FACULTY OF ENGINEERING AND TECHNOLOGY
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
CHIP DESIGN VERIFICATION
ENCS5337

Course Project
Design Verification of a Hardware Compression and Decompression Chip

---

Prepared By
Raghad Afaghani 1192423
Doha Hmeid 1190120

Instructor
Dr. Ayman Hroub

Birzeit University
June, 2024

Abstract

This project involves the design verification of a hardware compression and decompression chip using a dictionary-based algorithm. The chip compares input data to stored data, outputting an index as the compressed version, and retrieves data during decompression. It includes various input/output ports for commands and status signaling. The project is divided into three phases: reference model creation, verification plan development, and complete UVM verification environment implementation.

Contents

1 Introduction **1**

1.1 DUT Overview . . . 1
1.2 Compression Algorithm . . . 1
1.3 Decompression Algorithm . . . 1
1.4 Input/Output ports . . . 2
1.5 Block Diagram . . . 2

2 Verification plan **3**

2.1 Description of the Verification Levels . . . 3
2.2 Functions to be Verified . . . 3
2.3 Required Human Resources . . . 3
2.4 Required Tools . . . 3
2.5 Schedule Details . . . 3
2.6 Specific Tests and Methods . . . 3
2.7 Coverage Requirements . . . 3
2.8 Completion Criteria . . . 4
2.9 Test Plan (Matrix) . . . 4
2.10 Risks and dependencies . . . 4

3 Reference model workflow **5**

4 UVM **9**

4.1 RTL Design . . . 9
4.2 Interface . . . 9
4.3 TestBench . . . 9
4.4 Sequence Item . . . 10
4.5 Sequence . . . 10
4.6 Sequencer . . . 10
4.7 UVM Driver . . . 10
4.8 UVM Monitor . . . 11
4.9 UVM Agent . . . 11
4.10 Scoreboards . . . 11
4.11 Environment . . . 12

5 Simulation Results **13**

5.1 Tests . . . 13
5.2 UVM Report . . . 13
5.2.1 Compression case and `data_in` is not in the memory . . . 13
5.2.2 Invalid operation case . . . 13
5.2.3 Compression case and data_in is not in the memory . . . 14
5.2.4 Decompression case and compressed_in is not in the memory . . . 14
5.2.5 No operation case . . . 16
5.3 Simulation Waveforms . . . 17

6 Functional Coverage **19**

6.1 Coverage Report 19
6.2 Coverage Results 19

7 Instructions to run the simulation **21**

8 Conclusion **22**

9 Appendices **24**
9.1 Appendix A: design.sv 24
9.2 Appendix B: chip_if.sv 26
9.3 Appendix C: testbench.sv 26
9.4 Appendix D: chip_sequence_item.sv 27
9.5 Appendix E: chip_sequence.sv 28
9.6 Appendix F:chip_sequencer.sv 29
9.7 Appendix G: chip_driver.sv 29
9.8 Appendix H: chip_monitor.sv 30
9.9 Appendix I:chip_agent.sv 32
9.10 Appendix J:chip_scoreboard.sv 33
9.11 Appendix K:chip_env.sv 35
9.12 Appendix L:chip_test.sv 36

List of Figures

1 Chip block diagram. 2
2 Memory snapshot before the test 5
3 Memory snapshot after the test 6
4 Reference Model CaseTest 1 6
5 Reference Model CaseTest 2 6
6 Reference Model CaseTest 3 7
7 Reference Model CaseTest 4 7
8 Reference Model CaseTest 5 7
9 Reference Model CaseTest 6 7
10 Reference Model CaseTest 7 8
11 Reference Model CaseTest 8 8
12 Reference Model CaseTest 9 8
13 UVM Structure Block 9
14 Agent Functionality 11
15 Scoreboard Functionality 12
16 UVM summary report 13
17 Test cases 1 14
18 Test cases 2 15
19 Test cases 3 15
20 Test cases 4 16
21 Test cases 5 16
22 No operation test case 17
23 Simulation Waveform1 17
24 Simulation Waveform2 17
25 Simulation Waveform3 18
26 Simulation Waveform4 18
27 Coverage Driven Environment 19
28 Sample Coverage Report 19
29 Final Coverage Result 20
30 Run Instructions for EDA Platform 21

List of Tables

1 Description of Ports 2
2 Functions to be Verified in DUT 3
3 Test Plan 4

Acronyms and Abbreviations

Abbreviation	**Original Sequence**
DUT	Design Under Test
EDA	Electronic Design Automation
UVM	Universal Verification Methodology

1 Introduction

1.1 DUT Overview

The DUT is a hardware compression and decompression chip employing a dictionary-based algorithm. It performs data compression by receiving input data through the '$data\_in$' port, comparing it with stored data in its internal dictionary memory, and outputting an index on the '$compressed\_out$' port. If the data is not found, it is added to the dictionary, and an index is assigned. For decompression, the chip retrieves data based on the index received via the '$compressed\_in$' port and outputs the original data on the '$decompressed\_out$' port. The DUT includes various input/output ports, such as 'clk', '$reset$', '$command$', and '$response$', to manage operation commands and status signaling. It ensures efficient data handling and error reporting for full memory or invalid indices.

1.2 Compression Algorithm

Compression is a fundamental concept in data processing aimed at reducing the size of data while preserving its essential information. The compression algorithm implemented in a hardware chip typically follows a systematic set of steps to achieve efficient data compression.

When input data is received through the `data_in` port, it is compared to the existing data stored in the chip's internal memory, known as the dictionary memory. If a match is found between the input data and an entry in the dictionary memory, it indicates that the input data already exists. In this case, the chip writes the corresponding index of the stored data into the `compressed_out` port. This index value represents a compressed version of the input data, allowing for efficient representation and storage of repeated patterns or information.

In situations where the input data is not found in the dictionary memory, suggesting it is new data, the chip allocates the next available slot within the memory to store the input data. Subsequently, the chip writes the corresponding index of the newly stored data into the `compressed_out` port and increments the index value to reflect the addition of the new entry.

The physical index register of the chip typically has a width of 32 bits. However, the actual number of bits used from this register depends on the size of the dictionary memory. For instance, if the dictionary memory consists of 256 locations, each occupying 80 bits, the chip utilizes the least significant eight bits of the 32-bit index register. This ensures that the compressed data size is reduced to 8 bits, optimizing storage and transmission efficiency.

To handle scenarios where the internal memory becomes full during the compression process, the chip generates an error signal through the output response. This error signal acts as an indicator that the available memory space is insufficient to accommodate further compression operations, providing feedback to the system for appropriate error handling or memory management.

1.3 Decompression Algorithm

The decompression algorithm in the hardware chip functions as the inverse of the compression algorithm. When the chip receives compressed data through the `compressed_in` port, it compares the value of the compressed data to the current index register value. If the compressed data value is less than or equal to the index value, the corresponding decompressed data exists in the dictionary memory, and it is outputted through the `decompressed_out` port. However, if the compressed data value is greater than the index value, indicating the absence of the decompressed data in the dictionary memory, an error is reported.

1.4 Input/Output ports

The input/output ports of the compression/decompression chip are listed in Table 1 as follows:

Port	Default Width (#bits)	Direction	Description
clk	1	Input	
reset	1	Input	Clears the dictionary memory and the index register
command	2	Input	Specifies the chip operation: 00: No operation 01: Compression 10: Decompression 11: Invalid command, report an error
data_in	80	Input	Data to be compressed
compressed_in	8	Input	Data to be decompressed
compressed_out	8	Output	Output compressed data
decompressed_out	80	Output	Output decompressed data
response	2	Output	Shows the status of the output: 00: no valid output 01: valid compressed_out 10: valid decompressed_out 11: Error

Table 1: Description of Ports

1.5 Block Diagram

The block diagram illustrates a data processing system designed for compression and decompression tasks. It receives various inputs such as raw data, compressed data, commands, a reset signal, and a clock signal. The system consists of two primary components: an Index Register and a Dictionary Memory. These components collaborate to generate compressed and decompressed outputs, while also providing a response output that indicates the operation's status or outcome. The diagram showcases a well-organized data processing flow, incorporating control and synchronization mechanisms to manage the data effectively.

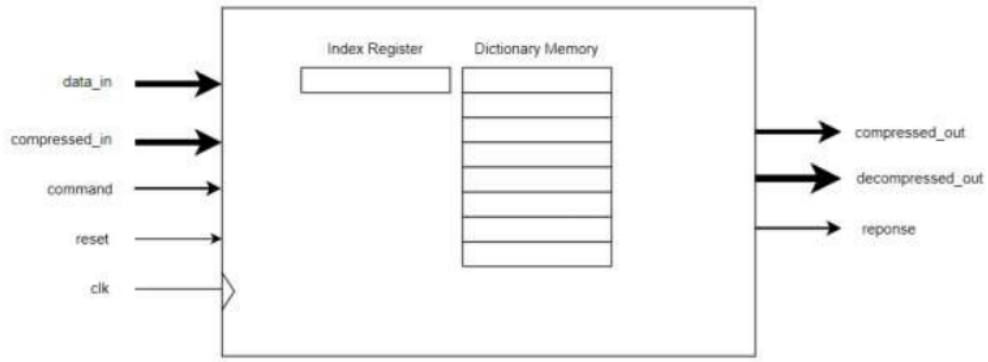

Figure 1: Chip block diagram.

2 Verification plan

2.1 Description of the Verification Levels

The verification of this chip is done at the top level to verify the functionality of the whole chip and ensure that it works as demonstrated in the chip specifications.

2.2 Functions to be Verified

The table below list the functions of the DUT that are going to be verified.

Critical functions	Secondary functions
The Compression Algorithm	Reset Function
The Decompression Algorithm	Clock Scheduling Function
Command Handling Function	Checking Memory Status function
Error Detection and Reporting Function	Generating the output (response) function

Table 2: Functions to be Verified in DUT

2.3 Required Human Resources

The human resources required are design verification engineers and for this project we (Doha Hmeid and Raghad Afghani) are the human resources.

2.4 Required Tools

Below are the needed tools for the verification process:

- Simulation engine/environment
- Waveform viewer
- Testbench (will be created using SystemVerilog/UVM)
- Memory – to store the test data and save the simulation results

2.5 Schedule Details

The whole verification process must take 2 weeks.

2.6 Specific Tests and Methods

The verification environment type is black box, since we are going to check the whole DUT chip. The verification strategy will include some deterministic test to make sure we covered the corner cases. It will also include pseudo random generated tests to make sure we get the highest coverage ratio possible. For checking, the testbench, waveforms will be used by checking the I/O ports to ensure data correctness.

2.7 Coverage Requirements

The verification must cover all the functional features of the DUT including the compression and decompression algorithms, input/output ports, command handling, and error detection. It must also achieve a code coverage percentage of around 85%.

2.8 Completion Criteria

The verification process is completed when:

- All the test cases are done successfully.
- The coverage target is achieved.
- There are no open issues or reviews left for the DUT.
- The final regression results are bug-free.

2.9 Test Plan (Matrix)

Topic	Test#	Description
Input Command sequences	1.1	No Operation: the module remains inactive with no changes to its state or outputs.
Input Command sequences	1.2	Compression: searches for existing data, outputs its index if found, or stores it and outputs the new index if not.
Input Command sequences	1.3	Decompression: retrieves and outputs data corresponding to the provided index, or reports an error if the index is invalid.
Input Command sequences	1.4	Invalid Command: Immediately sets the response to an error state (2'b11), performing no other action.
Data compression algorithm	2.1	Verify that the compressed output matches the index of stored data in dictionary memory for known inputs.
Data decompression algorithm	3.1	Verify that the decompressed output matches original data for inputs that have previously been compressed.
Error generation	4.1	Verify that an error is generated when the data input is empty.
Error generation	4.2	Verify that an error is generated when compression operation fails.
Error generation	4.3	Verify that an error is generated when decompression operation fails.
Ports	5.1	Verify that the response port reflects the output correctly.
Reset	6.1	All internal states (memory, index register) are reset to default values upon reset signal.
Full system integration	7.1	The entire chip works as expected under realistic, complex data patterns and operational conditions.
Corner cases	8.1	Memory overflow - Verify that an error is generated when the memory is full and a data needs to be written in it (compression case and input not found in memory).

Table 3: Test Plan

2.10 Risks and dependencies

The risks that might affect the verification process are:

- Not meeting the defined verification schedule.
- Incomplete or incorrect test cases.

3 Reference model workflow

The reference model implemented in this project serves as an executable specification, providing a golden model that accurately predicts the correct results based on the given stimulus. The implementation approach for the reference model can vary depending on factors such as the way specifications are written and the specific design requirements. It offers flexibility and adaptability, allowing for different approaches to be employed in implementing the reference model, tailored to the specific needs of the project.

A reference model in System Verilog was created due to its simplicity in integrating with our project. The code for the reference model can be found in Appendix A. Additionally, the test bench code can be found in Appendix B.

A file named ‘memory.hex‘ was created to simulate the actual memory. The initial values for the memory file were:

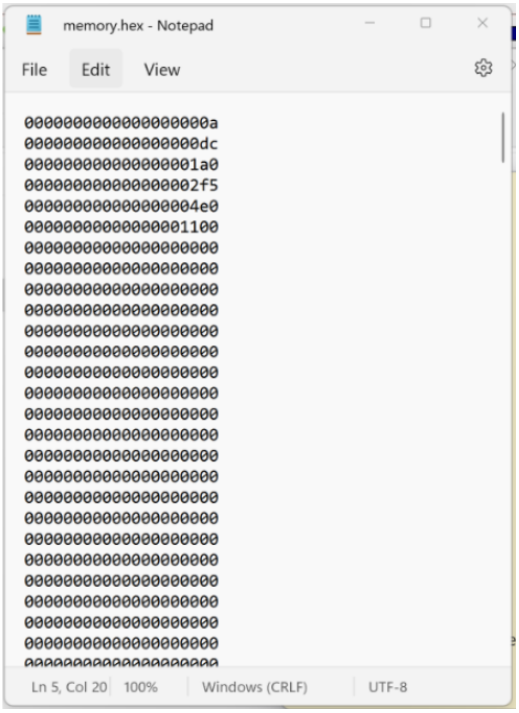

Figure 2: Memory snapshot before the test

After running the test bench, we got the following memory. Notice that when we entered ‘$data\_in$‘ as ‘0x200‘ and ‘0x03‘. These values were not in the original memory, so after running the program, they were written on the last empty cell from the memory as expected.

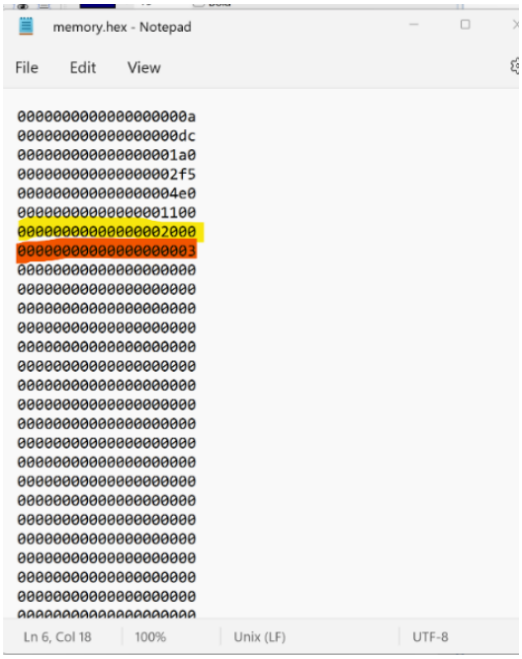

Figure 3: Memory snapshot after the test

The tests and their results are explained below:

1. When the command was 00, this means no operation and the response gave 00 which is no valid output since there is no operation.

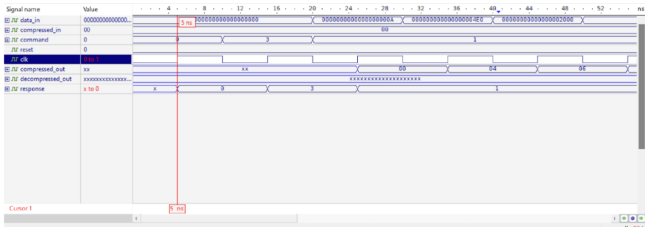

Figure 4: Reference Model CaseTest 1

2. When the command was 3 = 11, this means invalid command and the response gave 3 = 11 which indicates that an error happened.

Figure 5: Reference Model CaseTest 2

3. When the command was 1 = 01, this means compression operation and the $data\_in$ was $0x0A$ which exists in index 0 in the memory as shown in the memory snapshot in Figure **??**. We got the response as 1 = 01 which indicates that the data was found in the memory and the $compressed\_out$ is $0x00$ which refers to the index the data was found in the memory.

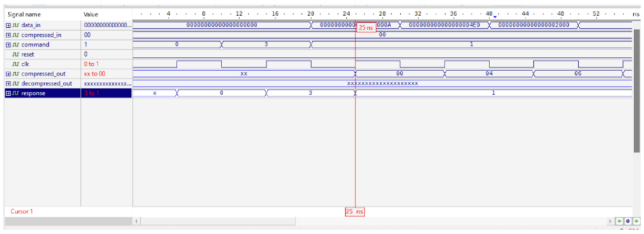

Figure 6: Reference Model CaseTest 3

4. When the command was 1 = 01, this means compression operation and the $data\_in$ was $0x2000$ which doesn't exist in the memory. The data was written at the current index register which points to the last empty slot in the memory which is at index $0x06$. So, $compressed\_out$ is $0x06$ and the response as 1 = 01. After that the index register was incremented.

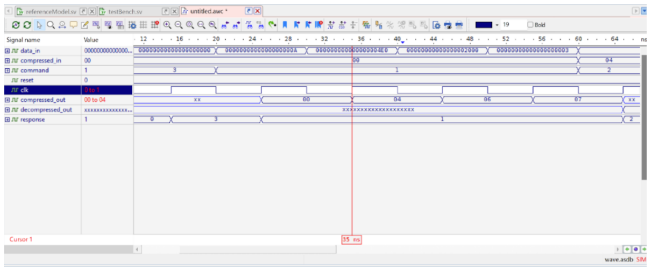

Figure 7: Reference Model CaseTest 4

5. When the command was 1 = 01, this means compression operation and the $data\_in$ was $0x03$ which doesn't exist in the memory. The data was written at the current index register which points to the last empty slot in the memory which is at index $0x07$. So, $compressed\_out$ is $0x07$ and the response as 1 = 01. After that the index register was incremented.

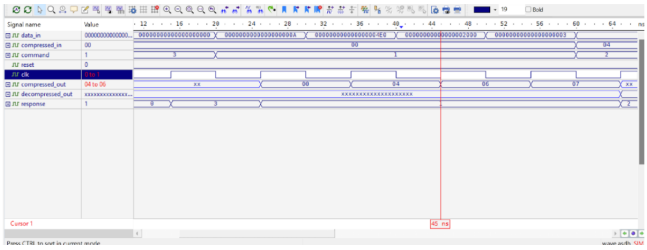

Figure 8: Reference Model CaseTest 5

6. When the command was 2 = 10, this means decompression operation and the $compressed\_in$ was $0x04$ which exists in the memory and the data at that index is $0x04E0$ so, $decompressed\_out$ is $0x04E0$ and the response is 2 = 10 which indicates valid decompressed output.

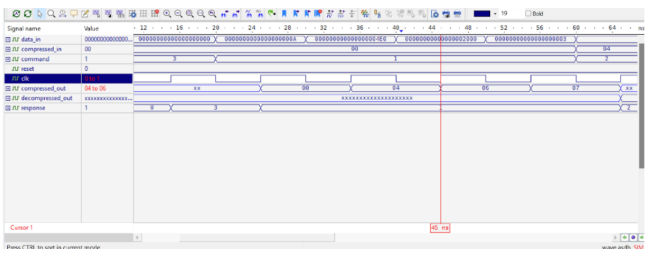

Figure 9: Reference Model CaseTest 6

7. When the command was 2 = 10, this means decompression operation and the $compressed\_in$ was $0x02$ which exists in the memory and the data at that index is $0x01A0$ so, $decompressed\_out$ is $0x01A0$ and the response is 2 = 10 which indicates valid decompressed output.

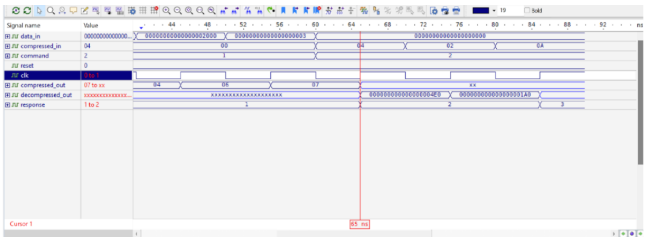

Figure 10: Reference Model CaseTest 7

8. When the command was 2 = 10, this means decompression operation and the $compressed\_in$ was $0x0A = 10$ which doesn't exist in the memory, the response was 3 = 11 which indicates an error and invalid decompressed output.

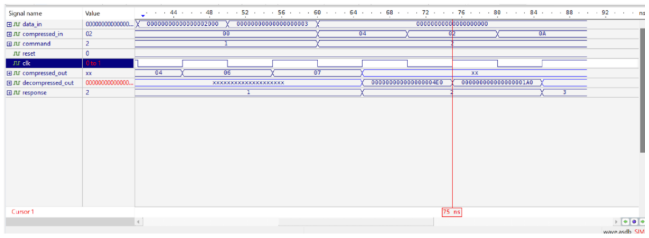

Figure 11: Reference Model CaseTest 8

9. When the command was 1 = 01, this means compression operation and the $data\_in$ was $0x4E0$ which exists in index 4 in the memory as shown in the memory snapshot in Figure **??**. We got the response as 1 = 01 which indicates that the data was found in the memory and the $compressed\_out$ is $0x04$ which refers to the index the data was found in the memory.

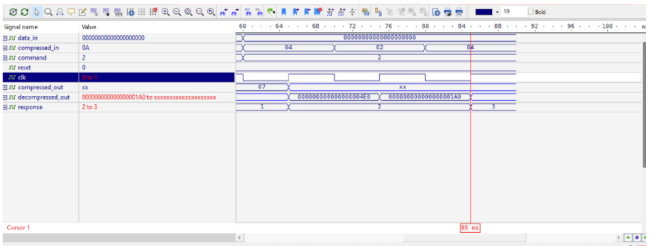

Figure 12: Reference Model CaseTest 9

4 UVM

UVM is a standard method used in the semiconductor industry to check and confirm that digital designs and chips work correctly. It is based on the SystemVerilog language and helps create flexible test parts that can be easily used in the checking process. UVM includes guidelines and good practices for making test setups and running tests. It the main method used by designers and engineers to make sure that the chip designs function as expected.[4].

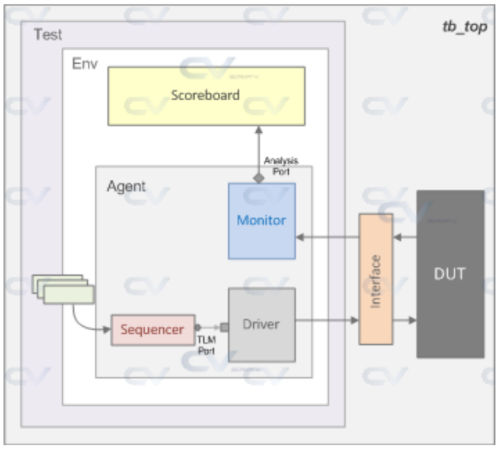

Figure 13: UVM Structure Block

4.1 RTL Design

RTL (Register Transfer Level) design is a module that implements a simple compression and decompression algorithm using a dictionary-based approach. It has various input and output signals, such as a clock, reset, command signals to specify the operation, 80-bit input data to be compressed, and 8-bit input data to be decompressed. The module maintains an internal memory array called a "dictionary" to store the data, with the dictionary size determined by a parameter. When compressing data, the module checks if the input exists in the dictionary and outputs the corresponding index if found, otherwise it tries to store the new data in the dictionary. For decompression, the module retrieves the decompressed data from the dictionary based on the input index. The module also includes a helper function to find the first available empty slot in the dictionary. Overall, this RTL design provides a basic implementation of a compression and decompression system using a dictionary-based technique.[5]

4.2 Interface

The interface defined provides a standardized way to interact with a module or component in the digital circuit design, encapsulating the various input and output signals required by the module and facilitating easier integration with other parts of the system. It includes common control signals such as clock and reset, as well as signals representing commands, input and output data in both compressed and decompressed forms, and response signals, all of which are specified with clear signal widths to establish a well-defined interface that improves the reusability and maintainability of the overall system design by making it simpler to understand and integrate the module within the larger system.

4.3 TestBench

Testbench is a top-level module that serves as the entry point for testing a digital circuit design. It sets up the necessary environment and signals required to exercise the Design Under Test (DUT), including generating

the clock and reset signals, creating an instance of the interface that defines the DUT's input and output signals, and connecting the interface to the DUT. Additionally, the testbench configures the verification environment, enables waveform dumping for analysis, and initiates the test execution. This type of testbench is a common approach in digital circuit verification, as it allows for the integration of various verification methodologies and the capture of debug information to ensure the correctness and functionality of the DUT.

In our project, the compression and decompression module is carefully validated using the testbench structure as a foundation. This verification process is handled by the `tb_top` module, which begins with the definition of the clock and reset signals. The testbench creates a clean, synchronised testing environment with clock and reset generation blocks. After that, it creates the compression/decompression module (DUT) and the interface (`chip_if`), enabling data interchange and communication between the DUT and the testbench. Configuration events improve debugging and observability. Examples include configuring waveform dumping and making the interface globally accessible. The `run_test` task is used to carry out the test, which starts pre-planned test sequences designed to fully verify the functionality of the module.

4.4 Sequence Item

The sequence item in UVM is a class that represents the smallest unit of test data sent from a sequence to a driver. This item, also called a transaction, includes the data and control actions needed for testing the DUT. In our code, the `chip_sequence_item` class is designed to handle different types of input commands, data to be compressed, and previously compressed data for decompression. Each instance of this class is randomized within its sequence to ensure varied testing scenarios before being sent to the driver. The class also has utility macros to help with common tasks like printing, copying, and cloning, making it easy to manage and track during simulations.[2].

4.5 Sequence

The sequence in our UVM code is responsible for creating, randomizing, and sending stimulus items to a driver. This sequence is used to generate different test scenarios for the DUT as in our `chip_sequence` code.[2].

4.6 Sequencer

As clear in our `chip_sequencer` code ,The sequencer is a component responsible for coordinating the execution of stimulus in the form of sequences and sequence items from a parent sequence. It feeds the driver component with these transactions. Essentially, a sequencer acts as an arbiter, determining which sequence gets access to the driver and, by extension, the interface.[2].

4.7 UVM Driver

The Driver component in a UVM-based verification environment bridges the gap between high-level transactions and the specific signal-level changes required by the design's interface protocol. This ensures seamless operation between the abstract and concrete layers of the testbench, simplifying system operation for improved efficiency. Our provided code defines a class named `chip_env`, which manages chip-level verification by overseeing the construction and interconnection of essential components like `chip_agent` and `chip_scoreboard`. This facilitates smooth communication between these components. Additionally, it includes a reporting method to provide feedback on test completion and coverage metrics.

4.8 UVM Monitor

The main responsibility of a UVM monitor is to observe and record the interactions between the testbench and the Design Under Test (DUT) at the interface level. The chip_monitor class in our code exemplifies this by collecting information about the DUT's behavior, including input commands, output responses, and data changes. This information is encapsulated into transaction objects and sent to other parts of the testbench, such as scoreboards or coverage collectors, for analysis. Additionally, the chip_monitor class contains cover groups to facilitate coverage analysis.

4.9 UVM Agent

An Agent in a verification component represents a specific logical interface, such as a SystemVerilog interface, and can be configured as either active or passive - an Active Agent generates and drives stimulus to the Design Under Test (DUT) using a Driver, Sequencer, and Monitor components, while a Passive Agent simply monitors the interface activity without actively driving the DUT, consisting only of a Monitor component. The chip_agent class in our code encapsulates the driver, monitor, and sequencer components. It is responsible for managing the interaction between these components during the verification process. In the build_phase, the agent initializes its components, creating a driver, sequencer (if the agent is active), and a monitor. The connect_phase establishes connections between the driver and sequencer components, enabling the flow of stimulus items generated by the driver to the sequencer for further processing. This architecture facilitates efficient and organized verification of the chip-level design.

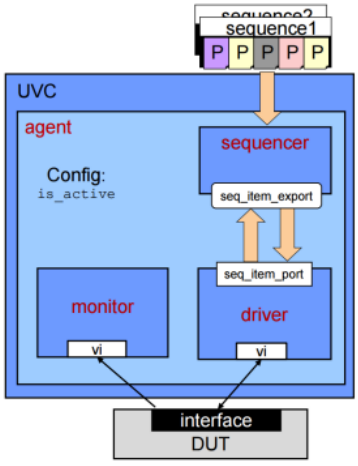

Figure 14: Agent Functionality

4.10 Scoreboards

As appears in our chip_scoreboard code,The scoreboard in a UVM testbench for a compression and decompression module serves as a critical component that checks the correctness of processed data against expected outcomes. It receives transactions from the monitor, which include both the original data and the compressed or decompressed results, and compares these results against a reference model or expected values. The scoreboard has an internal storage to keep track of these expected results and actual outcomes. If discrepancies are found, it logs errors; otherwise, it confirms the pass status of the test scenarios. This helps ensure that the DUT behaves as expected under various conditions, verifying both the functionality and reliability of the compression and decompression processes.[3]

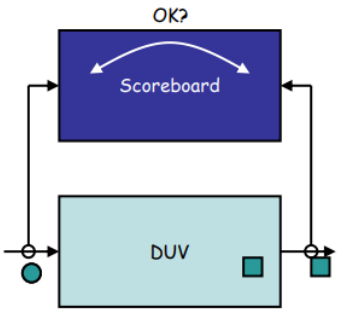

Figure 15: Scoreboard Functionality

4.11 Environment

Our environment UVM code called chip_env, is designed to verify a chip's functionality. It includes essential components like a chip_agent and a chip_scoreboard, which are instantiated during the build_phase. During the connect_phase, the agent's monitoring outputs are linked to the scoreboard for result analysis. The environment is finalized in the report_phase, where it logs a message indicating the successful completion of the simulation, confirming that the chip performs as expected based on the predefined specifications and tests.

5 Simulation Results

5.1 Tests

The chip_test.sv file was created to incorporate the tests produced for the chip. It creates instances from the chip interface, environment, and sequence, and proceeds to construct these instances during the build phase. Subsequently, in the run phase, it generates a total of 80 random test cases, with each test case commencing on the positive clock edge. The code for chip_test.sv is available in Appendix L.

5.2 UVM Report

Following the execution of the UVM project within the electronic design automation (EDA) environment, all generated test cases were examined to verify the proper functioning of the Universal Verification Methodology (UVM). The UVM Summary report, depicted in Figure 16, confirms the absence of any errors detected in the Design Under Test (DUT).

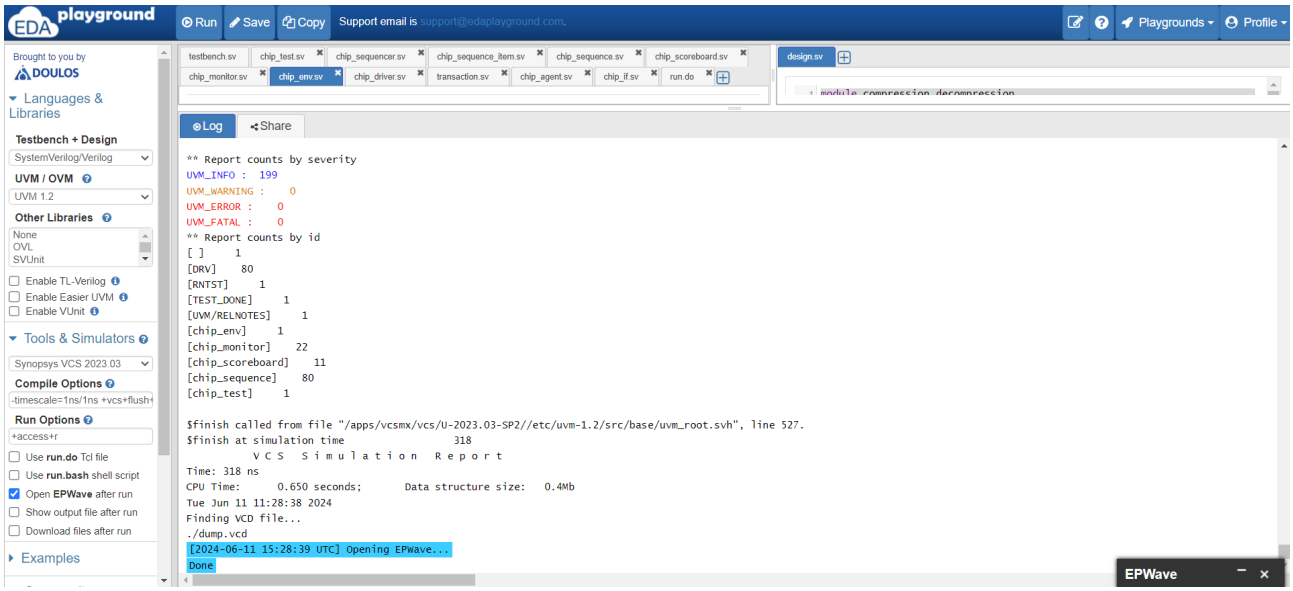

Figure 16: UVM summary report

As stated earlier, a total of 80 test cases were developed. The initial test cases are elucidated below. Figure 17 illustrates the initial group of tests. The yellow highlights indicate the test case inputs generated, while the pink highlights represent the output computed by the Device Under Test (DUT) and the determination of test case success or failure conducted by the Universal Verification Methodology (UVM).

5.2.1 Compression case and `data_in` is not in the memory

In the initial clock cycle, the monitoring system identified a compression test scenario with a command value of 1. Given that the memory was devoid of data, the DUT successfully wrote the `data_in` to the first memory index. The `compressed_out` value was noted as 0x00 at memory index 0, with a corresponding response of 0x01, thereby validating the compression output.

5.2.2 Invalid operation case

Subsequently, in the second clock cycle, the monitoring system detected an invalid operation scenario with the command set to 3. In response, the DUT correctly generated a 0x03 response signifying an error, thus

demonstrating successful handling of the test case.

5.2.3 Compression case and data_in is not in the memory

During the third clock cycle, the monitoring system identified another compression test scenario with the command set to 1. At this point, the memory contained one data item, distinct from the previous test. The DUT proceeded to write this new data_in to the next available memory index, which was now 0x02 (as index 0x01 was occupied from the previous step). The successful execution of this task was confirmed by observing a compressed_out value of 0x01 at memory index 1, along with a response of 0x01, indicating a valid compression output.

5.2.4 Decompression case and compressed_in is not in the memory

In the subsequent clock cycle, the monitoring system detected a decompression test scenario with the command set to 2. Given that the compressed input was 29 and memory index 29 remained unoccupied, the DUT correctly generated an error response of 0x03, as expected, signifying the absence of data at the specified index.

Figure 17: Test cases 1

Figure 18: Test cases 2

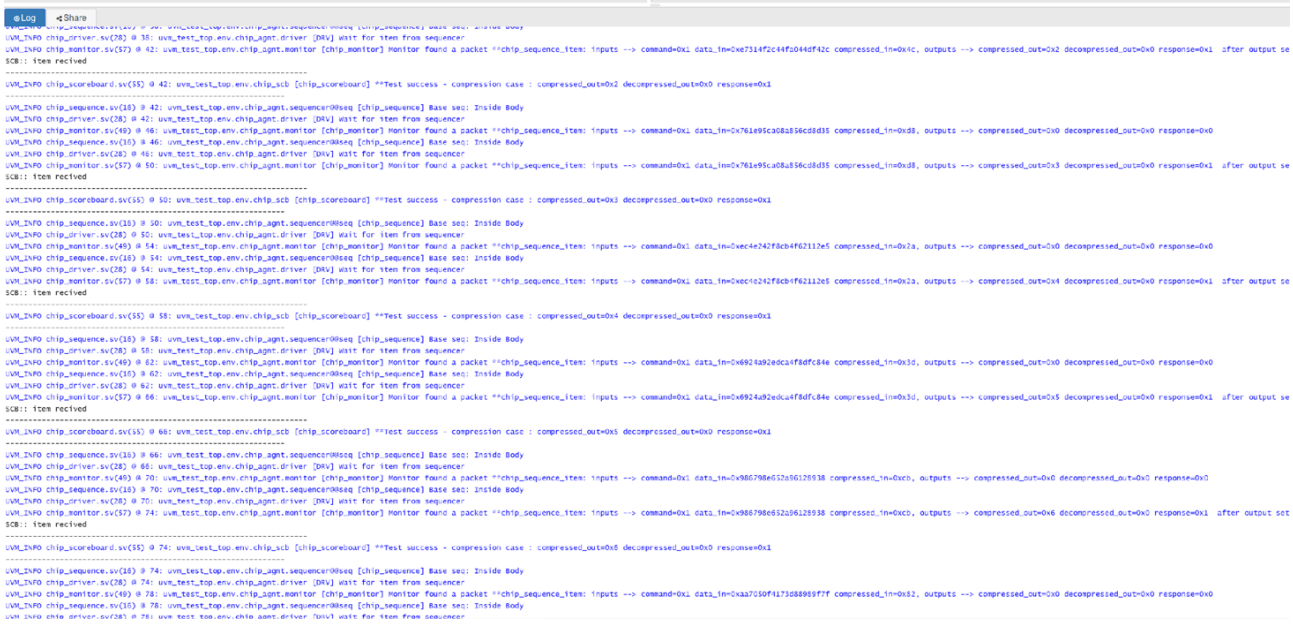

Figure 19: Test cases 3

Figure 20: Test cases 4

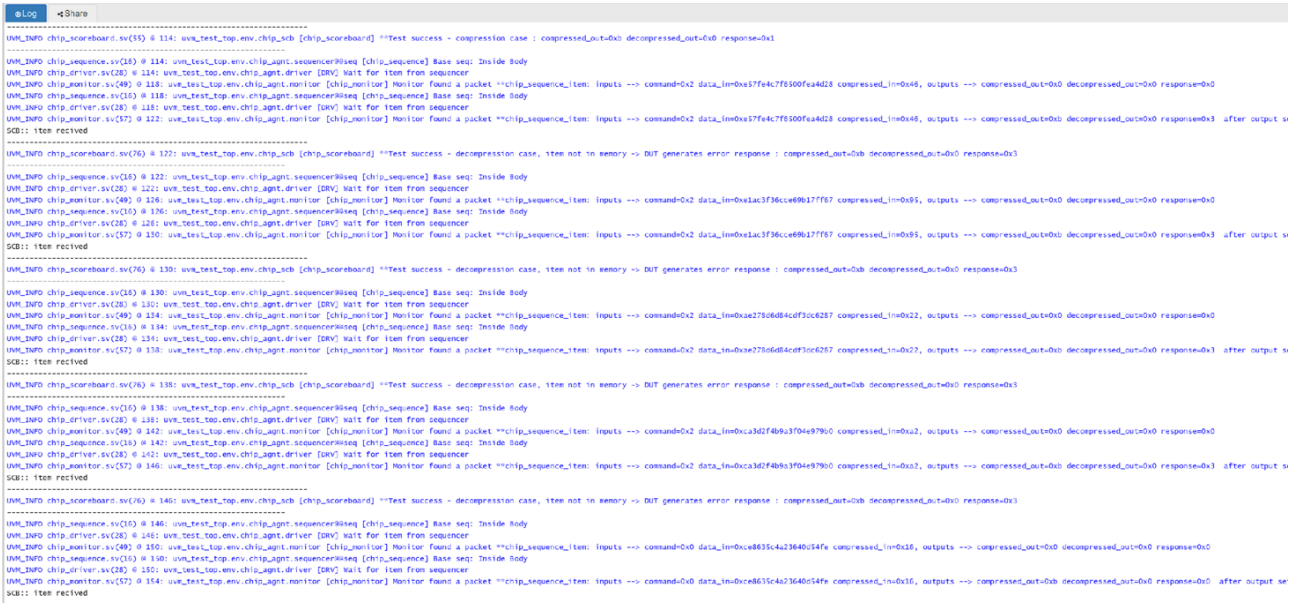

Figure 21: Test cases 5

5.2.5 No operation case

The test scenario provided demonstrates the clock cycle in which no operation occurs when the command is set to 0. The Device Under Test (DUT) appropriately produced a 0 response, indicating no output, and operated as intended, thus confirming the success of this test case.

Figure 22: No operation test case

5.3 Simulation Waveforms

In addition, we recorded the test cases along with their randomly generated inputs and outputs, and visualized them as waveforms to enhance the clarity of the simulation. The EPWave tool, integrated within the Electronic Design Automation (EDA) framework, was employed for the visualization of these waveforms. The figures presented below illustrate the waveforms spanning the entire simulation duration, which extended for a total of 318 nanoseconds. It is important to observe that at the commencement of the simulation, all ports of the Device Under Test (DUT) were reset, following which they began to exhibit varying values as the simulation progressed. Furthermore, it is evident that with each positive clock edge, a new test scenario was generated.

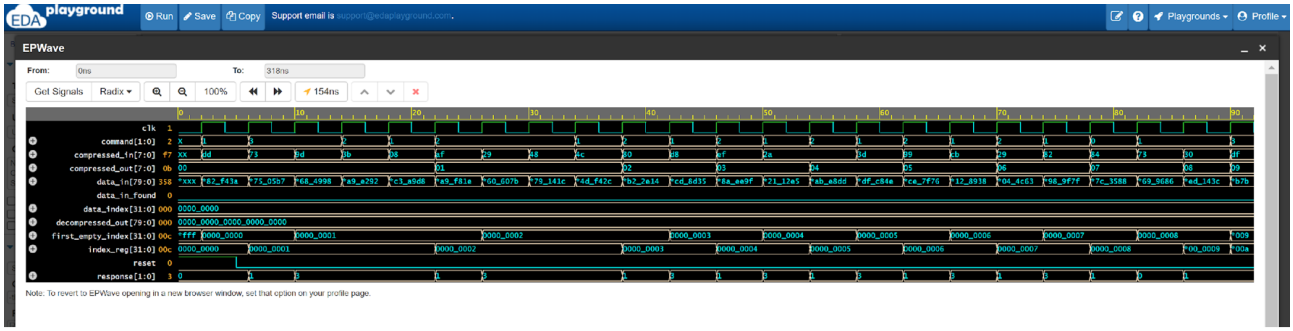

Figure 23: Simulation Waveform1

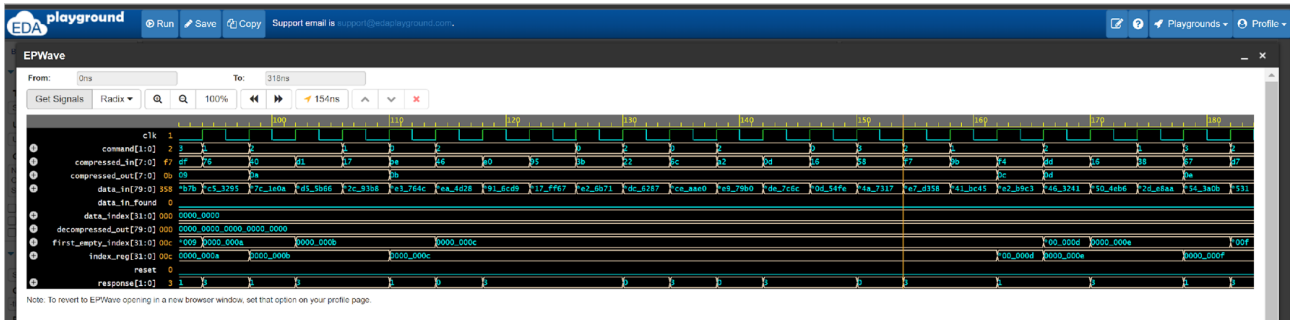

Figure 24: Simulation Waveform2

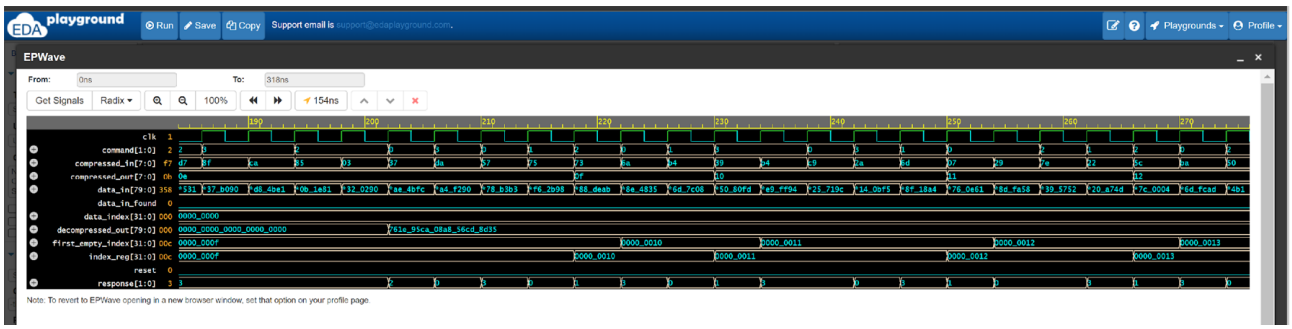

Figure 25: Simulation Waveform3

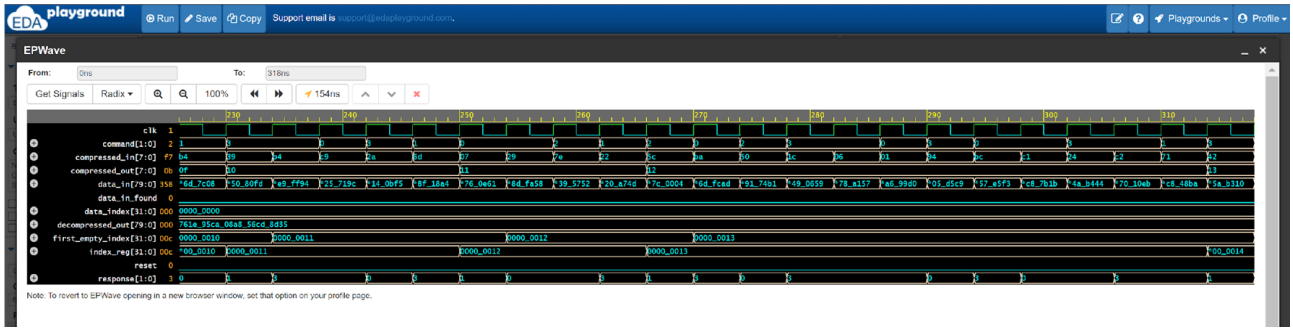

Figure 26: Simulation Waveform4

6 Functional Coverage

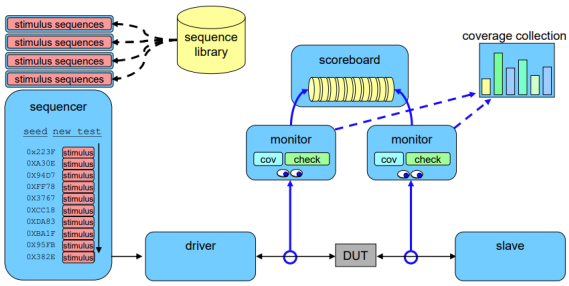

Figure 27: Coverage Driven Environment

Functional Coverage is a metric that measures the extent to which the design's functionality has been exercised and covered by the testbench or verification environment, as defined by the verification engineer through a functional coverage model. This model maps each design feature to a "cover point" with specific conditions, called "bins" . During simulation, as these bin conditions are hit, the coverage is tracked, and a report can be generated to analyze the verification progress and identify any gaps or untested scenarios. Covergroups can contain multiple cover points, and the collection of these covergroups forms the overall functional coverage model, providing a comprehensive assessment of the design's validation.[1]

6.1 Coverage Report

The coverage report is a critical output of the functional coverage analysis. It provides a detailed breakdown of the progress and completeness of the verification process. The report typically includes information such as the total number of cover points defined in the coverage model, the number of bins within each cover point, and the percentage of bins that have been hit during simulation. This data allows the verification team to identify areas of the design that have been thoroughly tested, as well as any gaps or untested scenarios that require additional attention.

6.2 Coverage Results

In our project, it was essential to achieve at least 80% coverage results, as this is a critical measure of the design's functionality being exercised by the testbench.Our results, as depicted in the figures, demonstrate that we have successfully achieved this target with overall coverage of 84.04%. Functional coverage is tracked through the coverage model, which maps design features to "cover points" with specific conditions called "bins" As these bins are hit during simulation, the coverage is recorded and reported, providing a comprehensive assessment of the verification process.

```
SCB:: item recived
----------------------------------------------------------------
UVM_INFO chip_scoreboard.sv(89) @ 3275: uvm_test_top.env.chip_scb [chip_scoreboard] **Test success - invalid operation case : compressed_out=0x0 decompressed_out=0x15 response=0x3
----------------------------------------------------------------
UVM_INFO chip_sequence.sv(16) @ 3275: uvm_test_top.env.chip_agnt.sequencer@@seq [chip_sequence] Base seq: Inside Body
UVM_INFO chip_driver.sv(28) @ 3275: uvm_test_top.env.chip_agnt.driver [DRV] Wait for item from sequencer
UVM_INFO chip_monitor.sv(83) @ 3285: uvm_test_top.env.chip_agnt.monitor [chip_monitor] Monitor found a packet **chip_sequence_item: inputs --> command=0x0 data_in=0xa compressed_in=0x31, outputs --> compressed_out=0x0 decompressed_out=0x0 response=0x0
UVM_INFO chip_sequence.sv(16) @ 3285: uvm_test_top.env.chip_agnt.sequencer@@seq [chip_sequence] Base seq: Inside Body
UVM_INFO chip_driver.sv(28) @ 3285: uvm_test_top.env.chip_agnt.driver [DRV] Wait for item from sequencer
UVM_INFO chip_monitor.sv(91) @ 3295: uvm_test_top.env.chip_agnt.monitor [chip_monitor] Monitor found a packet **chip_sequence_item: inputs --> command=0x0 data_in=0xa compressed_in=0x31, outputs --> compressed_out=0x0 decompressed_out=0x15 response=0x0  after output set
----------------------------------------------------------
Name                Type                Size  Value
----------------------------------------------------------
item                chip_sequence_item  -     @4995
  command           integral            2     'h0
  data_in           integral            80    'ha
  compressed_in     integral            8     'h31
  compressed_out    integral            8     'h0
  decompressed_out  integral            80    'h15
  response          integral            2     'h0
----------------------------------------------------------
UVM_INFO chip_sequence.sv(16) @ 3295: uvm_test_top.env.chip_agnt.sequencer@@seq [chip_sequence] Base seq: Inside Body
UVM_INFO chip_driver.sv(28) @ 3295: uvm_test_top.env.chip_agnt.driver [DRV] Wait for item from sequencer
 Coverage = 83.59 %
```

Figure 28: Sample Coverage Report

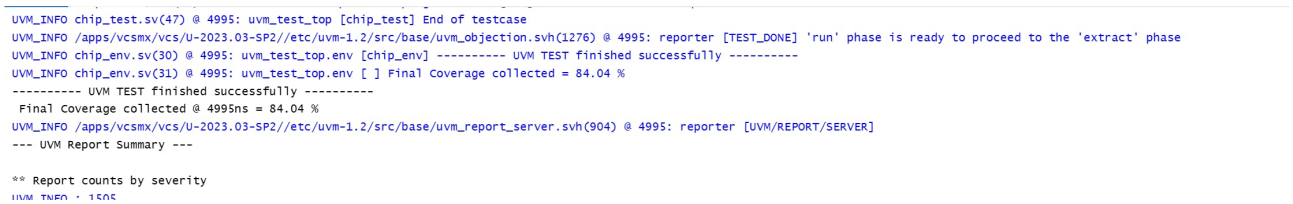

Figure 29: Final Coverage Result

7 Instructions to run the simulation

To run our SystemVerilog model in the EDA Playground environment, we configure several settings . First, under the "Testbench + Design" section the 'SystemVerilog/Verilog' is selected. Then, for the 'UVM / OVM' setting, select 'UVM 1.2' from the dropdown menu to utilize the UVM (Universal Verification Methodology) which provides a standardized methodology for verifying integrated circuit designs.

Next, for "Tools & Simulators" we used 'Synopsys VCS 2023.03' as your simulator. This simulator is well-suited for running complex verification environments like UVM due to its performance and debugging capabilities.

Finally, to view our results visually and analyze waveforms post-simulation, the 'Open EPWave after run' option is checked under "Run Options". This will automatically open the waveform viewer with the simulation output.

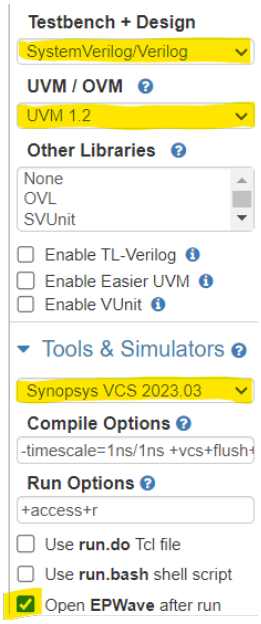

Figure 30: Run Instructions for EDA Platform

8 Conclusion

In conclusion, our project has successfully met its objectives by developing and verifying a chip capable of efficiently processing data compression and decompression using a dictionary-based algorithm. Throughout the project's three phases: Reference Model, Verification Plan, and UVM Verification Environment with Coverage, We have constructed a robust system that follows the initial specifications and performs as intended under various test scenarios. The successful simulation results and detailed demonstrations validate the functionality and reliability of the chip.

References

[1] *coverage*. Accessed: 2024-06-11. URL: https://www.chipverify.com/systemverilog/systemverilog-functional-coverage.

[2] *Lecture Slides*. Accessed: 2024-06-10. URL: https://ritaj.birzeit.edu/bzu-msgs/attach/2605114/Testbench+Structure.pdf.

[3] *scoreboard*. Accessed: 2024-06-19. URL: https://verificationguide.com/uvm/uvm-scoreboard-example/.

[4] *Universal Verification Methodology*. Accessed: 2024-06-08. URL: https://verificationguide.com/uvm/.

[5] *UVM Agent*. Accessed: 2024-06-08. URL: https://www.chipverify.com/uvm/uvm-agent#google_vignette.

9 Appendices

9.1 Appendix A: design.sv

```
module compression_decompression
  #( parameter DICTIONARY_DEPTH = 256 )(

    //input Ports
    input logic         clk,
    input logic         reset,
    input logic [ 1:0] command,
    input logic [79:0] data_in,       // 80 bit Input Data to be compressed
    input logic [ 7:0] compressed_in, // 8 bit Input Data to be decompressed

    // Output Ports
    output logic [ 7:0] compressed_out,    // 8 Output Output compressed
       data
    output logic [79:0] decompressed_out,
    output logic [ 1:0] response           // the status of the output
);


  logic [79:0] dictionary_memory[DICTIONARY_DEPTH-1:0];  // chips
     internal memory
  logic [31:0] index_reg;
  bit data_in_found;
  int data_index;
  int first_empty_index;


  always @(posedge clk or posedge reset) begin

    first_empty_index = find_first_empty();  // temp variable so we use
       blocking.

    // Either we reset the design.
    if (reset == 1) begin
      foreach (dictionary_memory[i]) dictionary_memory[i] <= 0;
      index_reg <= 0;
      compressed_out <= 0;
      decompressed_out <= 0;
      response <= 0;
      data_in_found <= 0;

      // Or we check if there is data in the dictionary_memory.
    end else begin
      data_in_found = 0;  // Initialize to 0 before the loop
      data_index = 0;     // Initialize data_index
      for (int i = 0; i < DICTIONARY_DEPTH; i++) begin
        if (dictionary_memory[i] == data_in) begin
          data_in_found = 1;
```

```
45            data_index = i;  // Store the index of the found element
46            break;
47          end
48        end
49
50        case (command)
51          2'b00:  /*No operation*/
52          begin
53            response <= 2'b00;
54          end
55          2'b01:  /*Compression*/
56          begin
57            if (data_in_found == 1) begin
58              compressed_out <= data_index;
59              response <= 2'b01;
60            end else begin
61              if (index_reg == DICTIONARY_DEPTH) begin
62                response <= 2'b11;
63              end else begin
64                dictionary_memory[first_empty_index] <= data_in;
65                compressed_out <= first_empty_index;
66                index_reg <= index_reg + 1;
67                response <= 2'b01;
68              end
69            end
70          end
71          2'b10:  /*Decompression*/
72          begin
73            if (compressed_in <= first_empty_index) begin
74              decompressed_out <= dictionary_memory[compressed_in];
75              response <= 2'b10;
76            end else response <= 2'b11;
77          end
78          2'b11:  /*Invalid command, report an error*/
79          response <= 2'b11;  // error
80
81        endcase
82
83      end
84    end
85
86    // Function to find the index of the first empty element
87    function int find_first_empty;
88      for (int i = 0; i < DICTIONARY_DEPTH; i++) begin
89        if (dictionary_memory[i] == 0) begin
90          return i;
91        end
92      end
93      // If no empty element is found, return -1
94      return -1;
95    endfunction
```

```
endmodule
```

9.2 Appendix B: chip_if.sv

```
/* Interface
   defines an interface for the DUT signals.
*//*************************************************************/

interface chip_if(input logic clk, reset);
    logic [1:0] command;
    logic [79:0] data_in;
    logic [7:0] compressed_in;

    logic [7:0] compressed_out;
    logic [79:0] decompressed_out;
    logic [1:0] response;
endinterface
```

9.3 Appendix C: testbench.sv

```
/* testbench Item
  - the top module that contains the DUT and interface.
  - This module starts the test.
*//*************************************************************/
import uvm_pkg::*;
`include "uvm_macros.svh"
`include "chip_if.sv"
`include "chip_test.sv"

module tb_top;

  //clock and reset signals
  bit clk;
  bit reset;

  //reset Generation - reset the memory and index register each run
  initial begin
    reset = 1;
    #5
    reset = 0;
  end

  //clock generation
  always #5 clk = ~clk;

  //interface instance
  chip_if vif(clk, reset);

  //DUT instance
  compression_decompression DUT (
```

```
    .clk(vif.clk),
    .reset(vif.reset),
    .command(vif.command),
    .compressed_in(vif.compressed_in),
    .data_in(vif.data_in),
    .compressed_out(vif.compressed_out),
    .decompressed_out(vif.decompressed_out),
    .response(vif.response)
  );

  initial begin
    //set interface in config_db
    uvm_config_db#(virtual chip_if)::set(null, "*", "chip_if", vif);

    //calling test
    run_test("chip_test");
  end


  initial begin
    //dump waves
    $dumpfile("dump.vcd");

    //dump all variables in the testbench hierarchy
    $dumpvars;
  end

endmodule
```

9.4 Appendix D: chip_sequence_item.sv

```
/* Sequence Item
   Contains the fields for the input and output signals of the DUT
*//*************************************************************/

class chip_sequence_item extends uvm_sequence_item;

  //input fields
  rand bit [ 1:0] command;
  rand bit [79:0] data_in;
  rand bit [ 7:0] compressed_in;

  //output fields
  bit [ 7:0] compressed_out;
  bit [79:0] decompressed_out;
  bit [ 1:0] response;


  //field automation for randomization and printing
  `uvm_object_utils_begin(chip_sequence_item)
```

```
    `uvm_field_int(command, UVM_DEFAULT)
    `uvm_field_int(data_in, UVM_DEFAULT)
    `uvm_field_int(compressed_in, UVM_DEFAULT)
    `uvm_field_int(compressed_out, UVM_DEFAULT)
    `uvm_field_int(decompressed_out, UVM_DEFAULT)
    `uvm_field_int(response, UVM_DEFAULT)
  `uvm_object_utils_end


  //add constraint on the random generated data_in
  //we minimized the data_it to 32 in decimal max, in order to get
     duplicate generated data_in, to test all the
  //test scenarios
  constraint value_data_in { data_in inside {[0:2**5]}; }
  constraint value_compressed_in {compressed_in inside {[0:50]};}


  //constructor
  function new(string name = "chip_sequence_item");
    super.new(name);
  endfunction


  //print generated data
  virtual function string convert2str();
    return $sformatf ( "**chip_sequence_item: inputs --> command=0x%0h
       data_in=0x%0h compressed_in=0x%0h, outputs --> compressed_out=0x%0h
       decompressed_out=0x%0h response=0x%0h ", command, data_in,
       compressed_in, compressed_out,decompressed_out,response );
  endfunction

endclass
```

9.5 Appendix E: chip_sequence.sv

```
/* Sequence
*//*************************************************************/

class chip_sequence extends uvm_sequence#(chip_sequence_item);
  `uvm_object_utils(chip_sequence)

  //constructor
  function new(string name = "chip_sequence");
    super.new(name);
  endfunction

  chip_sequence_item req;

  //create, randomize and send the item to driver
  task body();
```

```
    `uvm_info(get_type_name(), "Base seq: Inside Body", UVM_LOW);
    `uvm_do(req);
  endtask

endclass
```

9.6 Appendix F:chip_sequencer.sv

```
/* Sequencer
*//*************************************************************/

class chip_sequencer extends uvm_sequencer#(chip_sequence_item);
  `uvm_component_utils(chip_sequencer)

  //constructor
  function new(string name, uvm_component parent);
    super.new(name,parent);
  endfunction

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
  endfunction

endclass
```

9.7 Appendix G: chip_driver.sv

```
/* Driver
   uses the sequence item to drive the DUT
*//*************************************************************/

class chip_driver extends uvm_driver #(chip_sequence_item);
  `uvm_component_utils(chip_driver)

  //constructor
  function new (string name="chip_driver", uvm_component parent=null);
    super.new(name, parent);
  endfunction

  //virtual Interface to the DUT
  virtual chip_if vif;

  //build phase to get the virtual interface
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    if(!uvm_config_db#(virtual chip_if)::get(this, "", "chip_if", vif))
      `uvm_fatal("DRV" , "Virtual interface not found");
  endfunction

  //main run task to drive the DUT
```

```
  task run_phase(uvm_phase phase);
    //drive normal traffic
    forever begin
      seq_item_port.get_next_item(req); //get the next sequence item
      `uvm_info("DRV", $sformatf("Wait for item from sequencer"), UVM_LOW)

      //drive the req - apply inputs to the DUT -  transaction level to
         signal level
      vif.command <= req.command;
      vif.data_in <= req.data_in;
      vif.compressed_in <= req.compressed_in;

      seq_item_port.item_done(); //sequence item has been processed
    end
  endtask

endclass
```

9.8 Appendix H: chip_monitor.sv

```
/* Monitor
*//*************************************************************/

class chip_monitor extends uvm_monitor;
  `uvm_component_utils(chip_monitor)

  //virtual Interface
  virtual chip_if vif;
  //analysis port, to send the transaction to scoreboard
  uvm_analysis_port #(chip_sequence_item) mon_analysis_port;
  chip_sequence_item item;


  //define the coverage group
  real Gobind_Coverage;
  covergroup my_covergroup;
    option.per_instance = 1;

    cmd_cp: coverpoint item.command {
      bins commands[] = {[0:3]};
    }

    data_in_cp: coverpoint item.data_in {
      bins msb = {[2**79:2**80-1]};
      bins middle = {[2**39:2**40-1]};
      bins lsb = {[0:1]};
      bins random_samples = {1000000};
    }

    compressed_in_cp: coverpoint item.compressed_in {
```

```
31        bins ranges[] = {[0:'hFF]};
32      }
33
34      compressed_out_cp: coverpoint item.compressed_out {
35        bins ranges[] = {[0:'hFF]};
36        bins others = default;
37      }
38
39      decompressed_out_cp: coverpoint item.decompressed_out {
40        bins msb = {[2**79:2**80-1]};
41        bins middle = {[2**39:2**40-1]};
42        bins lsb = {[0:1]};
43      }
44
45      response_cp: coverpoint item.response {
46        bins responses = {[0:3]};
47      }
48    endgroup;
49
50
51    //constructor
52    function new (string name="chip_monitor", uvm_component parent=null);
53      super.new(name, parent);
54      mon_analysis_port = new("mon_analysis_port", this);
55      item = new();
56      my_covergroup = new();
57    endfunction
58
59
60    //build phase to get the virtual interface
61    function void build_phase(uvm_phase phase);
62      super.build_phase(phase);
63      if(uvm_config_db#(virtual chip_if)::get(this, "", "chip_if", vif) ==
          null)
64        `uvm_fatal("MON","Could not get virtual interface");
65    endfunction
66
67
68    //run phase to monitor the DUT signals
69    task run_phase(uvm_phase phase);
70      forever begin
71        item = chip_sequence_item::type_id::create("item");
72        wait(!vif.reset);
73
74        //wait for a clock edge
75        @ (posedge vif.clk);
76
77        //capture the DUT signals
78        item.command = vif.command;
79        item.data_in = vif.data_in;
80        item.compressed_in = vif.compressed_in;
```

```
81
82        //reset the prevoius outputs
83        item.compressed_out = 0;
84        item.decompressed_out = 0;
85        item.response = 0;
86
87        //write the transaction to the analysis port
88        `uvm_info(get_type_name(), $sformatf("Monitor found a packet
             %s",item.convert2str()), UVM_LOW)
89
90        //write the results on the output ports
91        @ (posedge vif.clk);
92        item.compressed_out = vif.compressed_out;
93        item.decompressed_out = vif.decompressed_out;
94        item.response = vif.response;
95
96        `uvm_info(get_type_name(), $sformatf("Monitor found a packet %s after
             output set",item.convert2str()), UVM_LOW)
97
98        item.print(); // factory print method
99        @(posedge vif.clk);
100       my_covergroup.sample(); // method for sampling coverage
101       Gobind_Coverage = my_covergroup.get_inst_coverage();
102       $display (" Coverage = %0.2f %% \n",
             my_covergroup.get_inst_coverage());
103
104       mon_analysis_port.write(item);
105       end
106   endtask
107
108 endclass
```

9.9 Appendix I:chip_agent.sv

```
1  /* Agent
2     The agent encapsulates the driver, monitor, and sequencer */
3  /*************************************************************/
4  class chip_agent extends uvm_agent;
5    `uvm_component_utils(chip_agent)
6
7    //constructor
8    function new (string name="chip_agent", uvm_component parent=null);
9      super.new(name, parent);
10   endfunction
11
12   //declaring agent components
13   chip_driver    driver;
14   chip_monitor   monitor;
15   chip_sequencer sequencer;
16
```

```
  //build_phase
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    if(get_is_active == UVM_ACTIVE) begin
      driver    = chip_driver::type_id::create("driver", this);
      sequencer = chip_sequencer::type_id::create("sequencer", this);
    end
      monitor   = chip_monitor::type_id::create("monitor", this);
  endfunction

  //connect_phase - connecting the driver and sequencer port
  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    if(get_is_active == UVM_ACTIVE) begin
      driver.seq_item_port.connect(sequencer.seq_item_export);
    end
  endfunction

endclass
```

9.10 Appendix J:chip_scoreboard.sv

```
/* Scoreboard
   - receives transactions from the monitor
   - perform checks to ensure the DUT is functioning correctly.
*//*************************************************************/

class chip_scoreboard extends uvm_scoreboard;
  `uvm_component_utils(chip_scoreboard)

  chip_sequence_item item;
  chip_sequence_item item_queue[$];
  //declaring port to recive packets from monitor
  uvm_analysis_imp#(chip_sequence_item, chip_scoreboard) m_analysis_imp;


  //constructor
  function new (string name="chip_scoreboard", uvm_component parent=null);
    super.new(name, parent);
        m_analysis_imp = new("m_analysis_imp", this);
  endfunction


  //build_phase - create port and initialize local memory
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
  endfunction


  //write task - recives the pkt from monitor and pushes into queue
```

```
29  function void write(chip_sequence_item item);
30    item_queue.push_back(item); //new added
31  endfunction
32
33
34  task run_phase (uvm_phase phase);
35    forever begin
36      wait(item_queue.size > 0);
37      if(item_queue.size > 0) begin
38        item = item_queue.pop_front();
39        $display("SCB:: item recived");
40        $display("-----------------------------------------------------------------------"
41
42        //no operation case
43        if (item.command == 2'b00) begin
44          if (item.response != 2'b00)
45            `uvm_error("SB", $sformatf("**Test failed - no operation case"))
46          else
47            `uvm_info(get_type_name(), $sformatf("**Testsuccess - no
                operation case : compressed_out=0x%0h decompressed_out=0x%0h
                response=0x%0h", item.compressed_out, item.decompressed_out,
                item.response), UVM_LOW)
48        end
49
50        //---------------------------------------
51        //compression case - add more test cases
52        else if (item.command == 2'b01) begin
53          //compression success
54          if (item.response == 2'b01)
55            `uvm_info(get_type_name(), $sformatf("**Test success -
                compression case : compressed_out=0x%0h
                decompressed_out=0x%0h response=0x%0h", item.compressed_out,
                item.decompressed_out, item.response), UVM_LOW)
56
57          //if data_in is not in the memory and memory is full must
                generate error
58            else if (item.response == 2'b11 && item.decompressed_out==0 )
59            `uvm_info(get_type_name(), $sformatf("**Test success -
                compression case data_in is not in the memory and memory is
                full, DUT generates error : compressed_out=0x%0h
                decompressed_out=0x%0h response=0x%0h", item.compressed_out,
                item.decompressed_out, item.response), UVM_LOW)
60
61          //any other cases - error in the DUT
62          else
63            `uvm_error("SB", $sformatf("**Test failed - compression case"))
64        end
65
66
67        //---------------------------------------
68        //decompression case
```

```
69          else if(item.command == 2'b10) begin
70            //decompression success
71            if (item.response == 2'b10)
72              `uvm_info(get_type_name(), $sformatf("**Test success -
                    decompression case : compressed_out=0x%0h
                    decompressed_out=0x%0h response=0x%0h", item.compressed_out,
                    item.decompressed_out, item.response), UVM_LOW)
73
74            //if decompression index does not have data in the memory -> must
                 generate error
75            else if (item.response == 2'b11)
76              `uvm_info(get_type_name(), $sformatf("**Test success -
                    decompression case, item not in memory -> DUT generates
                    error response : compressed_out=0x%0h decompressed_out=0x%0h
                    response=0x%0h", item.compressed_out, item.decompressed_out,
                    item.response), UVM_LOW)
77
78            //any other cases - error in the DUT
79            else
80              `uvm_error("SB", $sformatf("**Test failed - decompression
                    case"))
81          end
82
83          //---------------------------------------
84          //invalid operation case
85          else if(item.command == 2'b11) begin
86            if (item.response != 2'b11)
87              `uvm_error("SB", $sformatf("**Test failed - invalid operation
                    case"))
88            else
89              `uvm_info(get_type_name(), $sformatf("**Test success - invalid
                    operation case : compressed_out=0x%0h decompressed_out=0x%0h
                    response=0x%0h", item.compressed_out, item.decompressed_out,
                    item.response), UVM_LOW)
90          end
91
92          $display("------------------------------------------------------------------");
93        end
94      end
95    endtask
96
97  endclass
```

9.11 Appendix K:chip_env.sv

```
1  /* Environment
2     defines the environment to instantiate and connect the components.
3  *//*****************************************************/
4  class chip_env extends uvm_env;
5    `uvm_component_utils(chip_env)
```

```
6
7   //constructor
8   function new(string name="chip_env", uvm_component parent=null);
9     super.new(name, parent);
10  endfunction
11
12  chip_agent      chip_agnt;
13  chip_scoreboard chip_scb;
14
15  //build_phase - crate the components
16  function void build_phase(uvm_phase phase);
17    super.build_phase(phase);
18    chip_agnt = chip_agent::type_id::create("chip_agnt", this);
19    chip_scb  = chip_scoreboard::type_id::create("chip_scb", this);
20  endfunction
21
22
23  //connect_phase - connecting monitor and scoreboard port
24  function void connect_phase(uvm_phase phase);
25    chip_agnt.monitor.mon_analysis_port.connect(chip_scb.m_analysis_imp);
26  endfunction
27
28  function void report_phase(uvm_phase phase);
29    super.report_phase(phase);
30    `uvm_info(get_type_name(), "---------- UVM TEST finished successfully
        ---------- ", UVM_NONE)
31    `uvm_info(" ", $sformatf("Final Coverage collected = %0.2f %% ",
        chip_agnt.monitor.Gobind_Coverage), UVM_NONE)
32    $display ("---------- UVM TEST finished successfully ----------");
33    $display (" Final Coverage collected @ %0tns = %0.2f %% ",$time,
        chip_agnt.monitor.Gobind_Coverage);
34 endfunction
35
36 endclass
```

9.12 Appendix L:chip_test.sv

```
1  /* Test
2  *//*************************************************************/
3  `include "chip_sequence_item.sv"
4  `include "chip_sequence.sv"
5  `include "chip_sequencer.sv"
6  `include "chip_driver.sv"
7  `include "chip_monitor.sv"
8  `include "chip_scoreboard.sv"
9  `include "chip_agent.sv"
10 `include "chip_env.sv"
11
12 class chip_test extends uvm_test;
13   `uvm_component_utils(chip_test)
```

```
14
15    //constructor
16    function new(string name = "chip_test",uvm_component parent=null);
17      super.new(name,parent);
18    endfunction
19
20    virtual chip_if vif;
21    chip_env env;
22    chip_sequence seq ;
23
24    //build phase
25    function void build_phase(uvm_phase phase);
26      super.build_phase(phase);
27      env = chip_env::type_id::create("env", this);
28      if(!uvm_config_db#(virtual chip_if)::get(this, "", "chip_if", vif))
29        `uvm_fatal("TEST","Could not get virtual interface");
30      uvm_config_db#(virtual chip_if)::set(this, "env.chip_agnt.*",
          "chip_if", vif);
31    endfunction
32
33
34    //run phase
35    task run_phase(uvm_phase phase);
36      phase.raise_objection(this);
37      seq = chip_sequence::type_id::create("seq");
38
39      //generate 500 test cases
40      repeat(500) begin
41        @ (posedge vif.clk);
42        seq.randomize();
43        seq.start(env.chip_agnt.sequencer);
44      end
45
46      phase.drop_objection(this);
47      `uvm_info(get_type_name, "End of testcase", UVM_LOW);
48    endtask
49
50  endclass
```