST] 15 ~| 4294967280 = 0 , got 0 ... [PASSED]
# [TEST] 8 < 12 = 1 , got 1 ... [PASSED]
#
#       Total number of tests          11
#       Total number of pass           11
#
# ** Note: $stop    : C:/Users/Abdurrahman Mohamma
#    Time: 115 ns  Iteration: 0  Instance: /prj_01
```



Attached is the wave output:

The first row is the “opn” operation code. The second row is the value of the “op1” register. The third row is the “op2” register. The fourth row is the “result” value. All the values are in decinal, as specified by ‘d. The number before that is the number of bits. After every ‘:’, the value changes. To read a test, look at each column and match its values to the corresponding values and operatoins. For example: 6’d1, 32’d15, 32’d3, 32’d18 corresponds to 15 + 3 = 18. A simplified summary of the tests is located in the console output above the wave output.

6. Conclusion

In this project, I installed ModelSim successfully and learned how to use it. I was able to create a project and load existing files. I also learned how to simulate an ALU in Verilog and how to define operations for that ALU. I also learned how to test my ALU and how to read the output in waveform.

The ALU met the specificatoins mentioned in the comments above by Professor Patra. I completed the definitions of the ALU operations. I also tested every one of the defined operations. The program works as intended. The output on the console was consisten with the output in waveform. All tests were passed and the ALU works as designed and intended.