International Journal of Scientific Engineering and Research (IJSER)
ISSN (Online): 2347-3878
Impact Factor (2020): 6.733

Functional Verification of LC3 Microcontroller (UVM, System Verilog)

Sai Madhav Modepalli

National Institute of Technology Karnataka Surathkal, Karnataka, India
Email: *madhavmodepalli77[at]gmail.com*

Abstract: ***The continuous growth of the semiconductor industry has led to the development of advanced microcontrollers that power various electronic systems. Functional verification is a crucial step in the design and development process to ensure the correct operation of these microcontrollers. This research paper focuses on the functional verification of the LC3 (Little Computer 3) microcontroller using the Universal Verification Methodology (UVM) and SystemVerilog. The objective is to validate the functionality of the LC3 microcontroller design by employing industry-standard verification techniques and methodologies. The paper presents the architecture of the LC3 microcontroller, explains the UVM methodology, and demonstrates the application of SystemVerilog for functional verification. Additionally, various verification components and strategies are discussed to ensure comprehensive and effective verification of the LC3 microcontroller.***

Keywords: LC3 microcontroller, System Verilog, UVM, VLSI

1. Introduction

The proposed research paper aims to provide a comprehensive guide to functional verification of the LC3 microcontroller using UVM and SystemVerilog. The paper will discuss the LC3 microcontroller's architecture, instruction set, and memory organization. It will also present an in-depth explanation of the UVM methodology and its application in the functional verification process. Furthermore, the paper will delve into the key features of System Verilog and its usage in building effective testbenches.

2. LC3 Microcontroller

2.1 Artitectute

The LC3 (Little Computer 3) microcontroller is a simplified, educational microcontroller architecture that is widely used in academic settings to teach computer architecture and assembly language programming concepts. It was designed by Yale N. Patt and Sanjay J. Patel at the University of Texas at Austin and has gained popularity due to its simplicity and educational value.

The LC3 microcontroller follows a Von Neumann architecture, which means that both the program memory and data memory are stored in the same address space. The architecture consists of several key components, each playing a specific role in the microcontroller's operation. Let's explore these components in detail:

2.1.1 Program Counter (PC):
The Program Counter is a register that holds the memory address of the current instruction being executed. It is automatically incremented after each instruction fetch, pointing to the next instruction in memory.

2.1.2 Instruction Register (IR):
The Instruction Register is a register that holds the current instruction fetched from memory. It provides the necessary information to decode and execute the instruction.

2.1.3 Arithmetic Logic Unit (ALU):
The ALU performs arithmetic and logical operations on data. It supports basic operations such as addition, subtraction, logical AND, logical OR, etc.

2.1.4 General-Purpose Registers:
The LC3 architecture provides eight general-purpose registers (R0 to R7) that can be used to store data or intermediate results during program execution. These registers are primarily used for computation and data manipulation.

2.1.5 Condition Code Register (CC):
The Condition Code Register holds information about the result of the most recent arithmetic or logical operation. It contains four condition code flags: Negative (N), Zero (Z), Positive (P), and Overflow (V). These flags are used to determine the outcome of conditional branch instructions.

2.2 Memory

The LC3 microcontroller has a 16-bit address space, allowing it to address up to 64 kilobytes of memory. It uses a flat memory model, where both instructions and data are stored in the same address space.

2.3 Control Unit

The Control Unit coordinates the execution of instructions by generating control signals that activate various components of the microcontroller. It controls the flow of data between registers, memory, and the ALU based on the instructions being executed.

2.4 Input / Output (I/O)

The LC3 microcontroller supports simple I/O operations. It provides a set of memory-mapped I/O locations that can be used to interact with external devices such as keyboards, displays, or other peripherals.

The LC3 microcontroller uses a simple and orthogonal instruction set architecture, meaning that instructions are easy to decode and execute. The instruction set includes operations such as data movement, arithmetic and logic operations, control transfer (branching), and I/O operations. Each instruction is encoded as a 16-bit value, with specific fields for the opcode, source and destination registers, immediate values, and addressing modes.

The LC3 architecture's simplicity makes it an ideal platform for learning computer architecture concepts, assembly language programming, and low-level system design. It provides a solid foundation for understanding more complex microcontroller architectures and their associated concepts.

3. Functional Verification

3.1 Importance and Challenges

Functional verification is a crucial step in the design and development process of microcontrollers to ensure their correct operation and adherence to design specifications. It involves the rigorous testing of the microcontroller's functionality, performance, and compliance with the intended design requirements. Effective functional verification helps identify and rectify design flaws, ensuring the microcontroller's reliability, robustness, and optimal performance.

The challenges associated with functional verification arise from the increasing complexity of microcontroller designs and the need to validate their behavior under diverse operating conditions. These challenges include:

Complexity: Modern microcontrollers are highly complex, integrating various functional units, peripherals, and memory subsystems. Verifying the interactions and interdependencies between these components is a significant challenge.

Time-to-Market Pressure: Microcontroller development cycles are often constrained by market demands. Shorter development cycles put pressure on verification teams to deliver comprehensive test coverage within limited timeframes.

Design Changes: Design iterations and modifications are common during the development process, which necessitate frequent updates to the verification environment. Maintaining consistency and accuracy throughout these changes can be challenging.

Functional Coverage: Ensuring that all aspects of the microcontroller's functionality have been thoroughly tested requires defining and tracking functional coverage metrics. Achieving comprehensive coverage can be time-consuming and complex.

Verification Environment Complexity: Developing a robust verification environment that accurately models the microcontroller's behavior and allows efficient testbench development is a non-trivial task. Building reusable and scalable verification components is essential for productivity.

3.2 Universal Verification Methodology (UVM)

The Universal Verification Methodology (UVM) is an industry-standard verification methodology widely used in semiconductor design verification, including microcontroller verification. UVM provides a standardized framework and a set of guidelines for developing scalable, reusable, and modular verification environments.

3.2.1 UVM Components

UVM is composed of various key components, including:

Testbench: The testbench is responsible for creating stimuli to test the microcontroller design, applying transactions, and monitoring responses.

Agents: Agents act as the interface between the testbench and the microcontroller design. They transmit and receive signals and data between the two.

Scoreboard: The scoreboard verifies the correctness of the microcontroller's outputs by comparing them against expected results.

Sequences: Sequences define stimulus generation patterns for testing specific scenarios or use cases. They control the flow of transactions and drive the microcontroller design.

Monitors: Monitors observe the microcontroller's behavior by monitoring its inputs and outputs. They capture transaction information for analysis and coverage.

3.2.2 UVM Testbench Architecture

The UVM testbench architecture follows a hierarchical structure that promotes reusability and scalability. It typically consists of multiple layers, including the test, environment, sequence, driver, monitor, and interface layers. Each layer has specific responsibilities and interfaces with the adjacent layers, enabling modular development and verification.

3.2.3 UVM Phases

UVM defines several phases that govern the execution flow of the verification environment. These phases include build, connect, end_of_elaboration, start_of_simulation, run, extract, check, report, and final phases. Each phase provides a specific context for executing various tasks, such as building the testbench, connecting components, running sequences, and generating reports.

3.3 SystemVerilog

SystemVerilog is a hardware description and verification language that extends the capabilities of the Verilog HDL. It provides advanced features specifically designed for verification, making it a popular choice for functional verification of microcontrollers.

3.3.1 Key Features of SystemVerilog

SystemVerilog includes several key features that enhance the verification process, including:

Object-Oriented Programming (OOP): SystemVerilog supports OOP concepts, allowing the development of reusable and modular verification components. Classes, objects, and inheritance facilitate the creation of complex verification environments.

Assertions: SystemVerilog provides assertion constructs, such as immediate assertions and concurrent assertions, to define properties and constraints that the microcontroller design must satisfy. Assertions enhance the verification process by automating the checking of expected behaviors and detecting violations.

Coverage: SystemVerilog incorporates coverage features that allow the collection and analysis of functional coverage data. Coverage models and bins enable the quantification of how thoroughly the microcontroller design has been exercised during simulation.

3.3.2 SystemVerilog for Verification

SystemVerilog's advanced features, such as constrained randomization, dynamic data types, and DPI (Direct Programming Interface), enable efficient and effective verification of microcontrollers. Constrained randomization allows for the generation of randomized stimuli, covering a wide range of input scenarios. Dynamic data types enhance flexibility in representing complex data structures and behaviors. The DPI facilitates interfacing with other languages, such as C or C++, enabling the integration of pre-existing verification components or algorithms.

3.3.3 Writing Testbenches in SystemVerilog

Writing testbenches in SystemVerilog involves the creation of verification components, stimulus generation, result checking, and coverage collection. SystemVerilog testbenches leverage the language's features to define test scenarios, drive stimulus to the microcontroller design, and verify the correctness of its responses. Testbenches typically utilize OOP concepts to develop reusable and scalable verification environments, improving productivity and maintainability.

In conclusion, the Universal Verification Methodology (UVM) and SystemVerilog are widely employed in functional verification of microcontrollers. UVM provides a standardized methodology and framework, including key components like testbenches, agents, and sequences. SystemVerilog, with its advanced verification features, facilitates efficient testbench development, stimulus generation, result checking, and coverage collection. Together, UVM and SystemVerilog contribute to building comprehensive and effective verification environments for ensuring the correctness and robustness of microcontroller designs.

3.4 Writing Testbenches in SystemVerilog

SystemVerilog provides powerful features for designing and developing testbenches for functional verification. Testbenches play a crucial role in driving stimulus to the design under test (DUT), monitoring its behavior, and checking for expected results. Here is a step-by-step guide on writing testbenches in System Verilog:

Testbench Architecture:
Define the overall structure of your testbench. It typically includes the following components:

- **Testbench module:** The top-level module that instantiates other testbench components.
- **Interface instances:** Connect the DUT to the testbench using interfaces that mirror the DUT's input and output ports.
- **Monitor:** Observes the signals from the DUT and captures transaction-level information for analysis and debugging.
- **Driver:** Drives stimulus to the DUT by generating appropriate signals and transactions.
- **Scoreboard:** Compares the DUT's outputs with expected results to check for correctness.
- **Coverage:** Collects functional coverage data to ensure comprehensive testing.

Interface Definition:
Declare an interface that represents the signals and data paths between the testbench and the DUT. The interface should mirror the DUT's input and output ports, allowing seamless communication between the two.

Testbench Initialization:
Create an instance of the DUT and the testbench modules. Connect the DUT to the testbench via the defined interface.

Test Generation:
Write test sequences that generate input stimuli to exercise different scenarios or test cases. Use constructs like loops, if-else statements, and randomization to create diverse test scenarios.

Driving Stimulus:
In the testbench driver, generate appropriate signals and transactions to drive stimulus to the DUT. This involves applying inputs to the DUT's interface signals based on the test sequences. Use task or function calls to encapsulate reusable stimulus generation code.

Monitoring:
In the testbench monitor, observe the DUT's output signals and capture relevant transaction-level information. This data can be used for analysis, debugging, and coverage collection.

Result Checking:
Compare the DUT's output signals with expected values or use assertions to check for correct behavior. Assertions can

be used to specify properties that the DUT's signals must satisfy, helping to identify violations automatically.

Coverage Collection:
Define functional coverage models to capture the extent of stimulus coverage. Specify coverage bins and attributes to track which parts of the design have been exercised during simulation. Collect coverage data to ensure comprehensive testing.

Testbench Control:
Control the execution flow of the testbench using a test sequence or test control module. This module manages the sequence of tests, handles test initialization, and monitors test completion.

Simulation Setup and Execution:
Set up simulation directives, such as the simulation time, seed values for randomization, and verbosity levels for debug information. Compile and run the simulation to execute the testbench and observe the behavior of the DUT.

Debugging and Analysis:
Analyze simulation results, monitor output, and debug any issues or failures. Use waveform viewers and debugging tools to gain insights into the DUT's behavior and understand the cause of failures.

By following these steps, you can create a robust and effective testbench in SystemVerilog for functional verification of microcontroller designs. The modular and scalable nature of SystemVerilog allows for reusable testbench components and efficient development of comprehensive verification environments.

4. Verification Environment Setup

4.1 Testbench Architecture

The testbench architecture plays a crucial role in setting up the verification environment for the functional verification of a microcontroller design. The testbench architecture defines the structure and organization of the various components that interact with the design under test (DUT). Here are some key considerations for the testbench architecture:

Top-Level Testbench Module: Create a top-level module that instantiates and connects the different testbench components. This module acts as the central hub for controlling and coordinating the verification process.

DUT Instantiation: Instantiate the DUT module within the testbench module. Connect the DUT's input and output ports to the corresponding signals or interfaces in the testbench.

Testbench Components: Identify the necessary components for the testbench, such as monitors, drivers, scoreboards, and coverage collectors. These components work together to drive stimulus to the DUT, observe its behavior, check for correctness, and collect coverage data.

Hierarchical Organization: Consider organizing the testbench components in a hierarchical structure to enhance modularity and reusability. This allows for easier maintenance and scalability as the complexity of the microcontroller design increases.

4.2 Testbench Components

The testbench components in the verification environment are responsible for generating stimulus, observing the DUT's behavior, checking for correctness, and collecting coverage data. Here are some commonly used testbench components:

Monitor: The monitor component observes the signals or interfaces of the DUT and captures transaction-level information. It extracts relevant data from the DUT's outputs and records it for analysis, debugging, and coverage collection purposes.

Driver: The driver component drives stimulus to the DUT by generating appropriate signals or transactions based on the test scenario. It interacts with the DUT's input ports and applies the desired stimuli.

Scoreboard: The scoreboard component compares the DUT's outputs against expected results or golden reference models. It verifies the correctness of the DUT's behavior and flags any discrepancies or errors.

Coverage Collector: The coverage collector component collects functional coverage data during the verification process. It tracks which parts of the microcontroller design have been exercised and provides insights into the completeness of the test suite.

Test Sequencer: The test sequencer manages the sequence of tests or test scenarios. It controls the activation of different test sequences, ensuring that they are executed in the desired order.

4.3 Interface Modeling

Modeling the interfaces between the testbench and the DUT is crucial for establishing communication and data exchange. SystemVerilog provides several constructs for interface modeling, such as interface declarations, modports, and clocking blocks. Consider the following aspects when modeling interfaces:

Input and Output Ports: Define input and output ports in the DUT module and connect them to the corresponding interface signals or modports in the testbench.

Transaction-Level Interfaces: Use interfaces to abstract the communication between the testbench and the DUT. This allows for better encapsulation, modularity, and ease of connectivity.

Clock and Reset: Model the clock and reset signals as part of the interface to synchronize the testbench with the DUT.

Data Path and Control Signals: Identify the data path signals and control signals that need to be connected between the testbench and the DUT. Ensure that the interface accurately reflects the signals required for stimulus generation and result checking.

4.4 Stimulus Generation

Generating stimulus is a crucial aspect of the verification process. The testbench should be capable of generating diverse and meaningful test scenarios to thoroughly exercise the microcontroller design. Consider the following techniques for stimulus generation:

Directed Testing: Develop test sequences that follow specific scenarios or use cases to target specific functionalities or corner cases. These test sequences can be manually written to achieve specific goals.

Constrained Randomization: Utilize constrained randomization techniques

Functional Coverage: Leverage functional coverage to guide stimulus generation. Define coverage goals and use coverage-driven techniques to ensure that the testbench generates stimulus to cover all relevant aspects of the microcontroller design.

4.5 Coverage and Analysis

Coverage analysis is essential to assess the thoroughness and completeness of the verification process. By collecting coverage data, you can determine how well the microcontroller design has been exercised during simulation. Consider the following aspects for coverage and analysis:

Functional Coverage Models: Define functional coverage models to specify coverage goals and metrics. Identify the important aspects of the microcontroller design to be covered and create coverage points accordingly.

Coverage Bins and Attributes: Specify coverage bins to categorize the different scenarios or values that the microcontroller design can encounter. Define attributes to track the occurrence and coverage of these bins.

Coverage Collection: Instrument the testbench and the DUT to collect coverage data during simulation. Utilize coverage sampling techniques to efficiently collect coverage information without incurring excessive simulation overhead.

Coverage Reports: Generate coverage reports to analyze the coverage data and identify any gaps or areas that require additional testing. Use coverage visualization tools to gain insights into the coverage results and track progress towards coverage goals.

By setting up a comprehensive verification environment with well-designed testbench architecture, appropriate testbench components, accurate interface modeling, effective stimulus generation techniques, and coverage and analysis mechanisms, you can ensure the thorough and reliable functional verification of the LC3 microcontroller.

5. Results and Analysis

5.1 Simulation Results

Simulation results provide valuable insights into the behavior and performance of the LC3 microcontroller during the functional verification process. The simulation results should be thoroughly analyzed to identify any potential issues, bugs, or deviations from the expected behavior. Here are some key considerations for analyzing simulation results:

Signal Waveforms: Review the waveforms of the input and output signals to ensure they exhibit the expected behavior. Look for any anomalies, unexpected transitions, or timing violations

Error and Warning Messages: Pay attention to any error or warning messages generated during the simulation. These messages can provide valuable information about issues encountered during the verification process.

Stimulus and Response Analysis: Evaluate the effectiveness of the generated stimuli in exercising different functionalities and scenarios. Compare the responses of the microcontroller against the expected results to identify any discrepancies.

Performance Metrics: Measure and analyze the performance metrics of the microcontroller, such as execution time, latency, throughput, and power consumption. Compare these metrics against the design specifications to ensure they meet the requirements.

5.2 Coverage Analysis

Coverage analysis is essential to assess the completeness of the verification process and identify any gaps in the test suite. By analyzing coverage data, you can determine which parts of the microcontroller design have been exercised and which areas require further testing. Consider the following aspects for coverage analysis:

Coverage Metrics: Analyze coverage metrics, such as statement coverage, branch coverage, toggle coverage, and condition coverage. Assess the achieved coverage percentage and compare it against the defined coverage goals.

Uncovered Areas: Identify any uncovered or low-coverage areas of the microcontroller design. These areas may indicate missing or insufficient test scenarios and may require additional tests to achieve higher coverage.

Coverage Trends: Monitor coverage trends across different simulation runs or iterations. Identify any areas where coverage is consistently low or improving slowly, indicating potential testing gaps that need to be addressed.

Coverag Closure: Assess the coverage closure, which is the process of reaching the desired coverage goals. Determine if the achieved coverage is sufficient for the microcontroller's intended functionality and make informed decisions about coverage improvements if necessary.

6.3 Functional Coverage

Functional coverage focuses on capturing and analyzing the coverage of specific functionalities or features of the microcontroller. It ensures that critical parts of the design are adequately exercised during the verification process. Consider the following aspects for functional coverage:

Coverage Goals: Define functional coverage goals based on the microcontroller's specifications and requirements. Identify the important functionalities, corner cases, and scenarios that need to be covered.

Coverage Points: Define coverage points that represent specific functionalities, events, or scenarios. These coverage points can include instruction coverage, register access coverage, interrupt handling coverage, or peripheral interactions.

Coverage Metrics: Determine the appropriate coverage metrics to measure the completeness of the functional coverage. For example, instruction coverage may measure the percentage of executed instructions, and register coverage may measure the percentage of registers accessed.

Coverage Analysis: Analyze the functional coverage data to determine the completeness of the coverage for different functionalities. Identify any areas where coverage is lacking and devise additional test scenarios to improve coverage.

Coverage Closure: Similar to overall coverage, assess functional coverage closure by comparing the achieved coverage with the defined coverage goals. Determine if the coverage is sufficient for the critical functionalities and make adjustments as needed.

By thoroughly analyzing simulation results, assessing coverage metrics, and ensuring comprehensive functional coverage, you can gain confidence in the correctness and completeness of the LC3 microcontroller's functional verification process. This analysis helps identify any potential issues, improve the quality of the design, and ensure the microcontroller meets the desired specifications and requirements.

References

[1] Greg Tumbush, "Advanced Verification Techniques: A SystemVerilog Testbench and Verification Methodology Primer," Springer, 2014.
[2] Ben Cohen, "SystemVerilog for Verification: A Guide to Learning the Testbench Language Features," Springer, 2012.
[3] Chris Spear, Greg Tumbush, "SystemVerilog for Verification: A Guide to Learning the Testbench Language Features," Springer, 2012.
[4] Dennis Brophy, "UVM Primer for SystemVerilog," Springer, 2013.
[5] Stuart Sutherland, Simon Davidmann, Peter Flake, "SystemVerilog Assertions and Functional Coverage: Guide to Language, Methodology and Applications," Springer, 2016.
[6] IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language (IEEE Std 1800-2017).
[7] Accellera Systems Initiative, "Universal Verification Methodology (UVM) User's Guide," Accellera UVM Working Group, 2013.
[8] Open Verification Methodology Cookbook, https://verificationacademy.com/cookbook/uvm
[9] Doulos UVM Golden Reference Guide, https://www.doulos.com/knowhow/uvm/uvm-golden-reference-guide/
[10] VerificationAcademy, https://verificationacademy.com/
[11] Mark Glasser, Stuart Sutherland, "SystemVerilog Assertions Handbook, 4th Edition," Sutherland HDL, Inc., 2016.

Author Profile

Sai Madhav Modepalli received the B.tech degree in Electrical and Electronics Engineering from National Institute of Technology Karnataka Surathkal in the year 2022. Currently he's working with Honeywell, Inc. as Hardware Engineer.