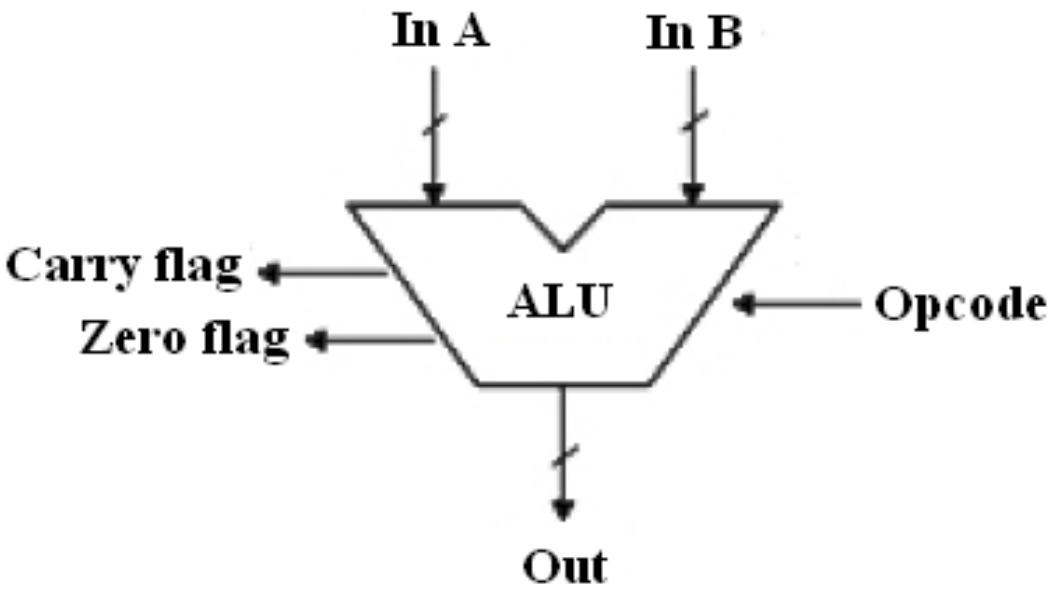
Arithmetic Logic Unit - ALU

Verification Plan



Prepared By: Jaber Darawsha

November 2025

Contents

List of Figures

Chapter 1

Introduction

1.1 Goals / Purpose of the Document

This document outlines a verification plan for an Arithmetic Logic Unit (ALU) designed to be part of a larger processor architecture. It aims to ensure that the ALU operates correctly in all scenarios, including standard operation, edge cases, and error handling.

1.2 Scope of Verification

The scope of verification for the ALU module will be limited to top-level verification, treating the ALU as a black box. We will focus on validating its inputs and outputs, such as Result and Error, without considering internal details. This approach ensures the ALU's functionality aligns with design specifications through external stimulus and response checks.

1.3 Assumptions / Constraints

1.3.1 Constraints

- No Constraints.

1.3.2 Assumptions

- The design is stable but not synthesizable.
- The design and verification environment uses UVM methodology.
- Functional coverage goals are achievable and measurable.
- All external dependencies, such as clocks and resets, are provided correctly.

Chapter 2

Design Overview

This section will cover the Desing Under Test (DUT), including its specifications, description, key features, and functionality, and will introduce the block diagram of the design and its interface.

2.1 Key features and functionality:

- **Addition**: The ALU performs binary addition of two input operands (A and B). This operation sums the values bit by bit, producing a result of 32-bit signal (Result).
- **Subtraction**: The ALU subtracts the second operand from the first by adding the first operand to the two's complement of the second operand. This results in the difference between the two values.
- **Bit wise logical AND operation**: The ALU performs a bitwise AND operation between two operands. Each bit of the output is 1 only if the corresponding bits of both operands are 1; otherwise, the result is 0.
- **Bit wise logical OR operation**: The ALU performs a bitwise OR operation between two operands. Each bit of the result is 1 if at least one of the corresponding bits in the operands is 1; otherwise, the result is 0.
- **Bit wise logical XOR operation**: The ALU performs a bitwise XOR (Exclusive OR) operation, where each bit of the result is 1 only if the corresponding bits of the operands are different (i.e., one is 1 and the other is 0); otherwise, the result is 0.
- **Error handling**: The ALU includes mechanisms to detect and respond to invalid operations or unexpected conditions.
- **Operation selection**: The ALU uses a control signal, called the "opcode," to select which arithmetic or logical operation to perform. The control unit decodes this signal and configures the ALU to execute the corresponding operation, ensuring the correct processing of data.

2.2 Block diagram and interface

This section presents the block diagram of the DUT and its interface. It highlights key components, their interactions, and the input/output signals, providing a clear view of the design's structure and how it communicates with external modules for proper verification.

The black box block diagram of the design is listed below in the figure.

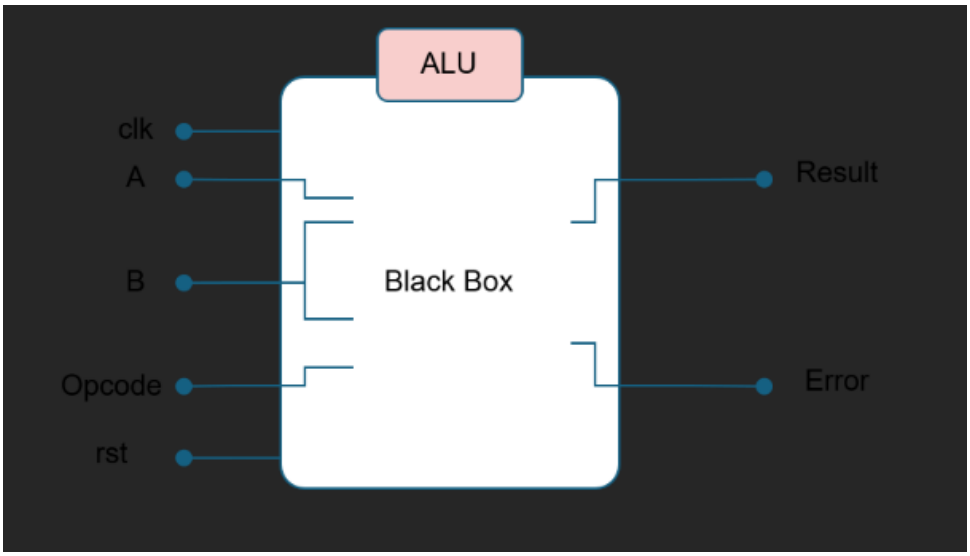


Figure 2.1: Black box block diagram of the ALU design.

The interface and signals details are listed below in the table

Table 2.1: interface

Signal	**Width**	**Direction**	**Description**
clk	1	input	Clock signal to the module
rst	1	input	Reset all internal registers of the design to its initial state
A	32	input	Represents the first operand of the arithmetic and logic
B	32	input	Represents the second operand of the arithmetic and logic operations
Opcode	3	input	A signal that selects the operation to be done on the two operands of the ALU
Result	32	output	Represents the outcome of the operation done on the input operands of the ALU
Error	1	output	A signal that indicates an overflow in the arithmetic operations and if an invalid operand sent to the ALU.

Chapter 3

Verification Strategy

3.1 Methodology

UVM will be used to create a scalable and reusable verification environment. The reference model will be written in SystemVerilog as a task included in the scoreboard, the ALU testbench will include drivers, monitors, scoreboards, a coverage collector (subscriber).

3.2 Selection of Verification Tests

Tests will cover all aspects of the ALU, including:

1. Addition Operation
    - Basic addition of positive numbers.
    - Addition of negative numbers.
    - Addition resulting in overflow (check overflow flag).
2. Subtraction Operation
    - Basic subtraction of positive numbers.
    - Subtraction of negative numbers.
    - Subtraction resulting in underflow (check underflow flag).
3. Bitwise AND Operation
    - AND operation with various bit patterns.
    - AND operation with all bits set to 1.
    - AND operation with all bits set to 0.
4. Bitwise OR Operation
    - OR operation with various bit patterns.
    - OR operation with all bits set to 1.
    - OR operation with all bits set to 0.

5. Bitwise XOR Operation
    - XOR operation with various bit patterns.
    - XOR operation with all bits set to 1.
    - XOR operation with all bits set to 0.
6. Unsupported Opcode
    - Test with an unsupported opcode to ensure the error flag is set.
7. Boundary Conditions
    - Operations with maximum and minimum values for A and B.
    - Operations with zero values for A and B.
8. Randomized Testing
    - Random combinations of A, B, and Opcode to ensure robustness.
9. Error Handling
    - Verify that the error flag is correctly set for overflow and underflow conditions.

3.3 Specific Corner Cases

1. Maximum and Minimum Integers for Addition and Subtraction
    - Adding the maximum positive integer to itself.
    - Subtracting the minimum negative integer from itself.
    - Adding a positive and a negative number that result in zero.
2. Bitwise Operations with Alternating Bit Patterns and Edge Cases
    - AND, OR, and XOR operations with 32'hAAAAAAAA and 32'h55555555.
    - AND, OR, and XOR operations with 32'hFFFFFFFF and 32'h00000000.
3. Overflow and Underflow Conditions
    - Addition that causes overflow: 32'h7FFFFFFF + 1.
    - Subtraction that causes underflow: 32'h80000000 - 1.
4. Zero and One Edge Cases
    - Operations where one operand is zero and the other is a non-zero value.
    - Operations where one operand is one and the other is a non-zero value.
5. Sign Bit Testing
    - Addition and subtraction where the sign bit changes (e.g., 32'h80000000 + 1).
    - Operations that toggle the sign bit.

6. Unsupported Opcode Handling
    - Test with opcodes outside the defined range (e.g., 3'b101, 3'b110, 3'b111).
7. Randomized Stress Testing
    - Randomly generated values for A, B, and Opcode to uncover unexpected issues.

Chapter 4

Testbench structure

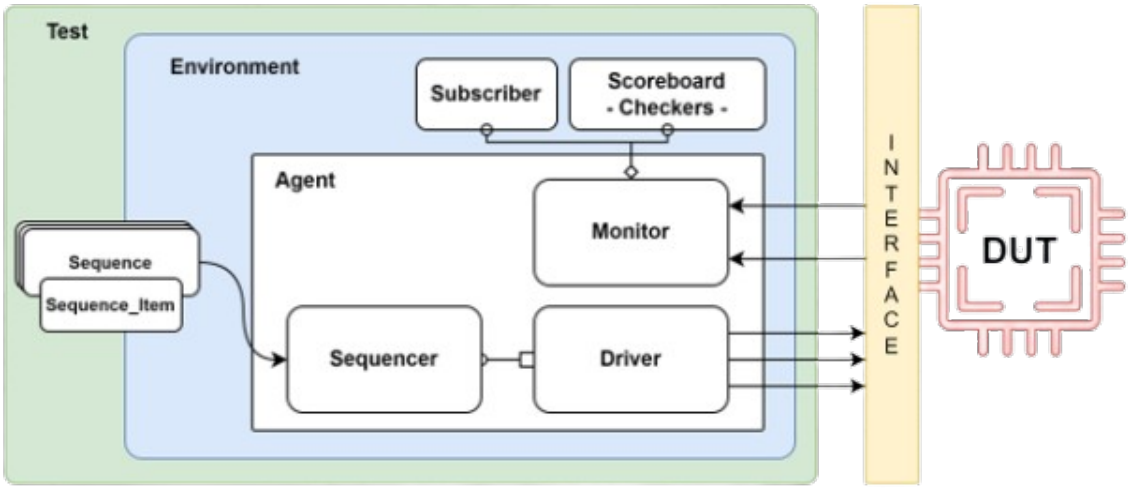


Figure 4.1: Testbench structure

Chapter 5

Tests

5.1 Coverage targets

- Code coverage
- Functional coverage
- Toggle coverage
- Cross coverage.
- Path coverage.

5.2 Testbench Architecture

- **Components**:
  - uvm_sequence_items
  - uvm_sequence
  - uvm_sequencer
  - uvm_driver
  - uvm_monitor
  - uvm_agent
  - uvm_scoreboard
  - uvm_subscriber
  - uvm_environment
  - uvm_test
  - Top
- **Interfaces**:Groups related signals together (like inputs, outputs, clocks, resets) to simplify driving and monitoring. Used by uvm_driver and uvm_monitor to send or observe signals from DUT.

5.3 Tools and technologies

- Visual Studio Code.
- MobaXterm - Linux
- Cadence Verisium.
- Integrated Metrics Center (IMC)

5.4 Regression Tests

Repeatedly run previous test cases to ensure that new changes do not break existing functionality.