Introduction to Simulation Tool, and Implementation of Arithmetic Logic Unit (ALU) module.

Student Anonymous
Computer Science Dept., San Jose State University
California United States
student.anonymous@sjsu.edu

I. Introduction

HDL simulator allows us to design and program logics of a digital circuit using a Hardware Design Language (HDL). In this project, we will use Verilog as a language, and ModelSim as a simulator. The objectives of this project are as follows:

1. To install and setup the simulation tool
2. To implement ALU module using HDL.
3. To implement testing of ALU module using HDL.
4. To simulate and observe signal waveforms using ALU test bench.

This report describes how to install and set up ModelSim. It explores the basic Verilog code and functionalities of ModelSim by implementing and testing arithmetic logic unit (ALU) module. Lastly, it examines the test results and waveforms on the ALU module to verify the accuracy of the implementation, and to observe the simulated circuit behavior.

II. Installation and Setup

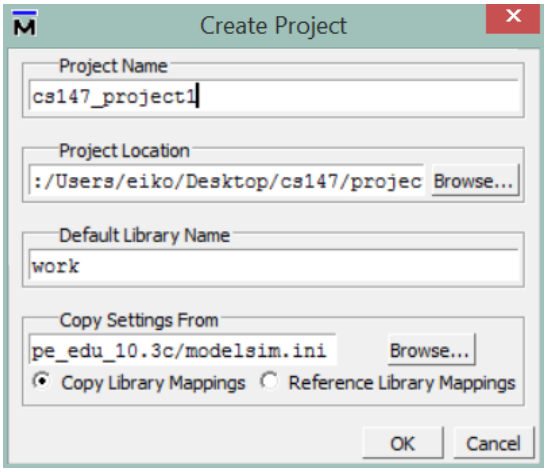
A. *Installation of the simulation tool*

ModelSim student edition is available for download from URL: http://www.mentor.com/company/higher_ed/modelsim-student-edition. Once downloaded, double-click on the binary to start the installation process. Click "Next" on the popup windows to proceed with the installation. In the license agreement screen, click "Yes" to accept the agreement. Accept the default destination directory and program folder.

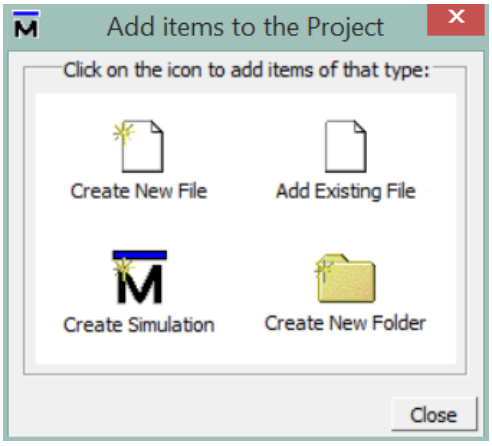
After installation is completed, a new browser window will open with an online license request form. Fill out the form and submit it. The license is emailed to the email address entered in the form. Upon receipt of the license file, copy it in the top ModelSim installation directory. The license is good for 180 days.

B. *Creation of Simulation project*

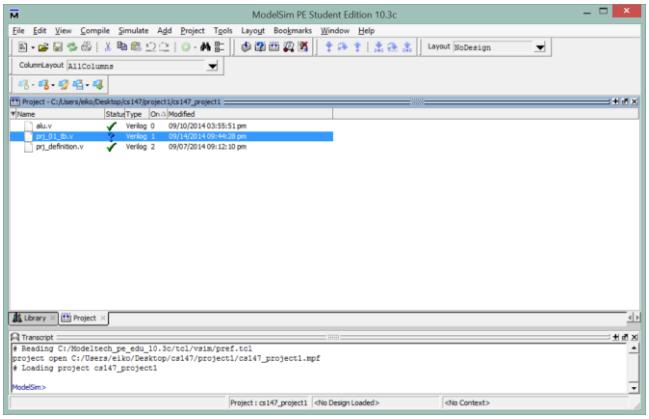
Once the tool is started, select "New" from the "File" menu. Select "Project…" In a "Create Project" screen, enter the Project Name (e.g. "cs147_project1") and select a project location using the "Browse…" button. Use default values for everything else, and click OK.



After that, the below screen will be shown. Since a starter code is already provided, select "Add Existing File" option.



Using the file browser, navigate to the directory which contains the starter source code, and select the files to add to the project. A new project is created with the selected files.



III. Requirements for ALU

"CS147sec05" instruction set is specifically designed for our CS147 class. The following operations of CS147sec05 should be implemented using HDL:

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

CS147sec05 instruction set includes a 6-bit long "OpCode" and "funct" to indicate which operation should be performed (for example, OpCode 0x01/funct 0x20 is addition). For our ALU implementation, an operation code ("oprn") is used for that purpose. ALU performs a specific operation (such as addition, subtraction etc.) based on the oprn value provided by a caller function. The following table is the list of available operation code and its corresponding operation.

Name	Operation Code
Addition	1
Subtraction	2
Multiplication	3
Shift right logical	4
Shift left logical	5
Bitwise AND	6
Bitwise OR	7
Bitwise NOR	8
Set less than	9

IV. Design and Inplementation of ALU

In HDL implementation, the source and destination registers of the CS147sec05 instruction set are replaced by op1 and op2 input parameters and result output parameter respectively. Similarly, *shamt* of shift instructions is replaced by op2 input parameter. For example, add instruction *R[rd] = R[rs] + R[rt]* shall be translated into *result = op1 + op2* in HDL implementation.

The details of each operation implementation is as follows.

A. Addition

Add operation is done by adding *op1* and *op2* input parameters, and assign the result to *result* output parameter. The operation code for add is h01, so addition is performed when *oprn* is 'h01.

The actual Verilog code is as follows:

```
case (oprn)
    `ALU_OPRN_WIDTH'h01 : result = op1 + op2;
```

B. Subtraction

Subtraction is done by subtracting op1 from op2 input parameter, and setting the result to result ouput parameter. The operation code for subtraction is h02, so subtraction is performed when *oprn* is 'h02.

The actual Verilog code is as follows:

```
xxxx(oprn)xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxx`ALU_OPRN_WIDTH'h02 : result = op1 - op2;
```

C. Multiplication

Multiplication is done by multiplying op1 by op2 input parameter, and setting the result to result ouput parameter. The operation code for multiplication is h03, so multiplication is performed when *oprn* is 'h03.

The actual Verilog code is as follows:

```
case (oprn)
    `ALU_OPRN_WIDTH'h03: result = op1 * op2;
```

D. Shift Right Logical

Shift Right Logical is performed by shifting op1 the number of bits specified by op2 input parameter, using ">>" operator. The result is then set to *result* ouput parameter.

The operation code for Shift right logical is h04, so logical right shift is performed when *oprn* is 'h04.

The actual Verilog code is as follows:

```
case (oprn)
    `ALU_OPRN_WIDTH'h04: result = op1 >> op2;
```

E. Shift Left Logical

Shift Left Logical is performed by shifting op1 the number of bits specified by op2 input parameter, using "<<" operator. The result is then set to *result* ouput parameter.

The operation code for Shift left logical is h05, so logical left shift is performed when *oprn* is 'h05.

The actual Verilog code is as follows:

```
case (oprn)
    `ALU_OPRN_WIDTH'h05: result = op1 << op2;
```

F. Logical AND

Logical AND is computed using op1 by op2 input parameters using "&" operator, and its result is set to *result* ouput parameter.

The operation code for logical AND is h06, so logical AND is performed when *oprn* is 'h06.

The actual Verilog code is as follows:

```
case (oprn)
    `ALU_OPRN_WIDTH'h06: result = op1 & op2;
```

G. Logical OR

Logical OR is computed using op1 by op2 input parameters, using "|" operator. Its result is then set to *result* ouput parameter.

The operation code for logical OR is h07, so logical OR is performed when *oprn* is 'h07.

The actual Verilog code is as follows:

```
case (oprn)
    `ALU_OPRN_WIDTH'h07: result = op1 | op2;
```

H. Logical NOR

Logical OR is computed using op1 by op2 input parameters, and its result is set to *result* ouput parameter. It takes the bitwise OR of op1 and op2, and inverse the result using "~" operator.

The operation code for logical NOR is h08, so logical NOR is performed when *oprn* is 'h08.

The actual Verilog code is as follows:

```
case (oprn)
    `ALU_OPRN_WIDTH'h08: result = ~(op1 | op2);
```

I. Set Less Than

Set less than is computed by comparing op1 and op2 input parameters using '<' operator, and its result is set to *result* ouput parameter. The result will be 1 if op1 is less than op2, and 0 if op1 is larger than op2.

The operation code for Set Less Than is h09, so this operation is performed when *oprn* is 'h09.

The actual Verilog code is as follows:

```
case (oprn)
    `ALU_OPRN_WIDTH'h09: result = op1 < op2;
```

V. Test Strategy and Test Implementation

This section describes how test cases are designed, and how they are implemented.

A. Test Cases

In order to test the ALU implementation, 26 test cases have been created as shown in the table in the following page.

Since op1, op2 and results are 32 bit in length, MAX in the table indicate the largest possible value in 32-bit integer, which is 4294967295.

"~15" in case no. 21 indicates an inverse of 15, which has a binary representation of all zero's except for the least significant four bits. It is to simplify the test result, (15) so it will be easier to check with human eyes.

Values begin with 'b' (such as "b0000) are binary numbers. Values in parenthesis are decimal presentation of the same value.

no.	operation	op1	op2	expected result
1	addition	15	3	18
2	addition	15	5	20
3	subtraction	15	5	10
4	subtraction	0	0	0
5	multiplication	3	3	9
6	multiplication	0	3	0
7	right shift	b1000 (8)	1	b0100 (4)
8	right shift	b1000 (8)	2	b0010 (2)
9	right shift	b1000 (8)	3	b0001 (1)
10	right shift	b1000 (8)	4	b0000 (0)
11	left shift	b0001 (1)	1	b0010 (2)
12	left shift	b0001 (1)	2	b0100 (4)
13	left shift	b0001 (1)	3	b1000 (8)
14	left shift	b0001 (1)	4	b10000 (16)
15	bitwise and	b0000	b0000	b0000 (0)
16	bitwise and	b0000	b1111	b0000 (0)
17	bitwise and	b1111	b1111	b1111 (15)
18	bitwise or	b0000	b0000	b0000
19	bitwise or	b0000	b1111	b1111 (15)
20	bitwise or	b1111	b1111	b1111 (15)
21	bitwise nor	0	~15	15
22	bitwise nor	0	MAX	0
23	bitwise nor	MAX	MAX	0
24	Less than	1	2	1
25	Less than	1	1	0
26	Less than	5	1	0

The above test cases are translated into Verilog in the test bench. The op1, op2 values are assigned into Verilog input parameters op1_reg, op2_reg respectively. Then the ALU module is invoked using a test_and_count function, which then compare the function return value and the expected value, then increment the total_test and pass_test counters. Total_test counter indicates the number of test that has been run, and pass_test keeps track of the number of tests that passed.

An example Verilog code which invokes the ALU module is as follows:

```
// test no1.
// 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_golden() now has additional logic to compute the expected value ("golden") for each of the 9 operations. It is also responsible for printing the test result on the console. The related Verilog code looks like the following:

```
`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("AND ");
                      golden = op1 & op2; end
`ALU_OPRN_WIDTH'h07 : begin $write("OR ");
                      golden = op1 | op2; end
`ALU_OPRN_WIDTH'h08 : begin $write("NOR ");
                      golden = ~(op1 | op2); end
`ALU_OPRN_WIDTH'h09 : begin $write("< ");
                      golden = op1 < op2; end
```

The ALU module computation results are tested against the golden values. The test is considered "passed" when the ALU computation result equals the golden value for a corresponding operation. The total test count and pass counts are also kept in total_test and pass_test variables respectively.

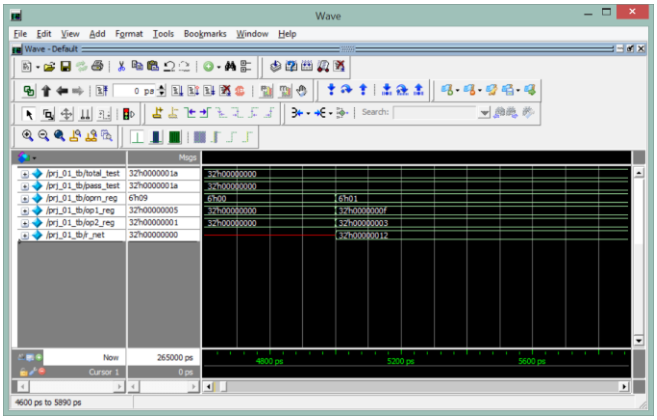
B. Test Results

The test results were observed in the simulator console, as well as in the Wave screen. Below is the console output of the test bench execution. It shows of all the 26 test cases, 26 passed (100% pass rate). The output values also match the expected values in the test cases table above.

```
# [TEST] 15 + 3 = 18 , got 18 ... [PASSED]
# [TEST] 15 + 5 = 20 , got 20 ... [PASSED]
# [TEST] 15 - 5 = 10 , got 10 ... [PASSED]
# [TEST] 0 - 0 = 0 , got 0 ... [PASSED]
# [TEST] 3 * 3 = 9 , got 9 ... [PASSED]
```

```
# [TEST] 0 * 3 = 0 , got 0 ... [PASSED]
# [TEST] 8 >> 1 = 4 , got 4 ... [PASSED]
# [TEST] 8 >> 2 = 2 , got 2 ... [PASSED]
# [TEST] 8 >> 3 = 1 , got 1 ... [PASSED]
# [TEST] 8 >> 4 = 0 , got 0 ... [PASSED]
# [TEST] 1 << 1 = 2 , got 2 ... [PASSED]
# [TEST] 1 << 2 = 4 , got 4 ... [PASSED]
# [TEST] 1 << 3 = 8 , got 8 ... [PASSED]
# [TEST] 1 << 4 = 16 , got 16 ... [PASSED]
# [TEST] 0 AND 15 = 0 , got 0 ... [PASSED]
# [TEST] 15 AND 0 = 0 , got 0 ... [PASSED]
# [TEST] 15 AND 15 = 15 , got 15 ... [PASSED]
# [TEST] 0 OR 0 = 0 , got 0 ... [PASSED]
# [TEST] 0 OR 15 = 15 , got 15 ... [PASSED]
# [TEST] 15 OR 15 = 15 , got 15 ... [PASSED]
# [TEST] 0 NOR 4294967280 = 15 , got 15 ... [PASSED]
# [TEST] 0 NOR 4294967295 = 0 , got 0 ... [PASSED]
# [TEST] 4294967295 NOR 4294967295 = 0 , got 0 ... [PASSED]
# [TEST] 1 < 2 = 1 , got 1 ... [PASSED]
# [TEST] 1 < 1 = 0 , got 0 ... [PASSED]
# [TEST] 5 < 1 = 0 , got 0 ... [PASSED]
#
#          Total number of tests      26
#          Total number of pass       26
```

In addition to the console output, Wave screen also provides time elapsed changes of the values.



It show that all values are initialized to zero at 0 picoseconds. The first value change occurs at 5000 picoseconds, where the following new values are set:

- oprn_reg = 1
- op1_reg = 15
- op2_reg=3
- r_net = 18.

The second value change occurs at 10,000 picoseconds, where total_test and pass_test are incremented by one. These 5000 picosecond interval corresponds to the pause "#5" command entered in the test bench right before the op1_reg and op2_reg assignment, and before calling test_and_count where total_test and pass_test counters are incremented. Also the waveforms show the result is computed and updated as soon as op1_req and op2_reg value changes.

VI. Conclusion

The simulation tool was installed successfully, and the project was created without a problem using the starter source code provided. The ALU module was implemented according to the specification, and was tested against the prepared test cases using a test bench.

The console output of the ALU execution results match the expected values in the test plan, and test workbench also indicated all of the 26 test cases passed. The waveforms show the value changes also occur at the exact timing specified in the Verilog code. It is therefore concluded that ALU are implemented correctly and work as expected.