# Arithmetic Logic Unit(ALU) Implementation and ModelSim Simulator Introduction

Xinken Zheng, CS147 Student, San Jose State University

## I. Introduction

The project helps to show how one can use Hardware Design Language (HDL) to program a digital circuit using the HDL simulator. The programming language that will be used for this project is Verilog, and the program/simulator we will be using is ModelSim. The goals for this project will be:

1. Have ModelSim installed and set up successfully.
2. Using Hardware Design Language (HDL) to implement Arithmetic Logi Unit module and its test cases.

This project report will show the tools and steps needed to set up ModelSim. It will show simple functions of Verilog code and uses for ModelSim with implementation and test cases for Rrithmetic Logic Unit.

## II. Tools needed for this project

Before starting, we need to have ModelSim student edition downloaded and the project files that are given. You will also need to understand how to use logical operations.

### *A. Downloading ModelSim and required files.*

In order to run this program you have to download the simulator ModelSim (which is the simulator that will be used in this program.) The files we will need for this project is given from the prompt, “alu.v”, “prj_01_tb.v”, “prj_definition.v”,

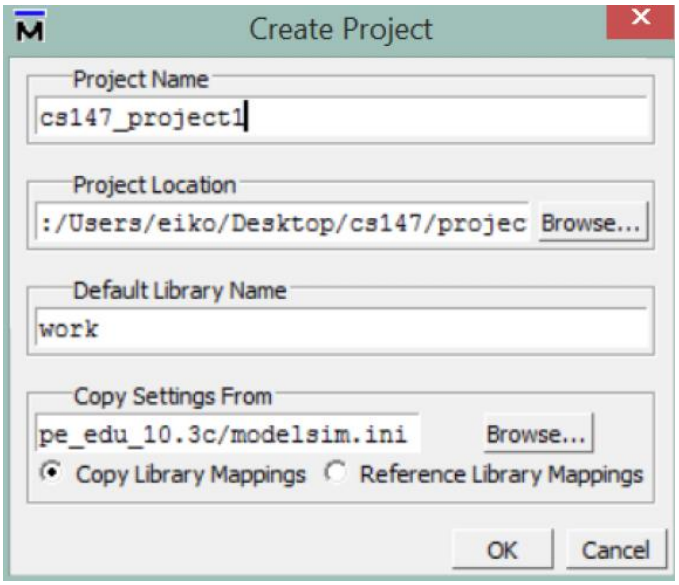
ModelSim is available for free download at the following website:

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

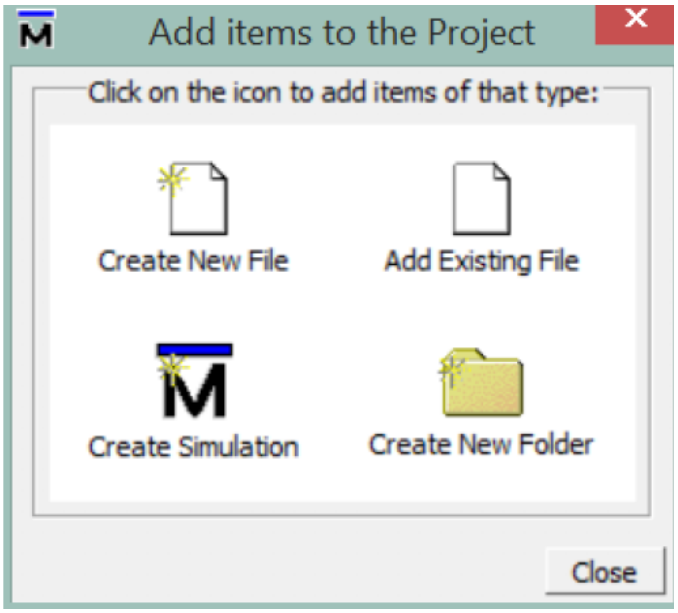
After installing ModelSim, User will have to fill out a form for license request in order to use ModelSim. When it is done, User will get a new file emailed to the addressed that is entered. User will have to copy and paste the license file to the top of the directory where ModelSim is installed. License will be good for this semester.

### *B. Start by creating a project*

Create a new project by selecting “File” → “New” →”Project” →”create a project”. Enter an appropriate name for the project(e.g. “CS147Pro_1”) and save it to an appropriate location in your computer, keep everything else the same.




After that, click “Add Existing File” and import all the files that are given in the prompt in the directory that you saved them in.




## III. ALU REQUIREMENT

We will be implementing the following instruction set using Hardware Design Language.

| 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 |

## IV. DESIGN AND IMPLEMENTATION

### A. *Addition*

*1) Addition*

We start with performing add operation to op1 and op2, op1 will be added to op2 and the result will be stored in the "result" variable. Addition is performed when oprn width is 1, ALU_OPRN_WIDTH`h01.

Code:

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

*2) Subtraction*

This same work will be done very similar to Addition except that it will be subtraction instead of addition. Subtraction operation is done when oprn width is 2, ALU_OPRN_WIDTH`h02. op1 will be subtracting op2 and result will be stored in the result variable.

Code:

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

*3) Multiplication*

The same work will be done like Addition and Subtraction, instead Multiplication operation is done when oprn width is 3, ALU_OPRN_WIDTH`h03.

Code:

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

*4) Shift Right Logical*

Shift Right Logical operation will shift the bits of the value in op1 by the value of op2. This operation is done using ">>" operator. The result will again will be stored in "result". Operation code for this Operation will be with ALU width is `h04.

Code:

```
`ALU_OPRN_WIDTH'h04 : result = op1 >> op2; //shift right
```

*5) Shift Left Logical*

Shift Left Logical operation is the same with Shift Right Logical except that the operation code for Shift Left is when ALU width is `h05.

Code:

```
`ALU_OPRN_WIDTH'h05 : result = op1 << op2; //shift left
```

*6) Bitwise AND*

Bitwise AND is determined by comparing op1 and op2 using "&" operator. The operation code for Bitwise AND is when Alu width is `h06.

Code:

```
`ALU_OPRN_WIDTH'h06 : result = op1 & op2; //Bitwse AND
```

*7) Bitwise OR*

Bitwise OR is calculated by comparing op1 and op2 using "|" operator. Operation code for Bitwise OR is `h07.

Code:

```
`ALU_OPRN_WIDTH'h07 : result = op1 | op2; //Bitwise OR
```

*8) Bitwise NOR*

Bitwise NOR is calculated by Using the Bitwise OR operation on op1 and op2, and negating it by using "~' and a parenthesis around op 1 and op2. Result is stored in the result variable. The operation code for Bitwise NOR is `H08

Code:

```
`ALU_OPRN_WIDTH'h08 : result = ~(op1 | op2); //Bitwise NOR
```

*9) Set Less than*

Set Less than is computed by setting op1 < op2, the result of the equality will be stored in "result". Result will return 1 if the condition holds true, and 0 if the condition holds false.

Code:

```
`ALU_OPRN_WIDTH'h09 : result = op1 < op2; //Less th
```

## V. TESTING

Before testing, we will have to code the 9 operations listed above to the function test_golden(). This function's purpose is to check and print all test results for our test cases. test_golden() function will tell the user the number of tests the program pass and failed, and the expected and actual value it computed. If the expected value is different than the actual value computed, the test is failed and User has to fix the error.

All of the files have to be compiled before simulation starts, if there's an "X" showing next to the project, it means the project is not compiled and has errors. If the user cannot find the error, they could double click on the "X" logo and it will pop up a window indicating the error.

### A. *Expected Values*

The table below includes all the tests I did and the expected values each test.

| # | Operation | op1 | op2 | Expected result |
|---|---|---|---|---|
| ***1*** | ***Addtion*** | 3 | 13 | 16 |
| ***2*** | ***Subtraction*** | 3 | 2 | 1 |
| ***3*** | ***Addition*** | 45 | 15 | 60 |
| ***4*** | ***Multiplication*** | 7 | 7 | 49 |
| ***5*** | ***Shift Right*** | 8 | 1 | 4 |
| ***6*** | ***Shift Left*** | 1 | 2 | 4 |
| ***7*** | ***Bitwise AND*** | 9 | 9 | 9 |
| ***8*** | ***Bitwise OR*** | 5 | 0 | 5 |
| ***9*** | ***Bitwise NOR*** | 5 | 8 | 4294967282 |
| ***10*** | ***Less than*** | 16 | 76 | 1 |

### B. *Explaintaion:*

*1)* 3+13=16
*2)* 3-2 =1
*3)* 45+15 =60
*4)* 7 * 7 = 49
*5)* 1000=8 shift right 1 is 0100= 4;
*6)* 0001 shift left 2 is 0100 = 4;
*7)* 9 AND 9 = 9;
*8)* 5 OR 0 = 5;
*9)* 5 NOR 8 is 4294967282 (*This value is computed by taking the inverse of 5, and subtracting 8)
5 = fffffffA – 8 = fffffff2
*10)* 16 < 76 = 1;(true)

### C. *Implementation and Coding*

1. Below is the code for the ALU module, combined from all the code we discussed above in "Design and Implementation"

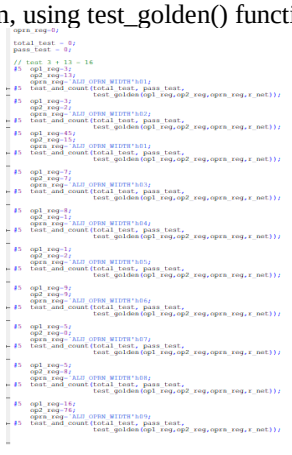
```
begin
    case (oprn)
        `ALU_OPRN_WIDTH'h01 : result = op1 + op2; // addition

        `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; //Bitwse AND

        `ALU_OPRN_WIDTH'h07 : result = op1 | op2; //Bitwise OR

        `ALU_OPRN_WIDTH'h08 : result = ~(op1 | op2); //Bitwise NOR

        `ALU_OPRN_WIDTH'h09 : result = op1 < op2; //Less than

        //
        // TBD: fill up rest of the operations from here
        //
        default: result = `DATA_WIDTH'hxxxxxxxx;

    endcase
end

endmodule
```

2. Below is the code for the function test_golden()

```
begin
    $write("[TEST] %0d ", op1);
    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("Bitwise AND "); golden = op1 & op2; end
        `ALU_OPRN_WIDTH'h07 : begin $write("Bitwise OR "); golden = op1 | op2; end
        `ALU_OPRN_WIDTH'h08 : begin $write("Bitwise NOR "); golden = ~(op1 | op2); end
        `ALU_OPRN_WIDTH'h09 : begin $write("Less than "); golden = op1 < op2; end

        //
        // TBD: fill out for the other operations
        //
        default: begin $write("? "); golden = `DATA_WIDTH'hx; end
    endcase
    $write("%0d = %0d , got %0d ... ", op2, golden, res);

    test_golden = (res === golden)?1'b1:1'b0; // case equality
    if (test_golden)
        $write("[PASSED]");
    else
        $write("[FAILED]");
    $write("\n");
end
endfunction

endmodule
```

### D. *Test cases:*

All ten of the cases given in the table is tested in the program, using test_golden() function.



*Pictures small font for format purposes, can be seen clear if zoom in

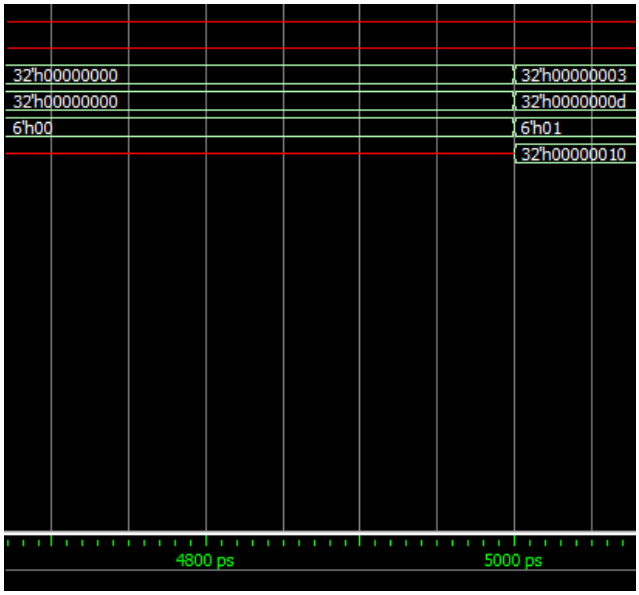
*E. Result:*

The result shows that all the values for test cases were correct, the program got the values we expected and passed all of the tests. This tells us that the implementation for ALU is correct.

```
# [TEST] 3 + 13 = 16 , got 16 ... [PASSED]
# [TEST] 3 - 2 = 1 , got 1 ... [PASSED]
# [TEST] 45 + 15 = 60 , got 60 ... [PASSED]
# [TEST] 7 * 7 = 49 , got 49 ... [PASSED]
# [TEST] 8  >> 1 = 4 , got 4 ... [PASSED]
# [TEST] 1  << 2 = 4 , got 4 ... [PASSED]
# [TEST] 9 Bitwise AND 9 = 9 , got 9 ... [PASSED]
# [TEST] 5 Bitwise OR 0 = 5 , got 5 ... [PASSED]
# [TEST] 5 Bitwise NOR 8 = 4294967282 , got 4294967282 ... [PASSED]
# [TEST] 16 Less than 76 = 1 , got 1 ... [PASSED]
#
#       Total number of tests          10
#       Total number of pass           10
#
```

*F. Wave Graphs:*

Wave graphs helps us to keep track of the time when values are being changed in each operation, in this particular graph, we can see that the value is being changed at 5000 picoseconds, with the first value of 3 in hexadecimal and 13 for the second value in hexadecimal. We can also see the operation code for this operation is 1, which tells us it is an addition operation. Finally, we can see that the result value is 16, in hexadecimal value.




Op1 = 13;
Op2 = 3;
Oprn = 6`h01;
Result = 16;

## CONCLUSION

This project was very challenging for me due to the little knowledge I have with HDL and Verilog. The set up was quite simple, but the environment is quiet hard to use and understand due to its old design. Overall the project was very fun after implementing the ALU and learning the basic functions of Verilog.

Lastly, ModelSim was installed and set up successful implemented and working as expected. It passes all of the 10 test cases and got the right answers. From that, I can conclude That the ALU for the program is implemented correctly.