Introduction to ModelSim and Designing an Arithmetic Logic Unit (ALU)

Michael Wong
Computer Science Department, San Jose State University
California United States
michael.j.wong@sjsu.edu

***Abstract*—This project explores the necessary steps to install and utilize the simulation tool, ModelSim, and how to create a functional Arithmetic Logic Unit (ALU). Upon implementation of the ALU by using Verilog HDL, it will be tested to prove its functionality.**

I. Introduction

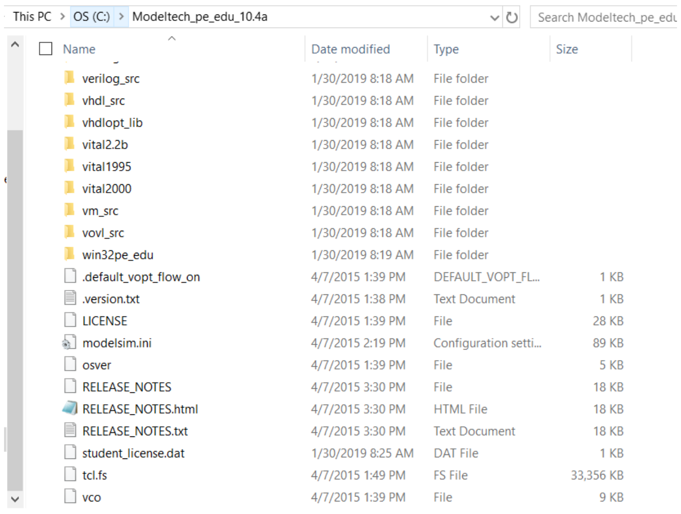
Using a HDLsimulator to write in an HDL language in order to program logic circuits can be a difficult task for beginners. This project will demonstrate how one can use the HDL simulator, ModelSim, to write in the HDL language, Verilog. A simple, yet effective, Arithmetic Logic Unit (ALU) will be implemented and tested as an example in order to show how to use the resources mentioned above. The purpose of this project is to demonstrate how to use ModelSim, Verilog, and to create and test a working ALU.

II. Setting up ModelSim

A. *Installing ModelSim*

In order to start creating a functional ALU, there needs to be an efficient HDL simulator to create it. For this project, I will be utilizing the HDL simulator, ModelSim. The link to download it can be found here: https://www.mentor.com/company/higher_ed/modelsim-student-edition. Follow the steps that are prompt forward, but there is a unique part to the setup. For ModelSim Student Edition to run on your computer, you need to apply for a student license.

After running the .exe file to set up ModelSim on your Windows machine, a pop-up window will appear from your default browser. This window will allow you to fill out a form to obtain a student license. The license will be emailed to the address you provided in the form and simply place the license in ModelSim directory. Once that's complete, you are ready to create your first project writing in the HDL Verilog.



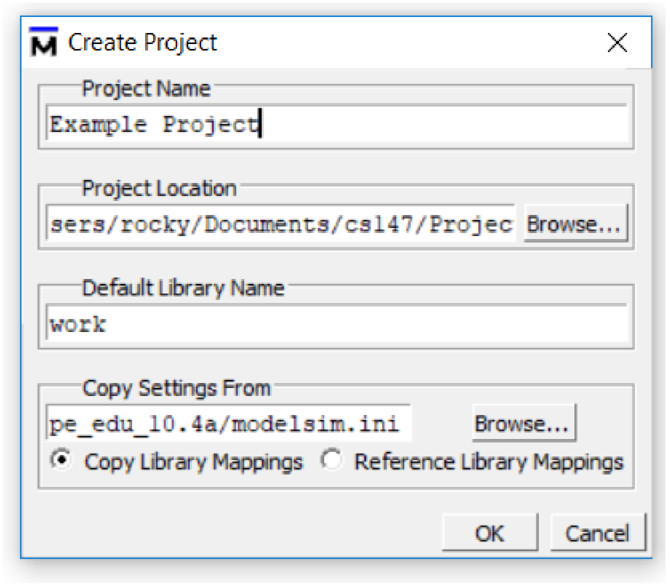
This PC > OS (C:) > Modeltech_pe_edu_10.4a

Name	Date modified	Type	Size
verilog_src	1/30/2019 8:18 AM	File folder	
vhdl_src	1/30/2019 8:18 AM	File folder	
vhdlopt_lib	1/30/2019 8:18 AM	File folder	
vital2.2b	1/30/2019 8:18 AM	File folder	
vital1995	1/30/2019 8:18 AM	File folder	
vital2000	1/30/2019 8:18 AM	File folder	
vm_src	1/30/2019 8:18 AM	File folder	
vovl_src	1/30/2019 8:18 AM	File folder	
win32pe_edu	1/30/2019 8:19 AM	File folder	
.default_vopt_flow_on	4/7/2015 1:39 PM	DEFAULT_VOPT_FL...	1 KB
.version.txt	4/7/2015 1:38 PM	Text Document	1 KB
LICENSE	4/7/2015 1:39 PM	File	28 KB
modelsim.ini	4/7/2015 2:19 PM	Configuration setti...	89 KB
osver	4/7/2015 1:39 PM	File	5 KB
RELEASE_NOTES	4/7/2015 3:30 PM	File	18 KB
RELEASE_NOTES.html	4/7/2015 3:30 PM	HTML File	18 KB
RELEASE_NOTES.txt	4/7/2015 3:30 PM	Text Document	18 KB
student_license.dat	1/30/2019 8:25 AM	DAT File	1 KB
tcl.fs	4/7/2015 1:49 PM	FS File	33,356 KB
vco	4/7/2015 1:39 PM	File	9 KB

B. *Creating your First Project*

To create a new project in ModelSim, open ModelSim which will take you to the main page. Navigate to the files tab, click new and then project. A pop-up menu will appear asking for a name for the project and the location that you want it to be in. Click "ok" and the project is made. Afterwards, you can choose to either create a new file for the project or to add already existing files to the project.

.



Project Name: Example Project

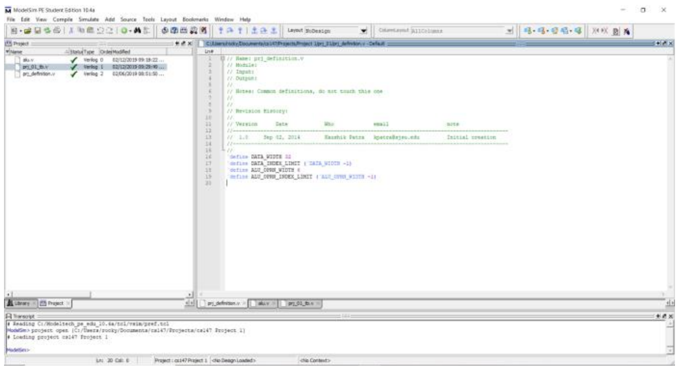
Project Location: sers/rocky/Documents/cs147/Projec

Default Library Name: work

Copy Settings From: pe_edu_10.4a/modelsim.ini

[✓ Copy Library Mappings] Reference Library Mappings

Since starter code has already been provided to me, I chose "add existing files." A file directory will open and use that to select the files that you want to upload to the project. Upon uploading the three files, your workspace should look like the image below:



```
// Name: prj_definition.v
// Module:
// Input:
// Output:
//
// Notes: Common definitions, do not touch this one
//
//
// Revision History:
//
// Version     Date           Who            email                 note
//------------------------------------------------------------------------
//  1.0     Sep 02, 2014      Kaushik Patra  kpatra@sjsu.edu       Initial creation
//------------------------------------------------------------------------
//
`define DATA_WIDTH 32
`define DATA_INDEX_LIMIT (`DATA_WIDTH -1)
`define ALU_OPRN_WIDTH 6
`define ALU_OPRN_INDEX_LIMIT (`ALU_OPRN_WIDTH -1)
```

III. Requirements for the ALU

In the task prompt, the requirements and necessities for the ALU were stated. Various operations and their behaviors were detailed and explained to implement the ALU. The following table breaks down the operations required for the ALU. The table's information was provided thorough the lecture slides.

Name of Operation	Mnemonic	Format of 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

On the subject of the operation process, there are three variables that are being utilized; rd, rs and rt. In the case for this project, when a shift operation is being called, the shamt value will be represented as the st variable.

IV. Designing the ALU

The starter code that was provided to me at the start of the project included three inputs and one output. The three input variables that were provided were op1, op2 and oprn. Op1 and op2 are to represent rs and rt respectively in the operation column of the table. Next, the purpose of oprn is to flag which operation is being performed on op1 and op2. For example, in the project code, if oprn was, "h01," then this indicates to perform an addition operation on op1 and op2. The one output that is provided, named result, is to store the result of the operation between op1 and op2. The way that the ALU is initially setup is that as long as any of the inputs are changed in anyway, it will loop through to find and perform the desired operation on op1 and op2. This functionality is displayed below:

```
always @ (op1 or op2 or oprn)
begin
   case (oprn)
      `ALU_OPRN_WIDTH'h01 : result = op1 +
op2;        // addition
   //
      // Finish the starter code here…
      //
```

```
        default: result = `DATA_WIDTH'hxxxxxxxx;

    endcase
end
```

The following sections will reveal more details regarding the implementing operations.

A. Addition

The addition operation stores the answer of op1 + op2 into result. The oprn for addition is h01. Below is the Verilog code:

```
`ALU_OPRN_WIDTH'h01 : result = op1 + op2;
```

B. Subtraction

The subtraction operation stores the answer of op1 - op2 to result. The oprn for subtraction is h02. Here is the Verilog code:

```
`ALU_OPRN_WIDTH'h02 : result = op1 - op2;
```

C. Multiplication

The multiplication operation allocates the answer of op1 * op2 into result. The oprn of multiplication is h03. Here is the Verilog code:

```
`ALU_OPRN_WIDTH'h03 : result = op1 * op2;
```

D. Shift Right Logical

The right shift logical operation sets the result of bit shifting op1 to the right. The bit number to shift op1 is determined by op2. The operation symbol used is “>>.” The oprn for a right shift logical is h04. The Verilog code can be found here:

```
`ALU_OPRN_WIDTH'h04 : result = op1 >> op2;
```

E. Shift Left Logical

The left shift logical operation sets the result of bit shifting op1 to the left. Op2 determines how many bits op1 will be left shifted. The operation symbol used is “<<.” The oprn for a left shift logical is h05. Here is the Verilog code:

```
`ALU_OPRN_WIDTH'h05 : result = op1 << op2;
```

F. AND

The bitwise AND operation calculates the result of anding op1 and op2 and sets it into result. It is implementing a bitwise calculator for the AND operation. The operation symbol used is “&.” The oprn for an AND is h06. Here is the Verilog code:

```
`ALU_OPRN_WIDTH'h06 : result = op1 & op2;
```

G. OR

The bitwise OR operation calculates the result of oring op1 and op2 and placing it into result. Similar to the AND operation, it behaves just like a bitwise calculator for the OR operation. The operation symbol used is “|.” The oprn for an OR is h07 Consider the Verilog code below:

```
`ALU_OPRN_WIDTH'h07 : result = op1 | op2;
```

H. NOR

The bitwise NOR operation stores the result of noring op1 and op2 and storing it into result. Just like the OR operation, it acts like a bitwise calculator except for a NOR logical. The operation expression that needs to be used is “~(a|b).” The oprn for a NOR is h08. Here is the Verilog code:

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

I. Set Less Than

The set less than operation stores the result of testing if op1 is less than op2. If the op1 is less than op2, the result will be 1. Otherwise result will hold a value of 0. The operation expression that is used is “<.” The oprn for a set less than is h09. Shown below is the Verilog code:

```
`ALU_OPRN_WIDTH'h09 : result = op1 < op2;
```

V. Testing

A. Implementing the Testing

To test my ALU for correctness and proper functionality, I built on the tester code that was provided to me. There are 12 test cases created to perform the 9 different operations. The results of the test cases are then compared against what was expected to be the result. The table below displays the 12 different test cases that I created. I made at least one test case for each of the operations. Certain operations, such as set less than have two cases to test both possible results of the operation. For the NOR tests, I tested the largest signed integer value for a 32-bit integer which is 2147483647 since in binary, this number would be represented in all 1's. I tried to pass 4294967295, the largest unsigned integer value for a 32-bit integer, but ModelSim was giving me a warning of overflow since it is treating it as signed integer. Since I did not know how to resolve this issue and to prevent the warning from affecting anything in the project, I decided to stay with the maximum signed integer of 2147483647.

Name of Operation	Op1	Op2	Expected Result
Addition	15	3	18
Subtraction	15	5	10
Multiplication	5	5	25
Shift Right	15	1	7
Shift Left	12	3	96
AND	3	5	1
OR	7	9	15
NOR	0	0	4294967295
NOR	2147483647	102	2147483648
Set Less Than	12	36	1
Set Less Than	99	9	0

These test cases were also implemented in ModelSim. An example of how the actual code would look like is below:

```
// test 5 * 5 = 25
#5  op1_reg=5;
   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));
```

Repeating these test cases another eleven times would generate the 12 test cases for the 9 different operations that this ALU needs to be functional. The 12 iterations keep track of the actual results and the expected results. The number of tests conducted and how many passed their test is also recorded. The actual performance of the operations is done under the function test_golden. In test_golden, there are nine implementations of the nine operations. This is where the expected value is calculated and stored. An example of the test_golden code to obtain the expected result of the answer is shown below:

```
`ALU_OPRN_WIDTH'h02 : begin $write("- ");
golden = op1 - op2; end
```

Similar code is repeated for the next eight operations in order to test them.

B. *Outputting the Test Results*

The ALU that I've created will generate the actual result and the golden function call provides the expected value. The two are compared with each other to see if they are equal to each other. If they are, the test will be tagged as "passed," otherwise, it will be tagged as "failed."

The outputted result of these test is printed into ModelSim's console terminal. Here is the output result of after running the tester:

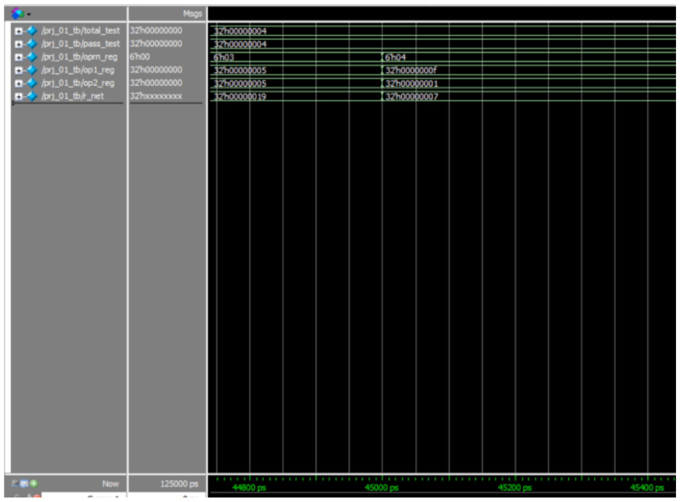
```
# [TEST] 15 + 3 = 18 , got 18 ... [PASSED]
# [TEST] 15 - 5 = 10 , got 10 ... [PASSED]
# [TEST] 15 + 5 = 20 , got 20 ... [PASSED]
# [TEST] 5 * 5 = 25 , got 25 ... [PASSED]
# [TEST] 15 >> 1 = 7 , got 7 ... [PASSED]
# [TEST] 12 << 3 = 96 , got 96 ... [PASSED]
# [TEST] 3 & 5 = 1 , got 1 ... [PASSED]
# [TEST] 7 | 9 = 15 , got 15 ... [PASSED]
# [TEST] 0 ~| 0 = 4294967295 , got 4294967295 ...
[PASSED]
# [TEST] 2147483647 ~| 102 = 2147483648 , got
2147483648 ... [PASSED]
# [TEST] 12 < 36 = 1 , got 1 ... [PASSED]
# [TEST] 99 < 9 = 0 , got 0 ... [PASSED]
```

```
#
#     Total number of tests          12
#     Total number of pass            12
#
# ** Note: $stop    :
C:/Users/rocky/Documents/cs147/Projects/Project
1/prj_01/prj_01_tb.v(135)
#    Time: 125 ns  Iteration: 0  Instance: /prj_01_tb
```

As evident from the printed output, my ALU passes all the tests that were created for it. The ALU is working properly and reaches the necessary requirements specified by the project description.

C. Waveforms

I can further analyze the functionality of my implemented ALU by observing the waveforms that they produced in ModelSim. Each register in the testing class of the project will produce a waveform for the different pieces of information that is passed through them. I can observe these waves as electric signals that give a more in-depth insight on what is happening within the project. Below is an image of the waveforms as the program passes through testing the shift right test case.


Msgs
/prj_01_tb/total_test 32'h00000000 32'h00000004
/prj_01_tb/pass_test 32'h00000000 32'h00000004
/prj_01_tb/oprn_reg 6'h00 6'h03 6'h04
/prj_01_tb/op1_reg 32'h00000000 32'h00000005 32'h0000000f
/prj_01_tb/op2_reg 32'h00000000 32'h00000005 32'h00000001
/prj_01_tb/r_net 32'hxxxxxxxx 32'h00000019 32'h00000007
Now 125000 ps
44800 ps
45000 ps
45200 ps
45400 ps

I can see that the op1 signal is holding a 15 along with the op2 signal holding a 1. The signal above them represents the oprn signal which is holding a h04. The three signals combined demonstrates the operation of shifting 15 right by 1 bit. Lastly the bottom signal is the result wave showing the result of this operation to be 7.

VI. Conclusion

In conclusion, I was able to successfully install a working HDL simulator and using that simulator, implement and test an ALU. I did not face any problems with the installation of ModelSim. The process was simple with obtaining a student license being the only tricky part. Once ModelSim was running properly, I had to learn the Verilog language. This was probably the hardest part of the project since I'm not familiar with this language and I had to get a crash course on it using multiple resources around me. After figuring out the basic fundamentals of Verilog I was able to easily create the ALU and its testers. The tests confirmed my implementation of the ALU showing that it is working properly. Overall, this was a simple and straightforward project that was able completed without much trouble. It was a good starting out project for beginners and I look forward to applying more advanced skills in future projects.