Arithmetic Logic Unit (ALU) Report

Bratin Mondal (21CS10016)
Somya Kumar (21CS30050)

October 4, 2023

1 Introduction

This report presents a comprehensive overview of the Arithmetic Logic Unit (ALU), a fundamental digital computing component. The ALU is responsible for executing arithmetic and logical operations and plays a pivotal role in computer architecture. This report discusses the ALU module's functionality, the testbench used for validation, and the application of the Xilinx Design Constraints (XDC) file for FPGA integration.

2 ALU Module Functionality

The ALU module is designed to perform a wide range of operations essential for digital computation. It operates on 32-bit data and offers support for both signed and unsigned numbers. The core functionality of the ALU module includes:

- **Arithmetic Operations:** The ALU performs addition and subtraction operations, crucial for numerical computation.
- **Logical Operations:** It executes logical AND, OR, XOR, and NOT operations, enabling data manipulation and decision-making.
- **Bit Shifting:** The module supports bit shifting operations, including shift left (arithmetic) and shift right (arithmetic and logical).
- **Modular Design:** The ALU is structured as a collection of sub-modules, enhancing maintainability and extensibility.

2.1 NOT32 Module

The NOT32 module performs a bitwise NOT operation on a 32-bit input.

2.2 AND32 Module

The AND32 module performs a bitwise AND operation on two 32-bit inputs.

2.3 SHIFT_RIGHT_ARITHMETIC Module

The SHIFT_RIGHT_ARITHMETIC module performs a 32-bit arithmetic right shift operation.

2.4 SHIFT_RIGHT_LOGICAL Module

The SHIFT_RIGHT_LOGICAL module performs a 32-bit logical right shift operation.

2.5 SHIFT_LEFT_ARITHMETIC Module

The SHIFT_LEFT_ARITHMETIC module performs a 32-bit arithmetic left shift operation.

2.6 OR32 Module

The OR32 module performs a bitwise OR operation on two 32-bit inputs.

2.7 XOR32 Module

The XOR32 module performs a bitwise XOR operation on two 32-bit inputs.

2.8 carry_look_ahead_4 Module

The carry_look_ahead_4 module implements a 4-bit carry look-ahead adder.

2.9 CLA_8 Module

The CLA_8 module implements an 8-bit carry look-ahead adder.

2.10 ADD32 Module

The ADD32 module performs a 32-bit addition using carry-look-ahead adders.

2.11 SUBTRACT32 Module

The SUBTRACT32 module performs a 32-bit subtraction using the ADD32 module and bitwise complement.

2.12 ALU Module

The ALU module implements an Arithmetic Logic Unit that can perform various arithmetic and logical operations based on the given opcode.

3 Testbench Functionality

The testbench, implemented in the "testbench.v" file, plays a critical role in verifying the ALU module's functionality. It conducts a comprehensive set of tests to validate different ALU operations and input scenarios. Key aspects of the testbench include:

- **Test Coverage:** The testbench covers a wide spectrum of operations, including addition, subtraction, logical operations, and bit shifting.
- **Input Generation:** It generates test inputs, configures ALU operation codes, sets operands, and specifies shift amounts for each test case.
- **Result Verification:** After executing an operation, the testbench validates the ALU's output against expected results to confirm correct operation.

3.1 Test Case 1: ADD Operation

```
opcode = 4'b0000; // Set operation code to ADD
A = 32'b00000000000000000000000000001010; // Set operand A
B = 32'b00000000000000000000000000000101; // Set operand B
shift_amount = 32'b00000000000000000000000000000000; // Set shift amount to 0
```

Expected Output:

```
Expected Result: 32'b00000000000000000000000000001111 (Binary)
Actual Result: 32'b00000000000000000000000000001111 (Binary)
```

The ADD operation test case passed.

3.2 Test Case 2: SUB Operation

```
opcode = 4'b0001; // Set operation code to SUB
A = 32'b00000000000000000000000000001010; // Set operand A
B = 32'b00000000000000000000000000000101; // Set operand B
shift_amount = 32'b00000000000000000000000000000000; // Set shift amount to 0
```

Expected Output:

```
Expected Result: 32'b00000000000000000000000000000101 (Binary)
Actual Result: 32'b00000000000000000000000000000101 (Binary)
```

The SUB operation test case passed.

3.3 Test Case 3: AND Operation

```
opcode = 4'b0010; // Set operation code to AND
A = 32'b11111111111111111111111111111111; // Set operand A
B = 32'b11111111000000000000000000000000; // Set operand B
shift_amount = 32'b00000000000000000000000000000000; // Set shift amount to 0
```

Expected Output:

```
Expected Result: 32'b00000011000000000000000000000000 (Binary)
Actual Result: 32'b00000011000000000000000000000000 (Binary)
```

The AND operation test case passed.

3.4 Test Case 4: OR Operation

```
opcode = 4'b0011; // Set operation code to OR
A = 32'sd-11; // Set operand A
B = 32'sd12; // Set operand B
shift_amount = 32'b00000000000000000000000000000000; // Set shift amount to 0
```

Expected Output:

```
Expected Result: 32'sd-3; // -3 in decimal
Actual Result: 32'sd-3; // -3 in decimal
```

The OR operation test case passed.

3.5 Test Case 5: XOR Operation

```
opcode = 4'b0100; // Set operation code to XOR
A = 32'b00000000000000000000000000001100; // Set operand A
B = 32'b00000000000000000000000000001111; // Set operand B
shift_amount = 32'b00000000000000000000000000000000; // Set shift amount to 0
```

Expected Output:

```
Expected Result: 32'b00000000000000000000000000000011 (Binary)
Actual Result: 32'b00000000000000000000000000000011 (Binary)
```

The XOR operation test case passed.

3.6 Test Case 6: NOT Operation

```
opcode = 4'b0101; // Set operation code to NOT
A = 32'b11111111111111111111111111111111; // Set operand A
B = 32'b00000000000000000000000000000000; // Set operand B
shift_amount = 32'b00000000000000000000000000000000; // Set shift amount to 0
```

Expected Output:

```
Expected Result: 32'sd-65536; // -65536 in decimal
Actual Result: 32'sd-65536; // -65536 in decimal
```

The NOT operation test case passed.

3.7 Test Case 7: SLA Operation

```
opcode = 4'b0110; // Set operation code to SLA
A = 32'b00000000000000000000000000001111; // Set operand A
B = 32'b00000000000000000000000000000000; // Set operand B
shift_amount = 32'b00000000000000000000000000000001; // Set shift amount to 1
```

Expected Output:

```
Expected Result: 32'b00000000000000000000000000011110 (Binary)
Actual Result: 32'b00000000000000000000000000011110 (Binary)
```

The SLA operation test case passed.

3.8 Test Case 8: SRA Operation

```
opcode = 4'b0111; // Set operation code to SRA
A = 32'sd-65536; // Set operand A (negative number)
B = 32'b00000000000000000000000000000000; // Set operand B
shift_amount = 32'b00000000000000000000000000000001; // Set shift amount to 1
```

Expected Output:

```
Expected Result: 32'sd2147450880; // 2147450880 in decimal
Actual Result: 32'sd2147450880; // 2147450880 in decimal
```

The SRA operation test case passed.

3.9 Test Case 9: SRL Operation

```
opcode = 4'b1000; // Set operation code to SRL
A = 32'sd-65536; // Set operand A (negative number)
B = 32'b00000000000000000000000000000000; // Set operand B
shift_amount = 32'b00000000000000000000000000000001; // Set shift amount to 1
```

Expected Output:

```
Expected Result: 32'sd-32768; // -32768 in decimal
Actual Result: 32'sd-32768; // -32768 in decimal
```

The SRL operation test case passed.

The testbench ensures that the ALU module functions reliably across diverse scenarios, bolstering its suitability for real-world applications.

4 Xilinx Design Constraints (XDC) File Usage

For deploying the ALU module within a Field-Programmable Gate Array (FPGA), the "master.xdc" Xilinx Design Constraints (XDC) file is provided. This file contains vital configuration settings and pin constraints necessary for FPGA integration:

- **Pin Mapping:** The XDC file specifies the mapping of module inputs and outputs to physical pins on the FPGA.
- **Clock Constraints:** It defines clock frequencies and synchronization requirements to ensure proper timing and performance.

- **FPGA-Specific Configurations:** The XDC file addresses voltage levels, I/O standards, and any FPGA-specific settings, ensuring compatibility and functionality.

4.1 Input Pins

The following input pins are connected to the ALU module:

- **clk**: This is the clock signal used to synchronize operations within the ALU module.
- **out**: This signal controls the direction of data flow and determines whether the module should output the upper 16 bits or the lower 16 bits of the result.
- **inp[0] to inp[7]**: These pins are connected to switches (sw[0] to sw[7]), and they provide 8-bit input data to the ALU module.
- **seg[0] and seg[1]**: These pins are connected to switches (sw[9] and sw[10]), and they determine which byte lane should be used for input data.
- **ab**: This pin is connected to a switch (sw[11]), and it determines whether the data should be written to operand 'a' or 'b' within the ALU module.
- **opcode[0] to opcode[3]**: These pins are connected to switches (sw[12] to sw[15]), and they represent the opcode for the ALU operation.
- **button**: This pin is connected to a button (btnu), and it triggers certain operations within the ALU module when pressed.

The XDC file is indispensable for guaranteeing that the ALU module functions correctly within the FPGA environment, meeting timing constraints and delivering dependable performance.

5 Conclusion

In summary, this report has provided a comprehensive overview of the Arithmetic Logic Unit (ALU) module, emphasizing its core functionality, the pivotal role of the testbench in validation, and the importance of the Xilinx Design Constraints (XDC) file for FPGA integration. The ALU module's versatile capabilities in executing arithmetic and logical operations make it a fundamental component of digital computing. Its successful implementation and rigorous testing validate its reliability for real-world applications.

This project enhances understanding of digital design principles, equipping individuals with the requisite skills to develop and validate intricate digital modules, seamlessly integrating them into FPGA-based hardware systems.