Arithmetic &Logic Unit(ALU)

Chihkuo Hsu

San Jose State University
chih-kuo.hsu@sjsu.edu

1. Introduction

The HDL simulator enables the design and programming of digital circuit logic using Hardware Description Language (HDL). In this project, Verilog will be used as the design language and ModelSim will be utilized for simulation. The main goals of this project include:

1. Setting up and configuring the simulation tool(ModelSim).
2. Implementing the ALU module using HDL.
3. Testing the functionality of the ALU module.
4. Simulating and analyzing the waves from the ALU testbench.

This report outlines the steps to install and configure ModelSim. It also discusses the use of Verilog to develop and test an arithmetic logic unit (ALU) module. Finally, the report presents the test results and simulated waveforms to validate the ALU's performance and observe the behavior of the circuit.

2. ModelSim installation and setup

1. **Installation**

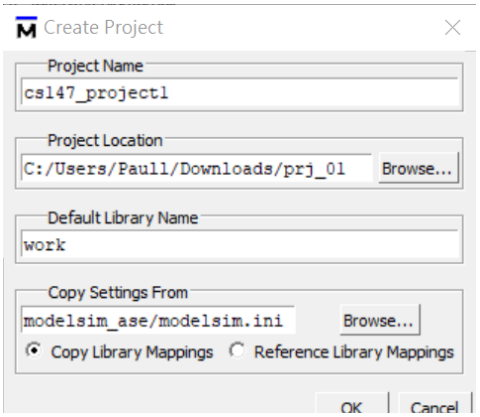
Go to the following website
URL:https://www.intel.com/content/www/us/en/software-kit/750368/modelsim-intel-fpgas-standard-edition-software-version-18-1.html

Download ModelSim-Intel® FPGA Edition (includes Starter Edition)

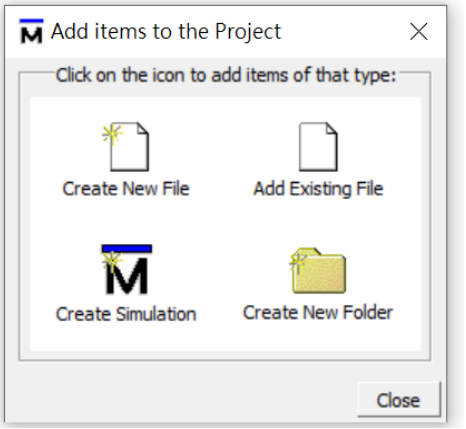
Once the download is complete, open the file, agree to all the agreements, and install.

2. Setup

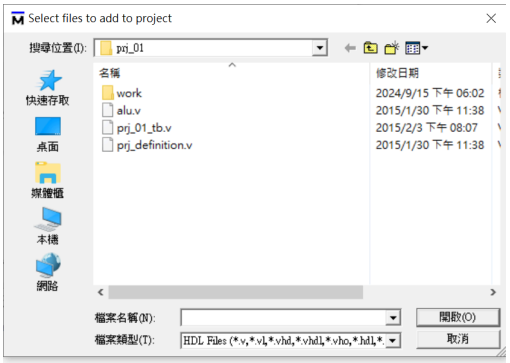
After opening ModelSim, click "New", then select "Project" and choose "Create Project". Enter your project name, click "Browse" to select your project location, and finally, click "OK".



The following screen will appear, select "Add Existing Files".



Select your source code and add all of them.



3. Requirements for ALU

The "CS147DV" instruction set has been created specifically for use in our CS147 class. The operations that need to be implemented using HDL are as follows:

Name: Addition
Mnemonic: add
R[rd] = R[rs] + R[rt]

Name: Subtraction
Mnemonic: sub
R[rd] = R[rs] - R[rt]

Name: Multiplication
Mnemonic: mul
R[rd] = R[rs] * R[rt]

Name: Shift right logical
Mnemonic: srl
R[rd] = R[rs] >> shamt

Name: Shift left logical
Mnemonic: sll
R[rd] = R[rs] << shamt

Name: Bitwise AND
Mnemonic: and
R[rd] = R[rs] & R[rt]

Name: Bitwise OR
Mnemonic: or
R[rd] = R[rs] | R[rt]

Name: Bitwise NOR
Mnemonic: nor
R[rd] = ~(R[rs] | R[rt])

Name: Set less than
Mnemonic: slt
R[rd] = (R[rs] < R[rt]) ? 1 : 0

The CS147DV instruction set includes a 6-bit "OpCode" and "funct" to specify which operation should be executed. For example, OpCode 0x01 combined with funct 0x20 represents addition. In our ALU implementation, the operation code ("oprn") will be used to control which operation is performed. Below are the operation codes:

Name: Addition
Operation Code: 1

Name: Subtraction
Operation Code: 2

Name: Multiplication
Operation Code: 3

Name: Shift right logical
Operation Code: 4

Name: Shift left logical
Operation Code: 5

Name: Bitwise AND
Operation Code: 6

Name: Bitwise OR
Operation Code: 7

Name: Bitwise NOR
Operation Code: 8

Name: Set less than
Operation Code: 9

These codes are utilized to execute the corresponding operations.

4. Implementation of ALU

The ALU (Arithmetic Logic Unit) was developed using Verilog HDL, designed to perform a range of arithmetic and logical operations. This unit takes two 32-bit input operands (**op1** and **op2**) and a 6-bit operation code (**oprn**) to determine the operation to be executed. The result of the selected operation is stored in a 32-bit output (**result**).

A case statement is used to handle various operations based on the value of oprn. The ALU supports essential operations such as addition, subtraction, multiplication, bitwise operations (**AND, OR, NOR**), logical shifts, and comparison for "set less than.

Addition: The addition of op1 and op2 is performed when the oprn code is h01.

`ALU_OPRN_WIDTH'h01 : result = op1 + op2;

Subtraction: Subtraction is triggered by h02, which subtracts op2 from op1.

`ALU_OPRN_WIDTH'h02 : result = op1 - op2;

Multiplication: The multiplication of op1 and op2 is executed when oprn is h03.

`ALU_OPRN_WIDTH'h03 : result = op1 * op2;

Logical Shifts: Right (>>) and left (<<) shifts are performed with h04 and h05, respectively.

`ALU_OPRN_WIDTH'h04 : result = op1 >> op2; // Shift Right
`ALU_OPRN_WIDTH'h05 : result = op1 << op2; // Shift Left

Bitwise Operations: AND, OR, and NOR are executed using h06, h07, and h08.

`ALU_OPRN_WIDTH'h06 : result = op1 & op2; // AND
`ALU_OPRN_WIDTH'h07 : result = op1 | op2; // OR
`ALU_OPRN_WIDTH'h08 : result = ~(op1 | op2); // NOR

Set Less Than (SLT): This operation compares op1 and op2, returning 1 if op1 is less than op2, and 0 otherwise (h09)

`ALU_OPRN_WIDTH'h09 : result = (op1 < op2) ? 1 : 0;

5. Testing the functionality of ALU

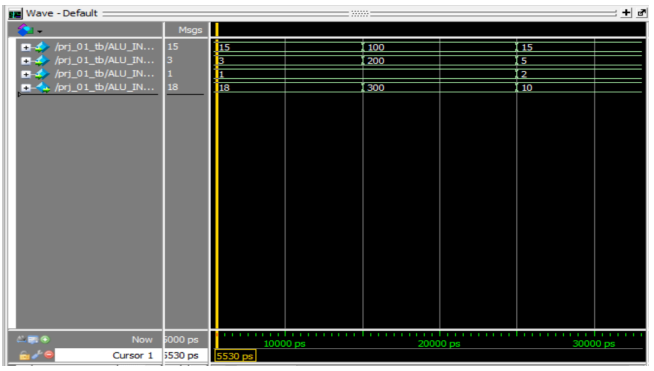
To validate the ALU's functionality, a testbench was created to perform various arithmetic and logical operations. Each test case was designed to cover operations such as addition, subtraction, multiplication, logical shifts, and bitwise operations.
For each test, values were assigned to op1, op2, and oprn to specify the operation. The test result was then compared to the expected "golden value." The test_and_count function was used to verify the result and track the total and passed tests.
A delay of 5-time units (#5) was used between input assignments and result checks to allow for proper signal propagation. The results were displayed in the simulation console, with waveforms confirming the timing and accuracy of the ALU's operations.

[TEST] 15 + 3 = 18 , got 18 ... [PASSED]
[TEST] 100 + 200 = 300 , got 300 ... [PASSED]
[TEST] 15 - 5 = 10 , got 10 ... [PASSED]
[TEST] 200 - 150 = 50 , got 50 ... [PASSED]
[TEST] 4 * 5 = 20 , got 20 ... [PASSED]
[TEST] 10 * 10 = 100 , got 100 ... [PASSED]
[TEST] 16 >> 2 = 4 , got 4 ... [PASSED]
[TEST] 32 >> 1 = 16 , got 16 ... [PASSED]
[TEST] 4 << 2 = 16 , got 16 ... [PASSED]
[TEST] 10 << 1 = 20 , got 20 ... [PASSED]
[TEST] 15 & 3 = 3 , got 3 ... [PASSED]
[TEST] 12 & 5 = 4 , got 4 ... [PASSED]
[TEST] 15 | 3 = 15 , got 15 ... [PASSED]
[TEST] 12 | 5 = 13 , got 13 ... [PASSED]
[TEST] 15 NOR 3 = 4294967280 , got 4294967280 ... [PASSED]
[TEST] 10 NOR 2 = 4294967285 , got 4294967285 ... [PASSED]
[TEST] 5 < 10 = 1 , got 1 ... [PASSED]
[TEST] 10 < 5 = 0 , got 0 ... [PASSED]
#
Total number of tests: 18
Total number of pass: 18
#

5. Analyzing the waves from ALU

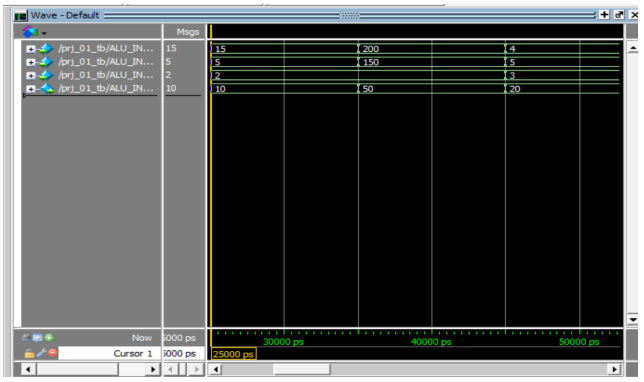
In this section, I analyze the waveforms generated by the ALU testbench, displaying the input and output signals for different arithmetic and logical operations. The results of the simulations confirm the correctness of the ALU implementation by showing the expected outputs for various operations. The following figures illustrate the behavior of the ALU module under different test cases, with the input signals (**op1, op2, oprn**) and the resulting outputs (**result**). Each waveform visually presents how the ALU processes the input values for various operations such as **addition, subtraction, and bitwise AND**, showcasing its accurate functionality.

1. Addition: 15 + 3



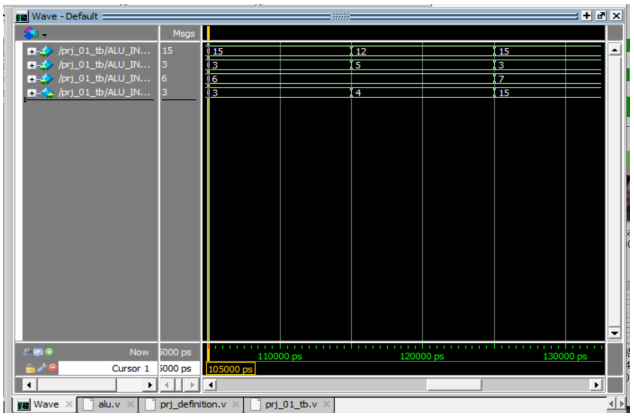
In the waveform, the input operands op1 is set to 15 and op2 is set to 3, with the operation code (oprn) set to 01, which corresponds to the addition operation. At 5530 ps, the result is displayed as 18, which is the correct output for the addition of 15 and 3. This confirms that the ALU correctly handles addition operations.

2. Subtraction: 15 - 5



In this waveform, the input operands are op1 = 15 and op2 = 5, with The operation code set to 02, indicating a subtraction operation. the result is shown as 10, verifying that the ALU correctly performs subtraction and produces the expected result.

3. Bitwise AND Operation: 15 & 3



For the bitwise AND operation, the inputs are op1 = 15 and op2 = 3, with the operation code set to 06 for the bitwise AND. In the waveform, the result is shown as 3, which is the correct outcome of the bitwise AND operation between 15 and 3, further confirming the ALU's correct functionality.

6. Conclusion

The ALU simulation successfully demonstrated its ability to perform various arithmetic and logical operations, including addition, subtraction, and bitwise operations, with accurate outputs. The test cases verified the correctness and efficiency of the ALU, proving it can handle inputs and deliver precise results within expected timing. The results show that the ALU is reliable and ready for integration into larger systems, making it a robust component for performing essential digital computations.