The American University in Cairo

CENTER OF NANOELECTRONICS & DEVICES

# CND 212: Digital Testing and Verification

**Final Project Submission**

## "UVM Verification of ALU Design"

**Section: 18 & 16**
**Group: 8**

**Submitted by:**


| Student Name | ID |
|---|---|
| Mohamed Yasser Mohamed | V23010367 |
| Karim Emad El-Din | V23010546 |
| Zyad Ahmed Fawzy | V23010332 |
| Youssef Ahmed Hassan | V23009895 |
| Taha Zaki Mohamed | V23010462 |


**Submitted to TA: Eng. Rana Badran**

**May 2024**

# Acknowledgement

We would like to extend our deepest gratitude to Professor **Yehea Ismail** for his exceptional oversight and facilitation of this training program. His unwavering support and commitment to nurturing our skills in ASIC design have been invaluable.

Additionally, we would like to express our sincere thanks to **Eng. Rana Badran** for her invaluable assistance throughout the project. Her dedication and support in helping us overcome challenges were greatly appreciated.

We also want to thank all the **CND team members** for their contributions to this outstanding training program. Your collective efforts have provided us with a comprehensive and enriching learning experience. Thank you all for your dedication and support.

# Abstract

We are delighted to announce the successful completion of our project focused on the UVM (Universal Verification Methodology) verification of a simple ALU (Arithmetic Logic Unit) design. This project aimed to guide the trainee through the comprehensive verification flow, ultimately developing a functional and effective UVM testbench for the ALU IP.

Our journey began with understanding the ALU design specifications and creating a robust test plan to cover all possible operational scenarios. We meticulously developed a UVM environment, including the testbench components such as the UVM agent, driver, monitor, scoreboard, and sequences, to ensure thorough verification coverage.

We implemented a variety of tests, leveraging SystemVerilog features such as interfaces, classes, randomization, assertions, and functional coverage. This ensured that the ALU design met all functional and performance requirements. Rigorous simulation runs were conducted to validate the design under different conditions, identifying and resolving any issues encountered during the verification process.

Our diligent efforts culminated in the creation of a high-quality UVM testbench that can efficiently verify the functionality of the ALU IP. This project underscores our commitment to mastering verification methodologies and our expertise in applying UVM for IP verification, ultimately contributing to the reliability and robustness of digital designs.

.

# Table of Content

# Chapter 1: Introduction

## 1.1 Basics of ALU Design

An Arithmetic Logic Unit (ALU) is a critical component of any microprocessor, including the 8051 IP core. The ALU performs a variety of arithmetic and logical operations essential for the execution of instructions within the microprocessor. The 8051 ALU specifically is designed to handle operations defined by the 8051 IP core specification, providing functionality for addition, subtraction, multiplication, division, logical operations, and bit manipulations.

## 1.2 How ALU Works

The ALU operates based on input operands and control signals, producing an output based on the specified operation. The core of the ALU module involves several sub-modules and logic blocks that handle different arithmetic and logic functions. Each operation is defined by an operation code (op_code), which dictates the specific arithmetic or logical function the ALU needs to perform on the given operands.

## 1.3 Operations in ALU

The ALU supports a wide range of operations, including but not limited to:

- **Addition (OC8051_ALU_ADD)**: Computes the sum of two operands, considering the carry input.
- **Subtraction (OC8051_ALU_SUB)**: Computes the difference between two operands, considering the carry input.
- **Multiplication (OC8051_ALU_MUL)**: Multiplies two operands.
- **Division (OC8051_ALU_DIV)**: Divides one operand by another.
- **Decimal Adjust (OC8051_ALU_DA)**: Adjusts the result of a binary addition to a decimal value.
- **Logical NOT (OC8051_ALU_NOT)**: Performs bitwise NOT operation.
- **Logical AND (OC8051_ALU_AND)**: Performs bitwise AND operation.
- **Logical XOR (OC8051_ALU_XOR)**: Performs bitwise XOR operation.
- **Logical OR (OC8051_ALU_OR)**: Performs bitwise OR operation.
- **Rotate Left (OC8051_ALU_RL)**: Rotates bits of the operand to the left.
- **Rotate Left through Carry (OC8051_ALU_RLC)**: Rotates bits of the operand to the left through the carry bit.

- **Rotate Right (OC8051_ALU_RR)**: Rotates bits of the operand to the right.
- **Rotate Right through Carry (OC8051_ALU_RRC)**: Rotates bits of the operand to the right through the carry bit.
- **Increment (OC8051_ALU_INC)**: Increments the operand by one.
- **Exchange (OC8051_ALU_XCH)**: Exchanges bits between operands based on the carry bit.
- **No Operation (OC8051_ALU_NOP)**: No operation, outputs are same as inputs.

### 1.4 Project Overview

The objective of this project is to verify the functionality and correctness of the ALU module using Universal Verification Methodology (UVM). UVM is a standardized methodology for verifying integrated circuit designs. It provides a framework to create modular, reusable verification environments.

This project will take the trainee through the complete verification flow to generate a working UVM testbench for the ALU module of the 8051 core. The steps will include developing a test plan, writing test sequences, running simulations, and analyzing the results.

### 1.5 Project Tasks

1. **Understanding the ALU Design**: Detailed study of the ALU module to understand its operation and functionalities.
2. **Setting Up the UVM Environment**: Establishing a UVM-based verification environment including all necessary components like agents, drivers, monitors, and scoreboards.
3. **Writing Test Sequences**: Developing test sequences to verify different operations of the ALU.
4. **Running Simulations**: Executing the test sequences in a simulation environment to verify the correctness of the ALU.
5. **Analyzing Results**: Analyzing simulation results to ensure the ALU performs as expected and meets the design specifications.
6. **Generating Reports**: Documenting the verification process, results, and coverage analysis. This project will provide hands-on experience with UVM and demonstrate the importance of thorough verification in the design process of a microprocessor component.

# Chapter 2 : UVM Overview

## 2.1 Introduction to UVM

The Universal Verification Methodology (UVM) is a standardized methodology for verifying integrated circuits. Developed by Accellera, UVM provides a comprehensive and flexible verification framework based on SystemVerilog. The methodology includes a rich set of features for creating reusable verification environments, allowing for more efficient and reliable verification processes. By using UVM, verification engineers can develop testbenches that are modular, reusable, and scalable, significantly reducing the time and effort required to verify complex digital designs.

## 2.2 UVM Components

UVM components are the building blocks of a UVM testbench. Each component serves a specific role in the verification environment, facilitating communication and synchronization between different parts of the testbench. Here are the key UVM components used in the provided verification environment:

- **Driver**: This component is responsible for driving the signals to the Design Under Test (DUT). It takes transactions from the sequencer and translates them into pin-level activities.
- **Monitor**: The monitor observes the signals from the DUT and converts them into transactions. These transactions can then be used for checking or further processing.
- **Sequencer**: This component controls the flow of sequences to the driver. It determines which sequence should be executed and when.
- **Agent**: An agent encapsulates a driver, monitor, and sequencer. It provides a complete verification agent that can be easily instantiated and configured.
- **Environment**: The environment contains multiple agents and other components necessary for the verification of the DUT. It provides a top-level encapsulation of the entire verification setup.
- **Scoreboard**: The scoreboard is used to check the correctness of the DUT's output. It compares the actual output with the expected output.
- **Subscriber**: This component listens to transactions from monitors and processes them, typically for checking or coverage collection.

### 2.3 UVM Testbench Architecture

The UVM testbench architecture is hierarchical and modular, allowing for easy integration and reuse of components. The key elements of the architecture in the provided example are:

- **Top Module**: This is the highest level of the testbench where the DUT and interface are instantiated. It also initializes the UVM environment and starts the test.
- **Interface**: The interface connects the testbench to the DUT. It encapsulates the signals and provides a mechanism for clocking and resetting the DUT.
- **Test Class**: The test class is derived from **uvm_test** and is responsible for configuring the environment and starting the required sequences.
- **Environment Class**: The environment class, typically derived from **uvm_env**, contains instances of agents and other verification components. It sets up the overall verification environment.
- **Agent Class**: The agent class encapsulates the driver, monitor, and sequencer. It coordinates their activities to perform verification tasks.
- **Driver Class**: The driver class, derived from **uvm_driver**, sends stimulus to the DUT based on transactions received from the sequencer.
- **Monitor Class**: The monitor class, derived from **uvm_monitor**, captures the DUT's output and converts it into transactions.
- **Sequencer Class**: The sequencer class, derived from **uvm_sequencer**, controls the order and timing of transactions sent to the driver.
- **Subscriber Class**: The subscriber collects coverage information based on monitored transactions. It analyzes transaction data and records coverage metrics specified by covergroups.
- **Scoreboard Class**: The scoreboard checks the correctness of the DUT's output by comparing it with the expected results.

### 2.4 Benefits of Using UVM

Using UVM in verification offers several advantages:

- **Reusability**: UVM promotes the development of reusable verification components, reducing the time and effort required for subsequent projects.
- **Modularity**: UVM's hierarchical architecture allows for modular testbenches, making them easier to maintain and extend.

- **Scalability**: UVM can handle the complexity of large designs, scaling from simple modules to complete systems.
- **Standardization**: As a standardized methodology, UVM ensures consistency and compatibility across different projects and teams.
- **Automation**: UVM includes features for automating many verification tasks, such as stimulus generation, checking, and coverage collection, increasing productivity and reducing manual errors.

The provided UVM testbench for the ALU (Arithmetic Logic Unit) illustrates these benefits. It leverages UVM components to create a robust and flexible verification environment, ensuring thorough validation of the ALU functionality. The modular nature of the testbench allows for easy updates and extensions as the design evolves.

# Chapter 3 : ALU Design

## 3.1 Test Plan Overview

The Test Plan is a comprehensive document that outlines the strategies, methodologies, and components utilized to verify the functionality and correctness of the Design Under Test (DUT). In this chapter, we provide an overview of the Test Plan, focusing on key components such as test cases, scenarios, functional coverage, and assertions.

## 3.2 Test Cases and Scenarios

Test Cases and Scenarios are crucial elements of the verification process, ensuring that the DUT's functionality is thoroughly exercised under various conditions.

### my_base_sequence Class

The **my_base_sequence** class defines a set of test cases and scenarios for verifying the DUT. Each sequence within **my_base_sequence** corresponds to a specific test case, covering different operations and scenarios. These sequences include:

- **Basic Operations**: Test cases for basic arithmetic and logical operations such as addition, subtraction, multiplication, division, logical AND, logical OR, logical XOR, logical NOT, rotate left, rotate right, rotate left with carry, rotate right with carry, decimal adjustment, and exchange.
- **Special Cases**: Test cases for special operations such as operation PCS add and exchange.

## 3.3 Functional Coverage

Functional coverage is critical for ensuring that all functional aspects of the DUT are adequately exercised during verification.

### my_subscriber Class

The **my_subscriber** class implements functional coverage tracking using covergroups. Each covergroup within **my_subscriber** captures specific aspects of the DUT's behavior, including operand values, carry flags, auxiliary carry flags, input bits, overflow flags, and more.

## 3.4 Assertions

Assertions play a crucial role in the verification process, providing real-time checks to validate the correctness of the DUT's behavior.

### Assertions Interface

The assertions interface defines a set of properties and assertions to validate the behavior of the DUT during simulation. These assertions cover various aspects of the DUT's operation, including arithmetic operations, logical operations, flag status, and special operations. Each assertion monitors specific signals and conditions within the DUT and triggers an error message if any discrepancy is detected.

# Chapter 4 : UVM Environment Setup

## 4.1 Environment Components

In a UVM-based verification environment, several components work together to enable comprehensive verification. These components include agents, drivers, monitors, scoreboards, coverage collectors, sequencers, and sequences. Each component serves a specific purpose in the verification process, contributing to the effectiveness and efficiency of the overall environment.

## 4.2 Agent, Driver, and Monitor

The agent acts as an intermediary between the testbench and the design-under-test (DUT). It coordinates the transfer of data between these two entities and manages the flow of transactions. Within the agent, the driver is responsible for driving stimulus to the DUT, while the monitor observes the DUT's behavior by capturing its outputs and transmitting them back to the testbench for analysis. The sequencer within the agent controls the generation and sequencing of transactions, directing sequences to be executed by the driver.

## 4.3 Scoreboard and Coverage Collector

The scoreboard is a critical component for verifying the correctness of the DUT's behavior. It compares the actual outputs produced by the DUT with expected values to detect any discrepancies or errors. The coverage collector is responsible for collecting coverage data during simulation. It monitors various aspects of the design's behavior and tracks which parts of the design have been exercised by the testbench.

## 4.4 Sequencer and Sequences

The sequencer plays a central role in controlling the flow of transactions within the agent. It receives requests from sequences and forwards them to the driver for execution. Sequences, on the other hand, define specific test scenarios and stimuli to be applied to the DUT. They encapsulate the sequence of transactions required to exercise particular features or functionalities of the design and are responsible for generating meaningful test stimuli.

## 4.5 Connecting UVM Components

Once the UVM components are instantiated, they need to be interconnected to establish the verification environment fully. This typically involves connecting the driver, monitor, sequencer, scoreboard, and coverage collector within the agent and ensuring that sequences are properly directed to the sequencer for execution. Proper interconnection ensures that data flows seamlessly between the testbench and the DUT and facilitates comprehensive verification of the design.

# Chapter 5 : Writing Test Sequences

## 5.1 Sequence Structure

Test sequences in UVM are structured sequences of transactions or events that drive stimuli to the design-under-test (DUT) to verify its functionality. They are typically organized into classes that extend the uvm_sequence base class. Sequences define specific test scenarios and encapsulate the sequence of operations or transactions required to verify particular features or behaviors of the DUT.

## 5.2 Basic Test Sequences

Basic test sequences in UVM are straightforward sequences that execute a predefined sequence of transactions or events. In the provided code snippet, the my_base_sequence class extends uvm_sequence and defines a sequence of transactions involving a my_sequence_item. These sequences are often used to verify simple functionalities or to establish a baseline for more complex test scenarios.

## 5.3 Randomized Sequences

Randomized sequences in UVM leverage the built-in randomization features of SystemVerilog to generate stimulus for the DUT dynamically. By using randomization, test sequences can provide a diverse set of stimuli that thoroughly exercise the design and uncover corner-case scenarios. In the body() task of my_base_sequence, randomization is applied to the First_seq item to generate different scenarios for testing.

## 5.4 Directed Test Cases

Directed test cases involve creating sequences that target specific functionalities or corner cases within the DUT. Unlike randomized sequences, directed test cases are explicitly defined to verify particular behaviors or scenarios. In the provided code snippet, specific test cases are created by setting values for op_code, srcCy, and src1 fields of the First_seq item to cover different scenarios and corner cases.

## 5.5 Sequence Library

A sequence library is a collection of reusable test sequences that cover a wide range of test scenarios and functionalities. By maintaining a sequence library, verification teams can leverage existing sequences to accelerate test development and ensure consistency across different projects. The my_base_sequence class serves as an example of a sequence that can be part of a sequence library, providing test scenarios for the DUT.

# Chapter 6 : Results and Analysis

In this chapter, we present the results of our simulations, provide a detailed coverage report, analyze the outcomes, and identify areas for improvement. The analysis is supported by various screenshots capturing different stages and aspects of the simulation process.

## 6.1 Simulation Results

The simulation results are crucial in verifying the functionality and performance of the system under test. The following screenshots illustrate the different phases and outputs of the simulation process.

```
Back to file 'my_top.sv'.
Parsing included file '/home/vlsi/VerificationFinalProject/NEW_UVM_Files/my_subscriber.sv'.
Back to file 'my_top.sv'.
Parsing included file '/home/vlsi/VerificationFinalProject/NEW_UVM_Files/my_enviroment.sv'.
Back to file 'my_top.sv'.
Parsing included file '/home/vlsi/VerificationFinalProject/NEW_UVM_Files/my_test.sv'.
Back to file 'my_top.sv'.
Top Level Modules:
       top
No TimeScale specified
Starting vcs inline pass...

6 modules and 0 UDP read.
       However, due to incremental compilation, no re-compilation is necessary.
rm -f _cuarc*.so _csrc*.so pre_vcsobj_*.so share_vcsobj_*.so
ld -shared  -Bsymbolic   -o .//../simv.daidir//_cuarc0.so objs/amcQw_d.o
rm -f _cuarc0.so
if [ -x ../simv ]; then chmod a-x ../simv; fi
g++  -o ../simv      -rdynamic  -Wl,-rpath='$ORIGIN'/simv.daidir -Wl,-rpath=./simv.daidir -Wl,-rpath=/eda/Synopsys/vcs/R-2020.12-SP1/linux64/lib -L/eda/Synopsys/vcs/R-2020.12-SP1/linux64/lib  -Wl,-rpath-link=./
 /usr/lib64/libnuma.so.1 /eda/Synopsys/vcs/R-2020.12-SP1/linux64/lib/vpdlogstub.o uvm_dpi.o     _19961_archive_1.so  _prev_archive_1.so  _cuarc0.so  SIM_l.o      rmapats_mop.o rmapats.o rmar.o rmar_nd.o  rmar_llvm
_0_1.o rmar_llvm_0_0.o          -lvirsim -lerrorinf -lsnpsmalloc -lvfs    -lvcsnew -lsimprofile -luclinative /eda/Synopsys/vcs/R-2020.12-SP1/linux64/lib/vcs_tls.o   -Wl,-whole-archive  -lvcsucli    -Wl,-no-whol
e-archive      ./../simv.daidir/vc_hdrs.o    /eda/Synopsys/vcs/R-2020.12-SP1/linux64/lib/vcs_save_restore_new.o -ldl  -lc -lm -lpthread -ldl
../simv up to date
CPU time: 10.339 seconds to compile + 6.024 seconds to elab + .651 seconds to link
```

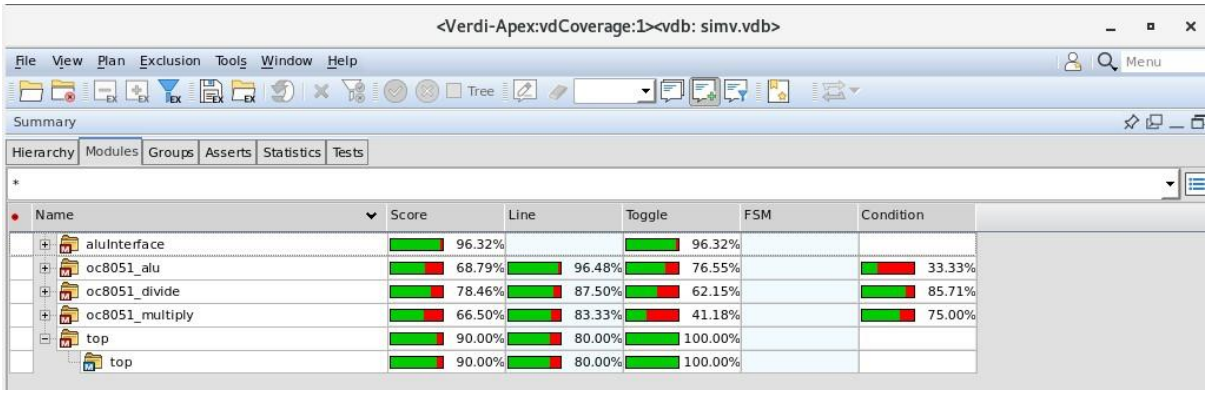
The VCS compile stage is shown in the screenshot below. This step involves compiling the source code and testbenches using the VCS compiler. The successful completion of this phase ensures that the code is syntactically correct and ready for simulation.

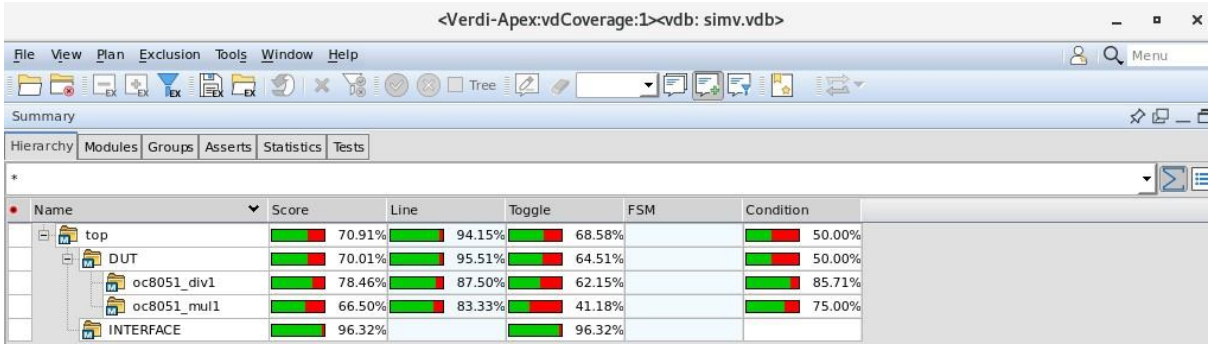
```
**********       IMPORTANT RELEASE NOTES         **********

You are using a version of the UVM library that has been compiled
with `UVM_NO_DEPRECATED undefined.
See http://www.eda.org/svdb/view.php?id=3313 for more details.

You are using a version of the UVM library that has been compiled
with `UVM_OBJECT_DO_NOT_NEED_CONSTRUCTOR undefined.
See http://www.eda.org/svdb/view.php?id=3770 for more details.

    (Specify +UVM_NO_RELNOTES to turn off this notice)

UVM_INFO /home/vlsi/VerificationFinalProject/NEW_UVM_Files/my_test.sv(9) @ 0: uvm_test_top [TEST_CLASS] Inside Constructor!
UVM_INFO @ 0: reporter [RNTST] Running test my_test...
UVM_INFO /home/vlsi/VerificationFinalProject/NEW_UVM_Files/my_test.sv(14) @ 0: uvm_test_top [TEST_CLASS] Build Phase!
UVM_INFO /home/vlsi/VerificationFinalProject/NEW_UVM_Files/my_enviroment.sv(9) @ 0: uvm_test_top.env [ENV_CLASS] Inside Constructor!
UVM_INFO /home/vlsi/VerificationFinalProject/NEW_UVM_Files/my_sequence.sv(10) @ 0: reporter@@reset_seq [BASE_SEQ] Inside Constructor!
UVM_INFO /home/vlsi/VerificationFinalProject/NEW_UVM_Files/my_enviroment.sv(14) @ 0: uvm_test_top.env [ENV_CLASS] Build Phase!
UVM_INFO /home/vlsi/VerificationFinalProject/NEW_UVM_Files/my_agent.sv(12) @ 0: uvm_test_top.env.agnt [AGENT_CLASS] Inside Constructor!
UVM_INFO /home/vlsi/VerificationFinalProject/NEW_UVM_Files/my_scoreboard.sv(20) @ 0: uvm_test_top.env.scb [SCB_CLASS] Inside Constructor!
UVM_INFO /home/vlsi/VerificationFinalProject/NEW_UVM_Files/my_agent.sv(17) @ 0: uvm_test_top.env.agnt [AGENT_CLASS] Build Phase!
UVM_INFO /home/vlsi/VerificationFinalProject/NEW_UVM_Files/my_driver.sv(10) @ 0: uvm_test_top.env.agnt.drv [DRIVER_CLASS] Inside Constructor!
UVM_INFO /home/vlsi/VerificationFinalProject/NEW_UVM_Files/my_monitor.sv(12) @ 0: uvm_test_top.env.agnt.mon [MONITOR_CLASS] Inside Constructor!
UVM_INFO /home/vlsi/VerificationFinalProject/NEW_UVM_Files/my_sequencer.sv(6) @ 0: uvm_test_top.env.agnt.sqr [SEQUENCER_CLASS] Inside Constructor!
UVM_INFO /home/vlsi/VerificationFinalProject/NEW_UVM_Files/my_driver.sv(15) @ 0: uvm_test_top.env.agnt.drv [DRIVER_CLASS] Build Phase!
UVM_INFO /home/vlsi/VerificationFinalProject/NEW_UVM_Files/my_sequence_item.sv(6) @ 0: reporter@@my_sequence_item [SEQUENCE_ITEM_CLASS] Inside Constructor!
UVM_INFO /home/vlsi/VerificationFinalProject/NEW_UVM_Files/my_monitor.sv(17) @ 0: uvm_test_top.env.agnt.mon [MONITOR_CLASS] Build Phase!
UVM_INFO /home/vlsi/VerificationFinalProject/NEW_UVM_Files/my_sequence_item.sv(6) @ 0: reporter@@item [SEQUENCE_ITEM_CLASS] Inside Constructor!
UVM_INFO /home/vlsi/VerificationFinalProject/NEW_UVM_Files/my_sequencer.sv(13) @ 0: uvm_test_top.env.agnt.sqr [SEQUENCER_CLASS] Build Phase!
UVM_INFO /home/vlsi/VerificationFinalProject/NEW_UVM_Files/my_sequence_item.sv(6) @ 0: reporter@@item [SEQUENCE_ITEM_CLASS] Inside Constructor!
UVM_INFO /home/vlsi/VerificationFinalProject/NEW_UVM_Files/my_scoreboard.sv(25) @ 0: uvm_test_top.env.scb [SCB_CLASS] Build Phase!
UVM_INFO /home/vlsi/VerificationFinalProject/NEW_UVM_Files/my_sequence_item.sv(6) @ 0: reporter@@First_Seq [SEQUENCE_ITEM_CLASS] Inside Constructor!
UVM_INFO /home/vlsi/VerificationFinalProject/NEW_UVM_Files/my_driver.sv(24) @ 0: uvm_test_top.env.agnt.drv [DRIVER_CLASS] Connect Phase!
UVM_INFO /home/vlsi/VerificationFinalProject/NEW_UVM_Files/my_monitor.sv(27) @ 0: uvm_test_top.env.agnt.mon [MONITOR_CLASS] Connect Phase!
UVM_INFO /home/vlsi/VerificationFinalProject/NEW_UVM_Files/my_sequencer.sv(19) @ 0: uvm_test_top.env.agnt.sqr [SEQUENCER_CLASS] Connect Phase!
UVM_INFO /home/vlsi/VerificationFinalProject/NEW_UVM_Files/my_agent.sv(33) @ 0: uvm_test_top.env.agnt [AGENT_CLASS] Connect Phase!
UVM_INFO /home/vlsi/VerificationFinalProject/NEW_UVM_Files/my_scoreboard.sv(33) @ 0: uvm_test_top.env.scb [SCB_CLASS] Connect Phase!
UVM_INFO /home/vlsi/VerificationFinalProject/NEW_UVM_Files/my_enviroment.sv(21) @ 0: uvm_test_top.env [ENV_CLASS] Connect Phase!
UVM_INFO /home/vlsi/VerificationFinalProject/NEW_UVM_Files/my_test.sv(21) @ 0: uvm_test_top [TEST_CLASS] Connect Phase!
UVM_INFO /home/vlsi/VerificationFinalProject/NEW_UVM_Files/my_driver.sv(29) @ 0: uvm_test_top.env.agnt.drv [DRIVER_CLASS] Run Phase!
UVM_INFO /home/vlsi/VerificationFinalProject/NEW_UVM_Files/my_monitor.sv(32) @ 0: uvm_test_top.env.agnt.mon [MONITOR_CLASS] Inside Run Phase!
UVM_INFO /home/vlsi/VerificationFinalProject/NEW_UVM_Files/my_scoreboard.sv(39) @ 0: uvm_test_top.env.scb [SCB_CLASS] Run Phase!
UVM_INFO /home/vlsi/VerificationFinalProject/NEW_UVM_Files/my_test.sv(26) @ 0: uvm_test_top [TEST_CLASS] Run Phase!
UVM_INFO /home/vlsi/VerificationFinalProject/NEW_UVM_Files/my_sequence_item.sv(6) @ 0: reporter@@First_seq [SEQUENCE_ITEM_CLASS] Inside Constructor!
```

The build phase for each component is captured in the next screenshot. During this phase, each module is individually built and verified to ensure that all components are correctly implemented and integrated.

## 6.2 Coverage Report

Coverage analysis is a critical aspect of the verification process. It provides insights into how thoroughly the testbench exercises the design under test (DUT). The coverage report screenshot below illustrates the coverage metrics achieved during the simulation.






## 6.3 Analysis of Results

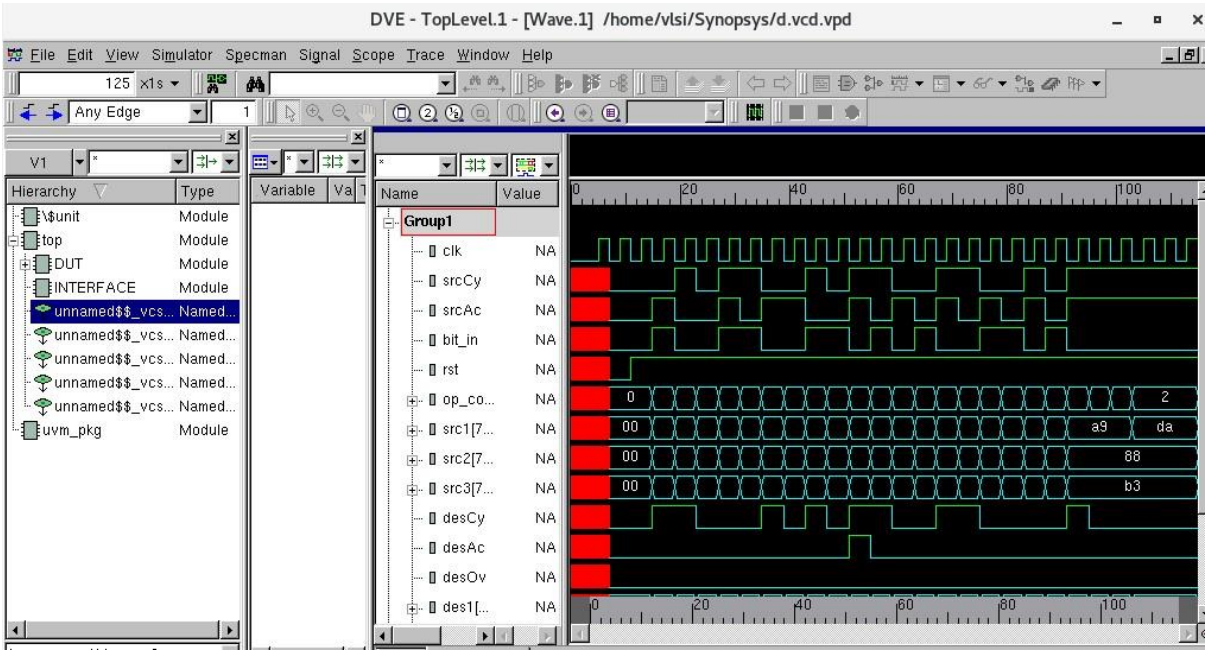
The analysis of the results includes a detailed examination of the outputs and logs generated during the simulation. This section highlights key findings and observations based on the simulation data.

```
------------------------------------------
[18446744073709551615] Not Logic Operation Pass
------------------------------------------
[3] operation decimal adjustment Pass
------------------------------------------
[7] Operation pcs Add Pass
------------------------------------------
[11] operation exchange Pass
------------------------------------------
[15] Division Pass
------------------------------------------
[19] Rotate Right with carry Logic Operation Pass
------------------------------------------
[23] AND Logic Operation Pass
------------------------------------------
[27] OR Logic Operation Pass
------------------------------------------
[31] Addition Pass
------------------------------------------
[35] Subtraction Pass
------------------------------------------
[43] Multiplication Pass
------------------------------------------
[47] Rotate Left with carry Logic Operation Pass
------------------------------------------
[51] XOR Logic Operation Pass
------------------------------------------
[55] Rotate Right Logic Operation Pass
------------------------------------------
[59] Rotate Left Logic Operation Pass
------------------------------------------
[63] AND Logic Operation Pass
------------------------------------------
[67] operation exchange Pass
------------------------------------------
[71] Rotate Left Logic Operation Pass
------------------------------------------
[75] XOR Logic Operation Pass
------------------------------------------
[79] Subtraction Pass
------------------------------------------
[83] Operation pcs Add Pass
------------------------------------------
[87] Subtraction Pass
------------------------------------------
[91] Subtraction Pass
------------------------------------------
[95] Subtraction Pass
```

The scoreboard report, shown in the following screenshot, provides a summary of the verification results, including pass/fail statistics for various test cases. This report is essential for assessing the overall health and reliability of the system.

```
** Report counts by severity
UVM_INFO :   34
UVM_WARNING :    0
UVM_ERROR :    0
UVM_FATAL :    0
** Report counts by id
[AGENT_CLASS]     3
[BASE_SEQ]     1
[DRIVER_CLASS]     4
[ENV_CLASS]     3
[MONITOR_CLASS]     4
[RNTST]     1
[SCB_CLASS]     4
[SEQUENCER_CLASS]     3
[SEQUENCE_ITEM_CLASS]     5
[TEST_CLASS]     4
[TEST_DONE]     1
[UVM/RELNOTES]     1

$finish called from file "/eda/Synopsys/vcs/R-2020.12-SP1/etc/uvm-1.2/base/uvm_root.svh", line 527.
$finish at simulation time                116
           V C S   S i m u l a t i o n   R e p o r t
Time: 116
CPU Time:      1.660 seconds;       Data structure size:   0.4Mb
Sat May 18 00:03:14 2024
```

Universal Verification Methodology (UVM) information is crucial for understanding the testbench structure and execution flow. The screenshot below captures the UVM info, which includes details about the UVM environment, components, and transactions.




The waveform screenshot provides a visual representation of the signals and their interactions over time. This graphical view helps in identifying timing issues, signal

integrity problems, and verifying that the system behaves as expected under different test scenarios.

### 6.4 Identifying Gaps and Improvements

Based on the simulation results and analysis, we identify gaps and suggest improvements to enhance the verification process and the system design. This section outlines specific areas where the coverage was insufficient, potential sources of errors, and recommendations for future work.

By addressing the gaps identified and implementing the suggested improvements, we can achieve higher confidence in the system's correctness and robustness. This iterative process of simulation, analysis, and enhancement is key to successful verification and validation.

# Conclusion

The journey of verifying the ALU design using UVM methodology has been a comprehensive and meticulous process, highlighting the expertise and dedication of the verification team. Each phase of this journey is critical in ensuring the ALU operates correctly and meets all design specifications.

The process begins with the strategic development of the UVM testbench architecture. This foundational step involves defining the essential components, sequences, and configuration settings that create a robust and scalable verification environment. The UVM framework allows for systematic and efficient testing of the ALU's functionality.

Implementing the UVM testbench includes developing the driver, monitor, scoreboard, and other vital components. These elements work in unison to generate stimulus, capture responses, and compare actual outputs against expected results, thus ensuring the ALU performs as intended under various scenarios.

Functional coverage analysis is conducted to evaluate the extent of testbench effectiveness in exercising the design. By defining and tracking coverage metrics, the verification team identifies untested scenarios and enhances the testbench to achieve comprehensive coverage, ensuring all functionality is verified.

Running simulations represents a critical phase where the ALU design is subjected to a wide range of test cases to validate its behavior. Utilizing simulators like VCS, the team executes numerous tests, collects data, and analyzes results to detect any discrepancies or errors in the design.

Debugging and resolving issues is an iterative and crucial process. Identified bugs are analyzed and fixed, refining the design to meet the required specifications and performance targets.

Formal verification techniques are employed to rigorously prove the correctness of the ALU design. These formal methods ensure that the design adheres to its specifications under all possible conditions, providing a higher level of confidence in its reliability.

Generating and analyzing waveforms is another essential step in the verification process. Waveform analysis helps visualize signal interactions over time, aiding in the identification of timing issues, signal integrity problems, and verifying that the design behaves correctly throughout its operation.

Finally, the coverage report provides a comprehensive summary of the verification process. It highlights the achieved coverage metrics, identifies any remaining gaps, and suggests improvements for future verification efforts.

In conclusion, the UVM Verification of the ALU Design is a testament to the collaborative efforts, technical proficiency, and unwavering dedication of the verification team. From testbench development to coverage analysis, each step contributes to ensuring the ALU design is reliable, robust, and ready for integration into larger systems. This journey underscores the importance of thorough verification in the design process, ensuring the final product meets the highest standards of quality and performance.