Arithmetic & Logic Unit (ALU)

NADA EL ZEINI
San Jose State University
nada.elzeini@sjsu.edu

***Abstract*— This report's main objective is to implement a 32-bit Arithmetic & Logic Unit (ALU) using ModelSim, a digital simulation tool. Related goals will also be accomplished through this work.**

1) Installing a digital simulation tool and setup: ModelSim.

2) Implementing ALU module using the Verilog Hardware Description Language (HDL) as well as ALU's essentials.

3) Testing strategy and implementation using HDL of the ALU.

4) Simulation of waveforms and analytical observation through ALU test bench.

I. STEPS TO INSTALL THE SIMULATION TOOL

Generally, a simulator's role is to virtualize real life operations. In terms of computer architecture , it is basically a computer program that reproduces the behavior of a certain system. The usage of a digital simulator is beneficial as it allows a virtual show of operations of digital circuits. An ideal simulator application program for this project would be ModelSim from MentorGraphics. This tool is available for free for students only and can solely be used in a Windows operating system (for other operating systems, using a Virtual Machine to access Windows is a good option). Other tools that have simulation features can be relatively used such as ISE Web pack from Xillinx, Quartus from Altera, Active-HDL from Aldec. This project's examples and verifications will be done using ModelSim. The following steps shows how to install ModelSim Student Edition for no charge:

1) Access the following link in your web browser: http://www.mentor.com/company/higher_ed/modelsim-student-edition

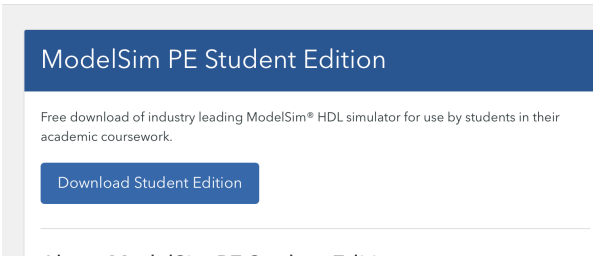

Figure 1.1:Download Student Edition

2) Click on the 'Download Student Edition' button (Figure 1.1).

3) MentorGraphics might ask you to create an account or sign in. Enter your information and submit. Either way, your installation file should be downloaded into your computer.

4) Open the folder in which the installation file was downloaded (it should be in Downloads by default), find 'modelsim-pe_student_edition' and double-click on it.

5) Run the installation file and complete the standard installation process.

6) After extracting the tool for you, at the end of the installation, a web browser will pop up with a form to fill in, in order to request a license (a necessary step). Fill out the form accurately and press on finish.

7) You will receive an email (to the one provided) from ModelSim with instructions and an attached file with the name of 'student_license.dat'. Save the file in somewhere different than the ModelSim file. Important note: the student license file should not be edited or modified in anyway or the it will fail.

8) Open the ModelSim file in a different window and find a folder named:'win32pe_edu'. Drag the student license file and drop it in win32pe_edu.

9) A shortcut for ModelSim will be created on the desktop. You will now be able to use ModelSim PE Student Edition.

II. STEPS OF SIMULATION PROJECT CREATION

This section will consist of setting up the required tools to achieve a behavioral model of ALU. The v files will help setting the Verilog code and further steps will help with the testing section related to waveforms observation.

1) Download the zip file 'prj_01.zip' attached in the project's assignment and extract it into a new folder. The file should consist of 3 Verilog files (indentifiable with extension .v). Each of the 3 file has a specification:

- 'alu.v' is basically theALU module code.
- 'prj_01_tb.v ' is the test bench file that will be necessary to be able to test the functionality of the ALU with different operations.
- 'prj_definition' consists of definitions and statements for the creation of the ALU (This file should remain untouched).

2) Open ModelSim, create a new project as in Figure 2.1.

- A 'Create Project' window will show up as in Figure 2.2.
- Give the project a name; for instance 'Project1_ALU', then click the OK button.

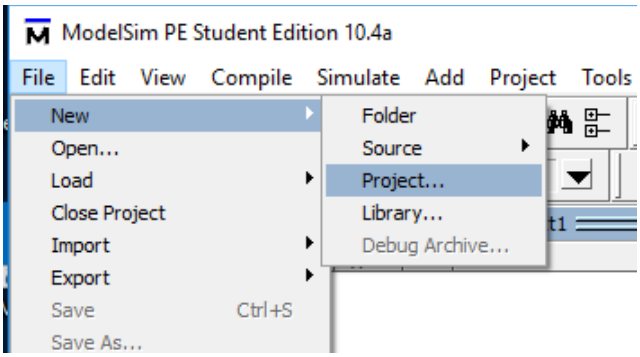

Figure 2.1: Creating a project

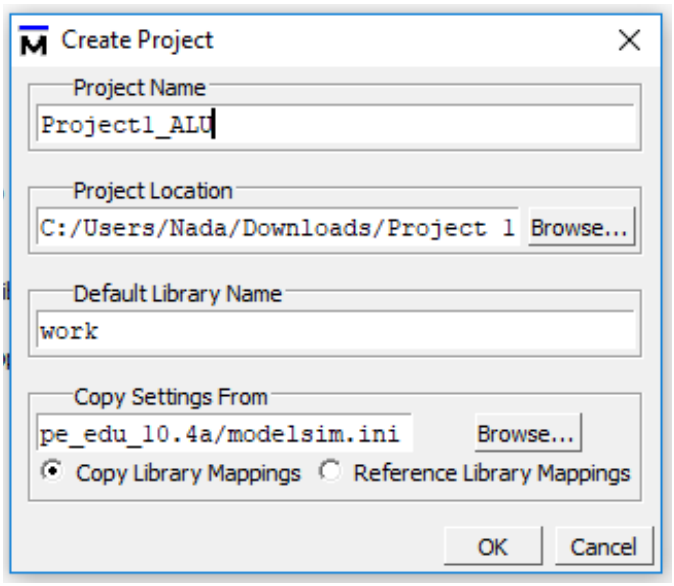

Figure 2.2: Create Project window

3) A window will show after the last step as shown in Figure 2.3. Click on 'Add Existing File'. Click on 'Browse' (Figure 2.4) then go to the folder where the file from step 1 was downloaded and select all 3 V files (Figure 2.5).

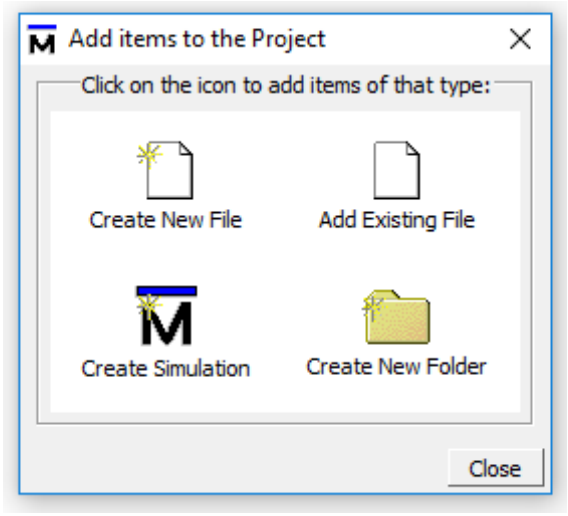

Figure 2.3: Add items to the project

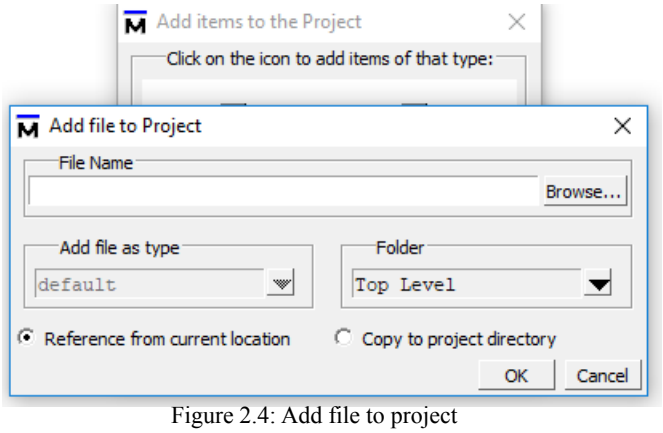

Figure 2.4: Add file to project

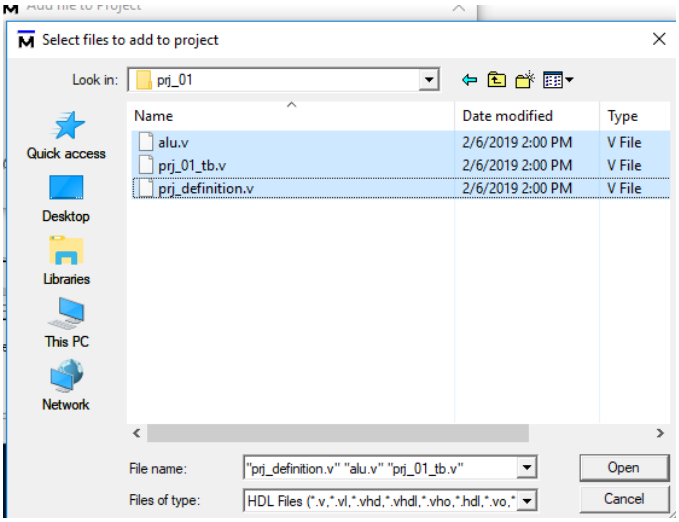

Figure 2.5: Select files to add to project

4) All 3 files are now added to the project and can be visible when in the 'project' tab. Click on the project tab)situated next the to 'library' tab, then click on the 'compile all' button (Figure 2.6) to make sure the files and the tool are working properly. A successful compilation will result in a green 'check' mark on each file as shown in Figure 2.7.

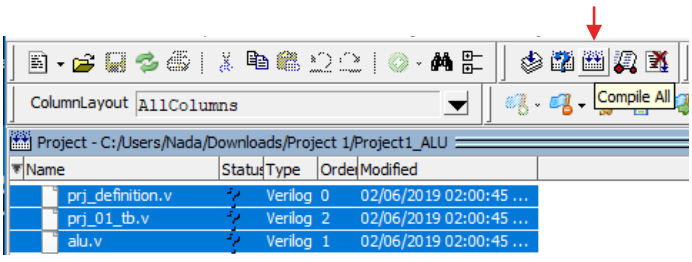

Figure 2.6: Compile All

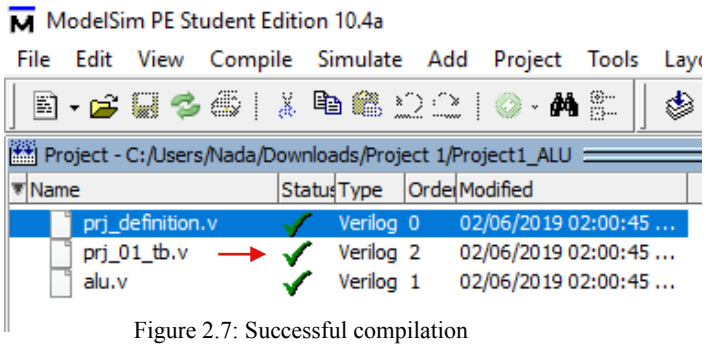

Figure 2.7: Successful compilation

5) This step is done in the 'library' tab. Access it, then double-click on a folder called 'work', and double-click on the test bench file that appears under work:'proj_01tb' (Figure 2.8).

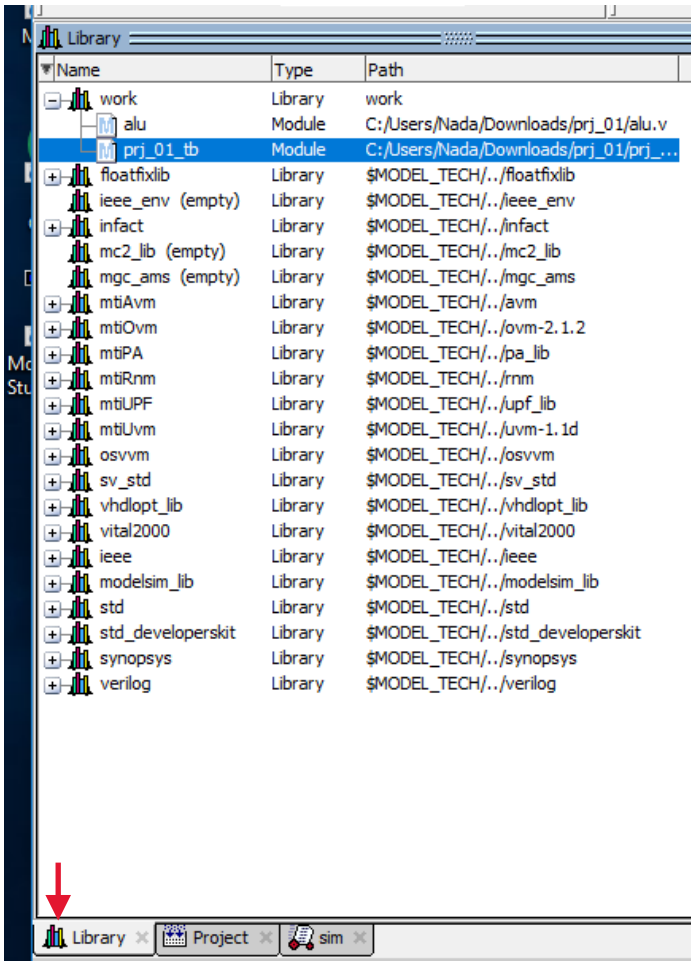

Figure 2.8: Running Test Bench Code

6) Now that you double-clicked on the test bench file, a new 'sim' tab will appear next to the project tab (Figure 2.9).

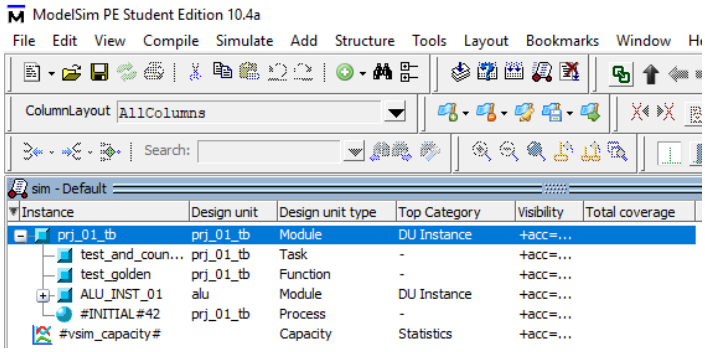

Figure 2.9: Sim tab

7) An 'Object' window should appear after step 6, as the one shown in figure 2.10. It will show ALU modules input, output and the test's name. Select all of them (using shift or manually) and right click to choose 'Add Wave' or simply press on Ctrl+W. A new black window will show up on the left side such as in figure 2.11.

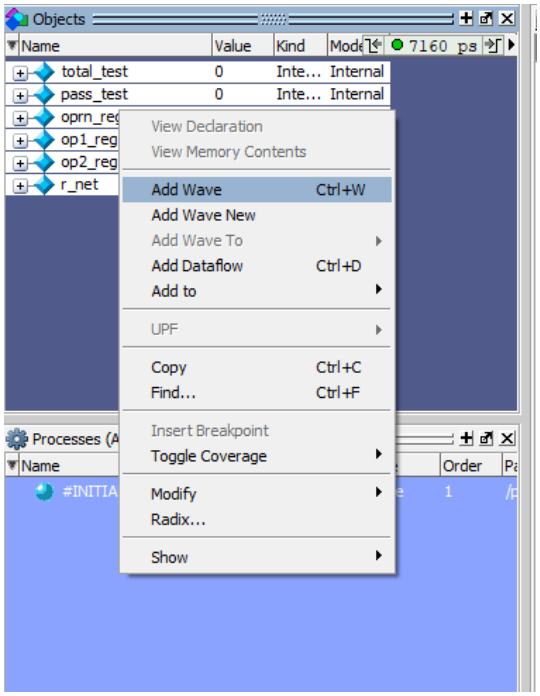

Figure 2.10: Object Window and adding wave

8) By default, the values on the wave will be in hexadecimal form to change that select all values as shown in figure 2.11 and click on 'Radix'. You should be able to choose the format you prefer. For this report, the format that will show most clearly the operation being executed will be selected.

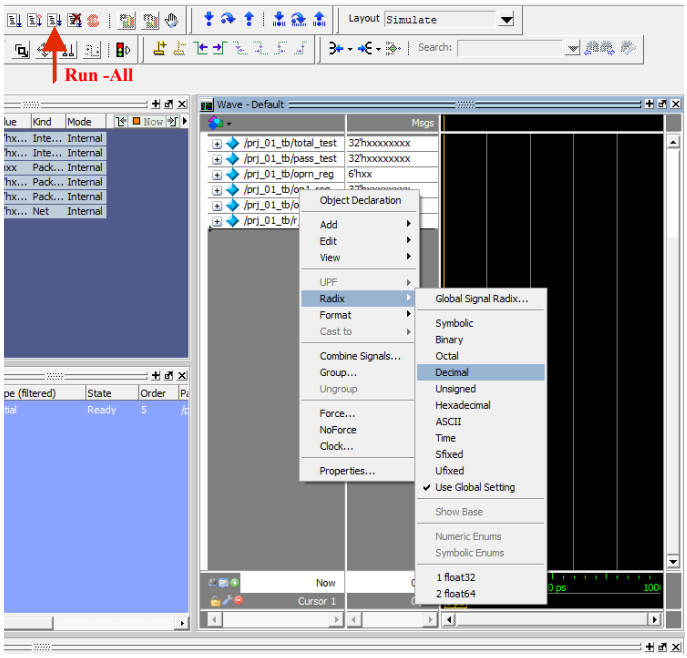

Figure 2.11:Waveform Window and changing radix

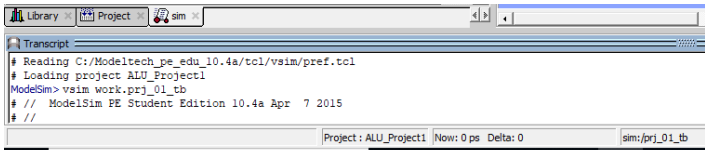

Figure 2.12: Transcript Window (partial view)

9) In this final step, you should use the 'Run all' button to see the output in the wave window (Figure 2.11).

- Also, the output will appear in text format in a 'Transcript' window (Figure 2.12) (output shown later in the report).

- Details of testing strategy and test implementation will be further reviewed in section V.

III. REQUIREMENTS OF ALU

An Arithmetic & Logic Unit (ALU) is the mathematical brain of the computer. It is a digital circuit used by the computer to handle arithmetic (addition, subtraction, multiplication…) and logic operations (like AND and OR). It is an essential building block of a computer processor. The ALU provides the fundamental functionality of a computer. Basically, any mathematical and logical program is broken down into an operation of two operands. For instance, this operation:' f = (g + h - i * j)' is broken down into this series of operations by the compiler:

- T1 = i*j
- T2 = h - T1
- f = g + T2

In terms of computer architectural objects, the ALU takes on a particular shape. The interface diagram of ALU is shown in figure 3.1. The schematic regards a operation performed.

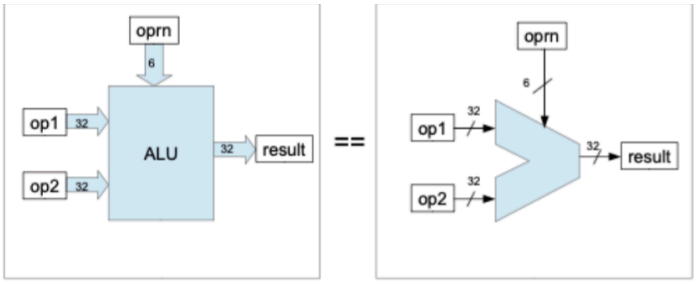

Figure 3.1: Interface Diagram for ALU

- 'op1' & 'op2' are the operands of 32-bits each, the arrow direction of data flow for these indicates that they are inputs. The single strike line specifies a multiple-bits operation contrary to a plain line which represents a one bit operation.
- The 'result' is basically the output (arrow going out of the ALU), also of 32-bits.
- 'oprn' represents the operation as code or instruction from the Control Unit (CU). Known as the Op-code, it is of 6-bits.

In summary, the ALU applies the operation/ instruction from fetched op-code to the input (op1 & op2), performs it and provides the output data in 'result' (uses registers for input/output).

IV. DESIGN AND IMPLEMENTATION OF ALU

This section will outline the usage of the Verilog Hardware Description Language to design and eventually implement the ALU. The process is simplistically about describing the functions of the ALU using Verilog and the simulation tool 'ModelSim'; set up earlier in the report, will be able to work on achieving the majority of this report 's goals regarding testing and simulation of signal waveforms; detailed in the next section.

A. ALU Module

Since the objective is to define a behavioral model of ALU, therefore the ALU, treated as a module, the structural definition will be the same and the module have the same basic requirements and functionality:

- 3 input elements, as seen in the file 'alu.v' added previously: op1, op2, and oprn (Figure 4.1). '[DATA_INDEX_LIMIT:0]' represents the data width (which is 32-bits as we are dealing with a 32-bit ALU) subtracted by 1 to indicate that the inputs are 32-bits each ([31:0]). Similarly, the operation input is defined with 6-bits using '[DATA_OPRN_INDEX_LIMIT:0]' as [5:0].

```
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 4.1: Declarations of ALU ports

- After these declarations will follow the set of instructions/operations that the ALU will handle (described in the following sub-section).
- After declarations of the operations, the module will end as it results the output (which will be referred to as 'golden').

B. ALU Operations

In this project, 9 operations will be set up (in the following order) to be performed by the ALU:

(1) Addition
(2) Subtraction
(3) Multiplication
(4) Shift Right
(5) Shift Left
(6) Bitwise AND
(7) Bitwise OR
(8) Bitwise NOR
(9) Set Less Than

These operations will be described and implemented in the 'alu.v' file. Each operation will begin with 'ALU_OPRN_WIDTH'h0#' where ALU_OPRN_WIDTH is already defined to be of 6-bits in 'prj_definition.v' (op-code width) and where # will be replaced by the number specified for that particular operation; in a way that when calling operation 7 for example, 'h07' will be used to refer to that operation (Bitwise OR corresponds to 7 as in the listed operations earlier). Note that when the operation, input value, or the opcode are not defined and unknown values of 'x' will appear indicating failure of operation. The declarations of the operation in the program will take the following form (Note the correct usage of specific operators for each operations) (Figure 4.2):

```
// 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; //substraction
        `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; //AND
        `ALU_OPRN_WIDTH'h07 : result = op1 | op2; //OR
        `ALU_OPRN_WIDTH'h08 : result = ~(op1 | op2); //NOR
        `ALU_OPRN_WIDTH'h09 : result = op1 < op2; //set less than

        default: result = `DATA_WIDTH'hxxxxxxxx;

    endcase
end
```

Figure 4.2: Declarations of operations of ALU

V. TEST STRATEGY AND TEST IMPLEMENTATION

Now that the Verilog code has been set for the operations of ALU, the functionality of that design and implementation should be tested. Basically, each operation put into effect will be tested to see whether or not it os being conducted correctly. Verilog HDL code can provide the user a repeatable set of stimuli or commands to test some functionality, this code also known as Simulation and Validation Code is the test bench which in this project is 'prj_01_tb.v'. The test bench will support the operations declared earlier (from addition to set less than). The figure 5.1 provides an accurate schematic of how the test bench will act with the ALU; in comparison to figure 3.1, the Stimulus and Validation are now the source and destination of input and output.

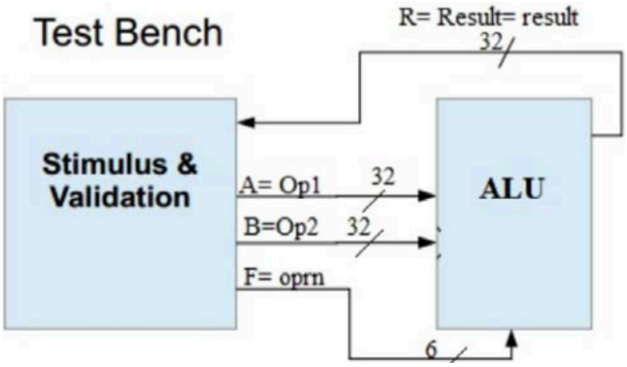

Figure 5.1: Diagram for simulation block

The code starts with instantiation of the ALU , basically with the same principles as in the module then driving the test and test pattern will begin (with 0 value for each of the operands and operation registers, 'total_test' and 'pass_test' are first declared as 0 and will increase as each operation is conducted successfully). As we fill out the code for multiple operations, they will be tested against the 'golden_test' function also coded with description of each of the 9 operations. You can choose to test each operations using different numbers and multiple times. In this project, 20 operations were executed; 2 for each kind of operation (Figure 5.2). An error line would appear if a mistake is made. If a operation doesn't pass, the test bench code will output [FAILED] by the operation in the transcript.

```
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] 0 - 0 = 0 , got 0 ... [PASSED]
# [TEST] 5 * 10 = 50 , got 50 ... [PASSED]
# [TEST] 9 * 4 = 36 , got 36 ... [PASSED]
# [TEST] 0 * 8 = 0 , got 0 ... [PASSED]
# [TEST] 8 >> 3 = 1 , got 1 ... [PASSED]
# [TEST] 1 >> 1 = 0 , got 0 ... [PASSED]
# [TEST] 8 << 3 = 64 , got 64 ... [PASSED]
# [TEST] 2 << 1 = 4 , got 4 ... [PASSED]
# [TEST] 15 & 15 = 15 , got 15 ... [PASSED]
# [TEST] 1 & 15 = 1 , got 1 ... [PASSED]
# [TEST] 0 | 0 = 0 , got 0 ... [PASSED]
# [TEST] 15 | 15 = 15 , got 15 ... [PASSED]
# [TEST] 1 | 5 = 5 , got 5 ... [PASSED]
# [TEST] 1 ~| 4 = 4294967290 , got 4294967290 ... [PASSED]
# [TEST] 2 ~| 0 = 4294967293 , got 4294967293 ... [PASSED]
# [TEST] 9 < 1 = 0 , got 0 ... [PASSED]
# [TEST] 6 < 7 = 1 , got 1 ... [PASSED]
#
#       Total number of tests          20
#       Total number of pass           20
#
# ** Note: $stop    : C:/Users/Nada/Downloads/prj_01/prj_01_tb.v(169)
#    Time: 205 ns  Iteration: 0  Instance: /prj_01_tb
```

Figure 5.2: Transcript output after Run-All

Recall that if any of the operands or operation code is unknown, incorrectly or not defined, the 'result' register will output an 'x' (Figure 5.3).

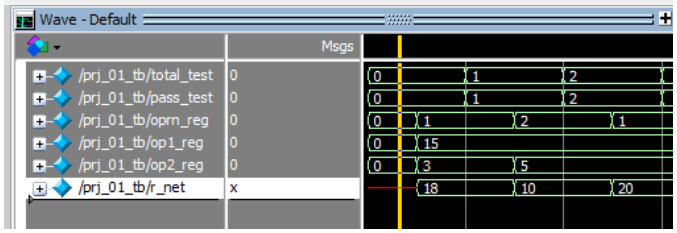

Figure 5.3: Undefined/Unknown result waveform

Observation of output waveforms will be done once for each of the 9 operations. For precision, you can zoom in and out by right-clicking in the wave window. You can observe each operation waveform by moving the yellow vertical line through execution time, noticing the time elapsed between each value change (for the first two operations op1 stays the same) in picoseconds.You can locate the operation on the waveform by looking at the order the operation was done or by its number in the testing process (total_test to check what operation and pass_test to check the last passed operation):

(1) Addition

Setup:

op1=15;

op2=3;

opn=01;

'oprn= 01' indicates that the case of 'h01' will be performed which is addition: result = op1 + op2 = 18 (figure 5.4).

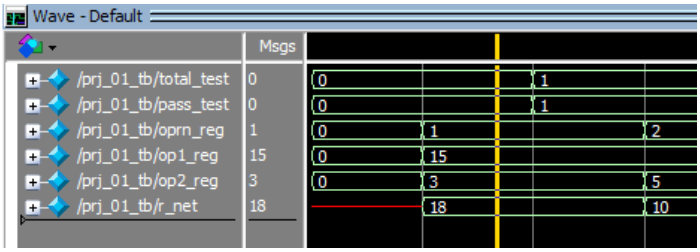

Figure 5.4: Waveform output of addition

(2) Subtraction

op1=15;

op2=5;

oprn=02;

'oprn= 02' indicates that the case of 'h02' will be performed which is subtraction (notice how oprn_reg changes in figure 5.5): result = op1 - op2 = 10 (figure 5.5).

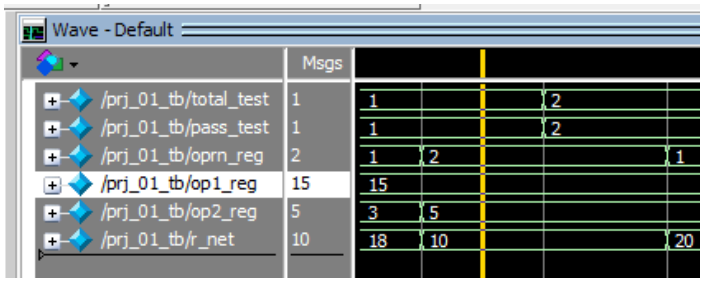

Figure 5.5: Waveform output of subtraction

(3) Multiplication

op1=5;

op2=10;

oprn=03;

'oprn= 03' indicates that the case of 'h03' will be performed which is multiplication: result = op1 * op2 = 50 (figure 5.6).

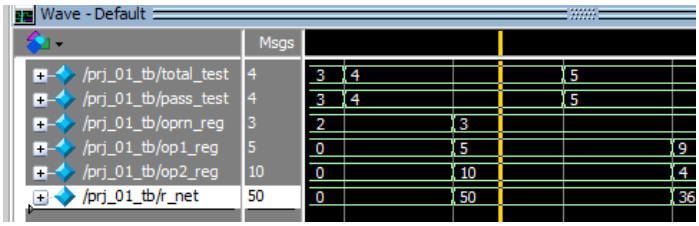

Figure 5.6: Waveform output for multiplication

(4) Shift Right

op1=8;

op2=3;

oprn=04;

'oprn= 04' indicates that the case of 'h04' will be performed which is multiplication: result = op1 >> op2 = 1 (figure 5.8). In this operation, to make shifting simpler you can use binary format for operands by putting them in this format: 'b####. 'b' for binary. So op1='b1000, op2='b0011 and result= op1 >> op2 = 'b0001 which is 1 (figure 5.8).

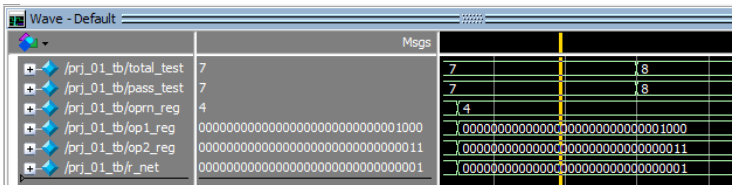

Figure 5.8: Waveform output for Shift Right

(5) Shift Left

For the second test for this operation:

op1=2;

op2=1;

oprn=05;

'oprn= 05' indicates that the case of 'h05' will be performed which is multiplication: result = op1 << op2 = 4 (figure 5.9). In this operation, to make shifting simpler you can use binary format for operands by putting them in this format: 'b####. 'b' for binary. So op1='b0010, op2='b0001 and result= op1 << op2 = 'b0100 which is 4 (figure 5.9).

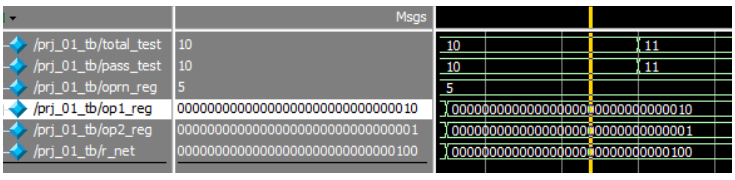

Figure 5.9: Waveform output of Shift Left

(6) Bitwise AND

op1=1 (or 'b0001);

op2=15 (or 'b1111);

oprn=06;

'oprn= 06' indicates that the case of 'h06' will be performed which is Bitwise AND: 0001 AND 1111 will result with 0001 so result= op1 & op2 = 'b0001 which is 1 (figure 5.10).

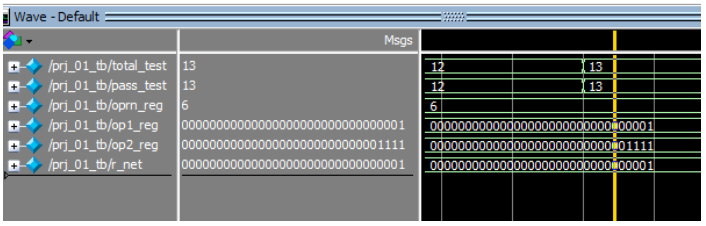

Figure 5.10: Waveform output of Bitwise AND

(7) Bitwise OR

op1=1 ('b0001);

op2=5 ('b0101);

oprn=07;

'oprn= 07' indicates that the case of 'h07' will be performed which is Bitwise OR: result = op1 | op2 = 5(figure 5.11): 0001 OR 0101 will result in 0101 which is 5 in decimal.

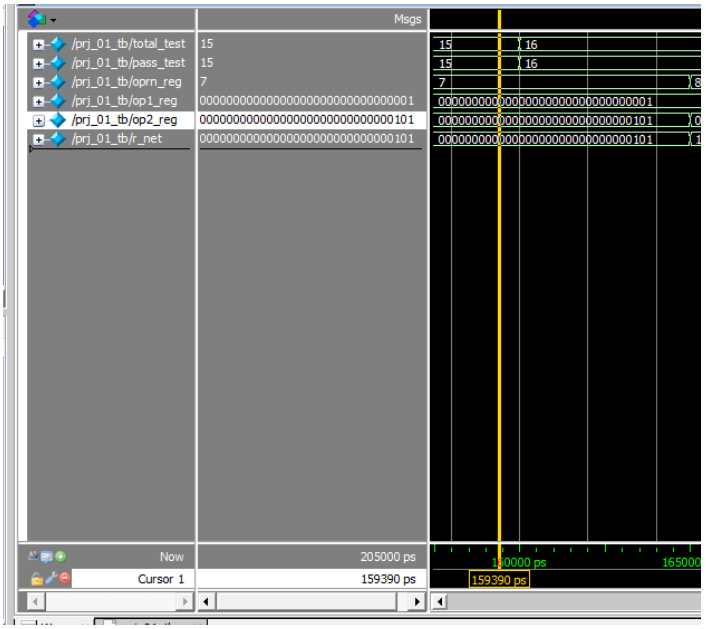

Figure 5.11: Waveform output for Bitwise OR

(8) Bitwise NOR

op1=1 (‘b0001);

op2=4 (‘b0100);

oprn=08;

‘oprn= 08’ indicates that the case of ‘h08’ will be performed which is Bitwise NOR: result = op1 ~| op2 = 1 (figure 5.12): 0001 NOR 1000 will result in extended:11111111111111111111111111111010 (4294967290 unsigned)

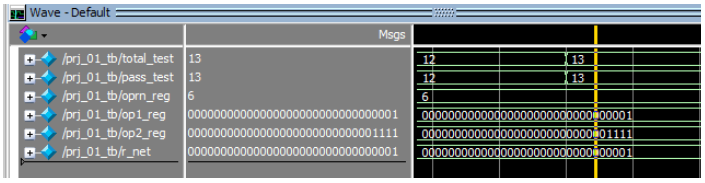

Figure 5.12: Waveform output for Bitwise NOR

(9) Set Less Than

op1=9;

op2=1;

oprn=09;

‘oprn= 09’ indicates that the case of ‘h09’ will be performed which is Set Less Than: result = op1 < op2 = 0 meaning false, 9 is not less than 1(figure 5.13).

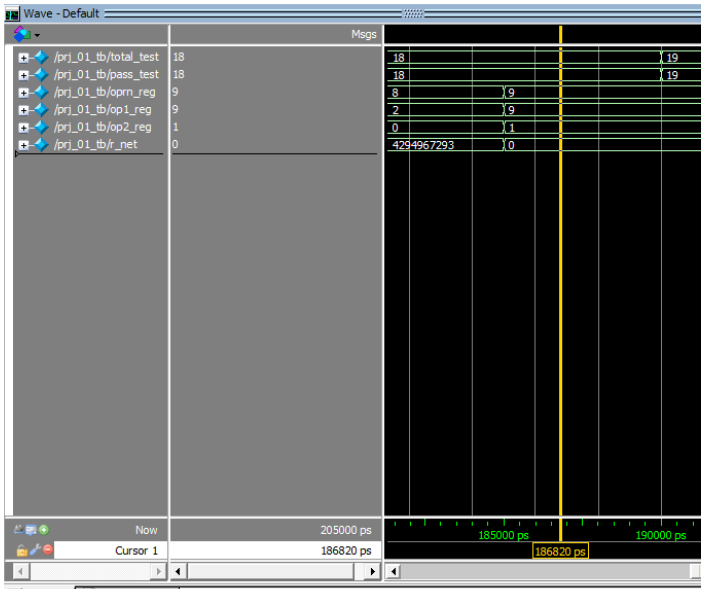

Figure 2.13: Waveform output of Set Less Than

Figure 5.14 shows waveform output of all 20 operations tried in this project to test the implementation of the ALU:

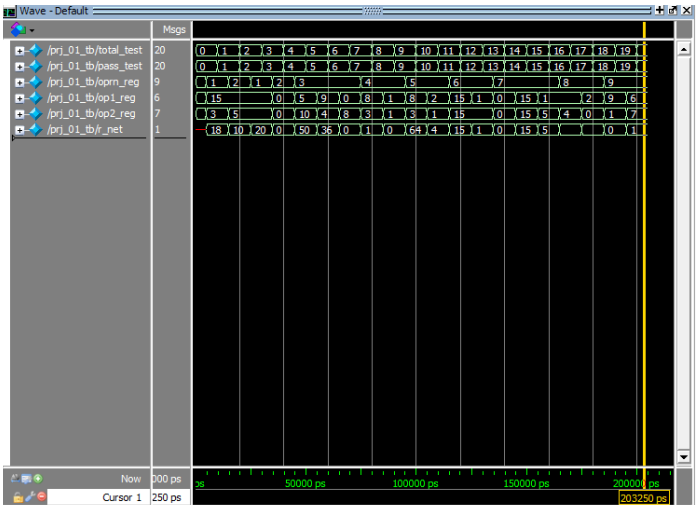

Figure 5.14: Waveform output of all operations

VI. CONCLUSION

This project has undoubtedly delivered many important lessons. I am now able to install a simulation tool ‘ModelSim’, access an operating system using Virtual Box, I learned the fundamentals and principles of the ALU, its basic design and I am now confident in implementing a 32-bit ALU that can handle 9 foundational operations. Developing a test strategy and implementation I believe are the most necessary parts of this project as they validate all the work done. One last achievement through this project would be getting familiar with Verilog, an underlying hardware language.

VII. REFERENCES

- Digital Design (4th Edition) by M. Morris Mano and Michael D. Ciletti .
- Computer Organization and Design (5th edition) by David A. Patterson & John L. Hennessy.
- FPGA Design Flow Overview, retrieved from www.xilinx.com/support/documentation/sw_manuals/xilinx10/isehelp/ise_c_simulation_test_bench.html.
- CS147 Lectures SJSU Spring 2019