Assertions-based Coverage Tool

A thesis submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

In Electronics and Communication Engineering

at Alexandria University

Under the supervision of

Prof. Dr. Mohamed Rizk, Alexandria University

Dr. Khaled Salah, Mentor Graphics Egypt

Alexandria University, Egypt, July 2016

ACKNOWLEDGEMENT

The project in itself is an acknowledgement of the inspiration, guidance and the technical assistance contributed to it by many people. It would not have been possible without the help received from them.

First and foremost, we would like to convey our sincere gratitude and deepest regards to our guide **Dr. Khaled Salah, Mentor Graphics, Egypt** who has been the continuous driving force behind this work. We thank him wholeheartedly for giving us the opportunity to work with him by trusting our credentials and capabilities, and helping us to explore our potential to the fullest.

We are thankful to **Prof. Dr.** ***Mohamed Rizk*****,** Department, Electronics and Communication Engineering, for permitting us to use the facilities available in the department to carry out the project successfully, helping us and allowing us to execute our project in **Virginia Tech - MENA** labs. .

Finally, we are grateful for all the encouragement and support we got whether from our teaching staff or our colleagues.

Team Members

Hatem Mohamed El-Kharashy
01063384085
hatemelkharashy@gmail.com

Abdel-Rahman Gaber Abu-Bakr
01116825274
eng.abdelrahman.gaber@gmail.com

Abstract

Over the past decade, the demand for a justified complete hardware functional verification process reached its peak. Throughout the design process, it is obligatory to run in parallel a verification process that may consume more time and money than the actual design process. The verification environments are usually hard to implement and time consuming.

With the pressure of time-to-market, the need for a fast and reliable verification procedure is at-most. Now, one of the most important parameters the industry seeks in the verification process is the coverage closure, since it is the one to justify when the verification process has to stop. Here, we are proposing an environment based on regression tests, assertions for protocol checking and a novel algorithm for functional coverage based on assertion technique to calculate the coverage percentages. This environment offers a fast and reliable solution through a GUI interface with optimized features that even designers might find it handy to shorten the long verification process when the deadline is hard to meet.
.

Table of Contents

LIST OF FIGURES

1 Introduction

1.1 Overview of the Problem

In a recent study by Harry Foster, Chief Scientist at Mentor Graphics, he showed that about 20% of the design projects in the industry have 70% of the project time spent in the verification processes. The increasing curve of verification need in the design process and the proof of its importance drove the attention to handling the designers more verification duties than the ordinary regression testing, as the verification environment would cover aspects as the power, clock edges, layered sequences, security and even hardware/software co-verification.

Assertions-Based Verification (ABV) has recently been extensively used to verify certain design parts comprehensively due to its simplicity and capabilities in dealing with sequences and different protocols. Assertion is a powerful tool when used to verify protocols especially when wrapped through an environment with a running regression tests.

With the increase of complexity in System-on-Chip (SoC) design, the time-to-market applies high pressure to run both verification and design processes in parallel to meet the rough deadlines. Since assertions are simple and reliable, designers can easily run their regression tests through them and complete another step towards a complete verification process.

The whole idea of a complete functional testing is entirely based on reaching a closure coverage, implying the importance of coverage percentages in the verification process. The straight forward thinking was to include two powerful parameters in one environment wrapped together to reach a higher reliability in the verification results, and at the same time as simple as a designer could implement and verify the work when rough deadlines are at the post.

1.2 Goal

Any proposed environment should have two main targets: assertions to verify the protocol, and coverage groups telling when the verification is finished. In this paper, we use assertions to cover the input signals and transitions which are basically what functional coverage do in any verification environment. Using SystemVerilog Assertions (SVA), we could also cover sequences, which gives more capability to the verification process. With this approach, Assertions-Based Verification would easily maintain the coverage results, which even may control regression random sequences until reaching full coverage.

We propose here not just the methodology, but also a complete GUI-based tool, which provides some interesting features such as a colored coverage percentages, and automatic generation of assertions. The tool was implemented using Python scripts, and it can be fully automated as the user can input the module signals, bins to be covered and test type, then the tool generates the assertions file to be binded with the Design Under Test (DUT), run the test and generate the results.

1.3 Structure

The thesis has been divided into seven chapters including this one.

Chapter 1 introduces the project and motivation behind it.

Chapter 2 deals with the state of the art of the essentials of the project.

The third chapter presents the proposed methodology and its importance.

Chapter 4 shows the SD-Card controller as a case study, which will be later used to test the tool.

Chapter 5 presents the Tool implementation details.

Chapter 6 presents the results and the verification methods in the project.

Conclusion and future scopes are proposed in chapter 7.

2 Verification Background

2.1 Introduction

Silicon chip technology powers many of today’s innovations and newest products. Rapid technology advancement fuels ever-increasing chip complexity, which in turn enables the latest round of amazing products. These chips touch many aspects of our everyday life, from the pocket-sized combination cell phone, and digital camera to the high-end server that searches an online bookseller’s database, verifies credit card funds, and sends the order to the warehouse for immediate delivery. [1, p.5]

Expectations for these chips grow at an equal rate, despite the additional complexity. For example, consumers do not expect the chips that monitor safety processes in our cars to fail during the normal life of the vehicle. Nor is it acceptable to be denied access to our on-line accounts because the server is down. Thus, it has become a major engineering feat to design these silicon chips correctly. [1, p.5]

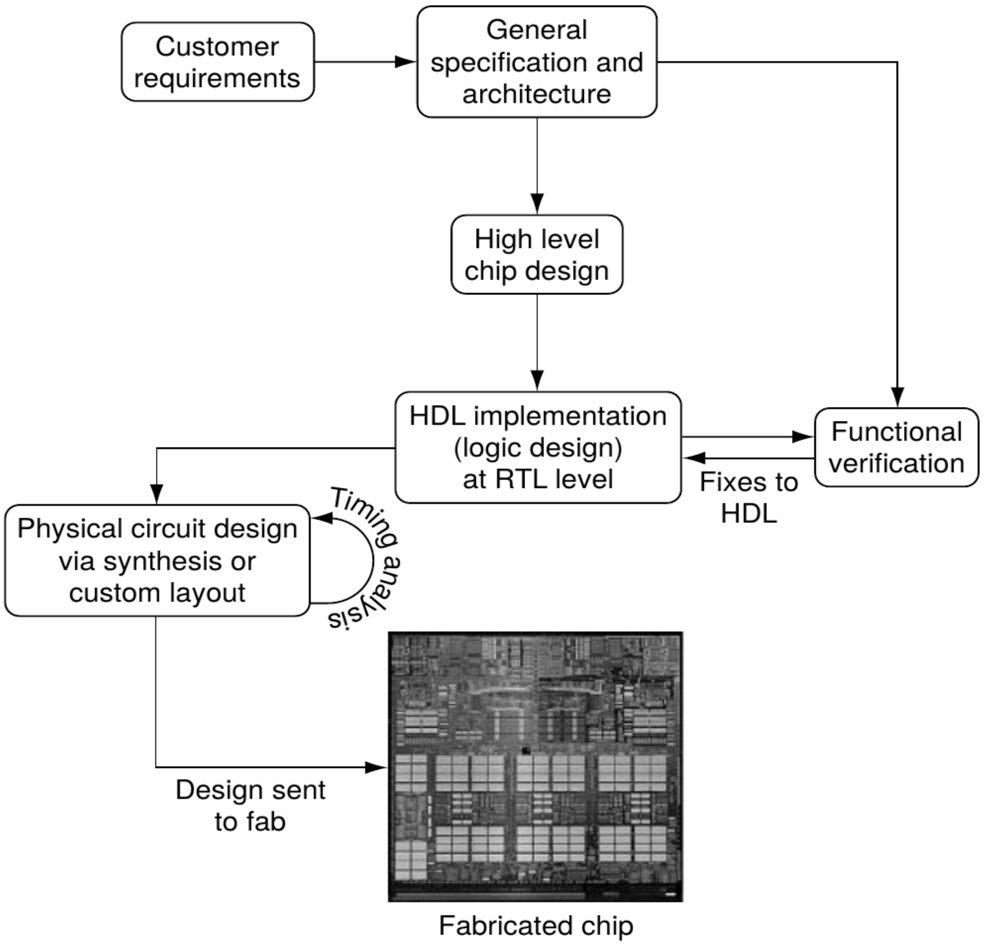

Customer requirements
General specification and architecture
High level chip design
HDL implementation (logic design) at RTL level
Functional verification
Fixes to HDL
Physical circuit design via synthesis or custom layout
Timing analysis
Design sent to fab
Fabricated chip

Fig 2.1 Chip design process [1, p.6]

An enormous amount of engineering goes into each of these chips, whether it is a microprocessor, memory device, or entire system on a chip (SoC). All types of chips have a set of challenges that the engineers must solve for the final product to be successful in the marketplace.

Figure 2.1 shows a chip's design flow, which starts with the customer's requirements. Those requirements include function, chip size, chip power consumption, and processing speed. Requirements are compiled and prioritized during the high-level design phase, in which designers architect the chip's main internal components and goals. As the chip design process progresses, engineers face challenges to balance goals that often conflict, for example: higher chip performance may require faster cycle times, but that will raise power consumption; faster cycle times may require adding more stages in the logic, but that increases chip complexity and area. [1. p.6]

After an engineer creates logic, written in a hardware description language (HDL), design automation tools can synthesize that logic into appropriate gates that correspond to the original design. But two questions remain:
1- what if the HDL did not express the correct function in the first place ?
2- what if the designer missed a critical corner condition?
Detecting incorrect function has become one of the trickiest challenges in the entire chip design process. This is the challenge of functional verification. [1, p.7]

This chapter introduces verification concepts, starting with defining Functional Hardware Verification, to introducing Assertions and Coverage concepts, which form the basis of the proposed methodology.

2.2 Functional Hardware Verification Overview

In the early beginnings, electronic systems were designed directly at the transistor level by hand, but due to the increasingly complexity of electronic circuits since the 1970s [2], it became unpractical to design the core logic directly at the transistor level, so circuit designers started to develop new ways to describe circuit functionality independently of the electronic technology used. The result was the Hardware Description Languages and the era of Electronic Design Automation was born.

Hardware description languages are languages that are used to define the behavior and the
structure of digital integrated circuits before they are translated into their own architecture. These kind of languages enable the modeling of a circuit for posterior simulation and, most importantly, translation into a lower level specification of physical electronic components. [3, p.5].

A hardware description language resembles a typical programming language consists in a textual description of expressions and control structures and although they both share some similarities, they are not the same. One main difference is that HDL code is translated concurrently, which is required in order to mimic hardware, and while programming languages, after compilation, are translated into low level instructions for the CPU to interpret, HDL specifications are translate to digital hardware, so using hardware description languages requires a different mindset than using programming languages.

Hardware description languages are tools that help engineers to easily specify abstract models of digital circuits to translate them into real hardware, but after the design is complete, another issue becomes noticeable: how can a designer know that the design works as intended? [3, p.6]

This brings up the need for verification. Verification is defined as a process to demonstrate the functional correctness of a design. [4, p.1] This process is done by the means of a testbench, an abstract system that provides stimuli to the inputs of the design under Test (DUT) and analyses its behavior. A basic verification environment is represented in the figure 2.2.

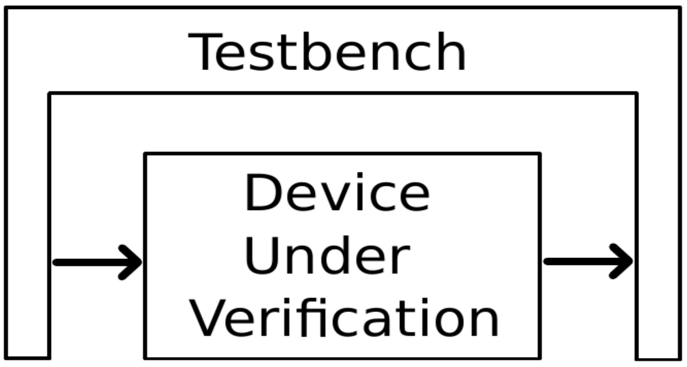


Fig 2.2 basic verification environment

One of the most common uses of a testbench is to show that a certain design implements the functionality defined in the specification. This task is known as functional verification. Normally, the testbench implements a model of the functionality that the designer wants to test and it is responsible to compare the results from that model with the results of the design under test. But it is important to take in account that functional verification can show that a design meets the specifications that have been verified but it cannot prove that whole design is correct. [4, p.2].

The traditional approach to verification relies on directed test. Verification engineers conceive a series of critical stimulus, apply them directly to the device under test (DUT) and check if the result is the expected one. This approach makes steady progress and produces quick initial results because it requires little effort for setting up the verification infrastructure. So given enough time, it maybe possible to write all the tests needed to cover 100% of the design. This scenario is represented in the figure 2.3. [5, p.5]

But as the design grows in size and complexity, this becomes a tedious and a time consuming task. Most likely, there will be not enough time to cover all the tests needed in a reasonable amount of time and there will be bugs that the verification engineer won’t be able to predict. So, random stimuli help to cover the unlikely cases.

However, in order to use random stimuli, there is the need of automating a process to generate them and there is also the need of a block that predicts, keeps track of the results and that analyses them: a scoreboard. Additionally, when using random stimulus, it will be needed to check what cases were covered by the generated stimuli, so the testbench will need functional coverage as well. Functional coverage is the process of measuring what space of inputs have been tested, what areas of the design have been reached and what states have been visited. [5, p.13]

This kind of testbench requires longer time to develop, causing an initial delay in the start of the verification process. However, random based testing can actually promote the verification of the design by covering cases not achieved with directed tests, as seen in the figure 2.3. [5, p.8]

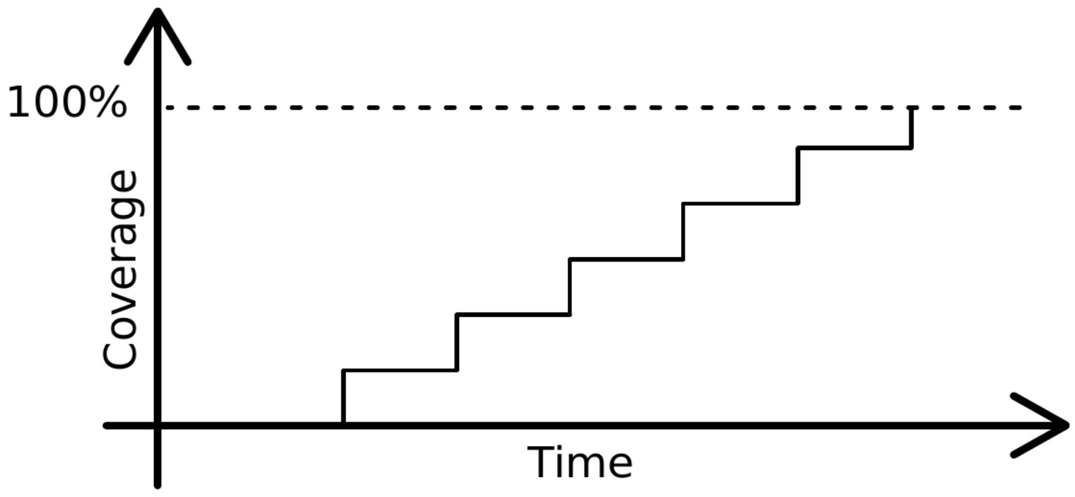


Fig 2.3 Direct testing progress [5, p.6]

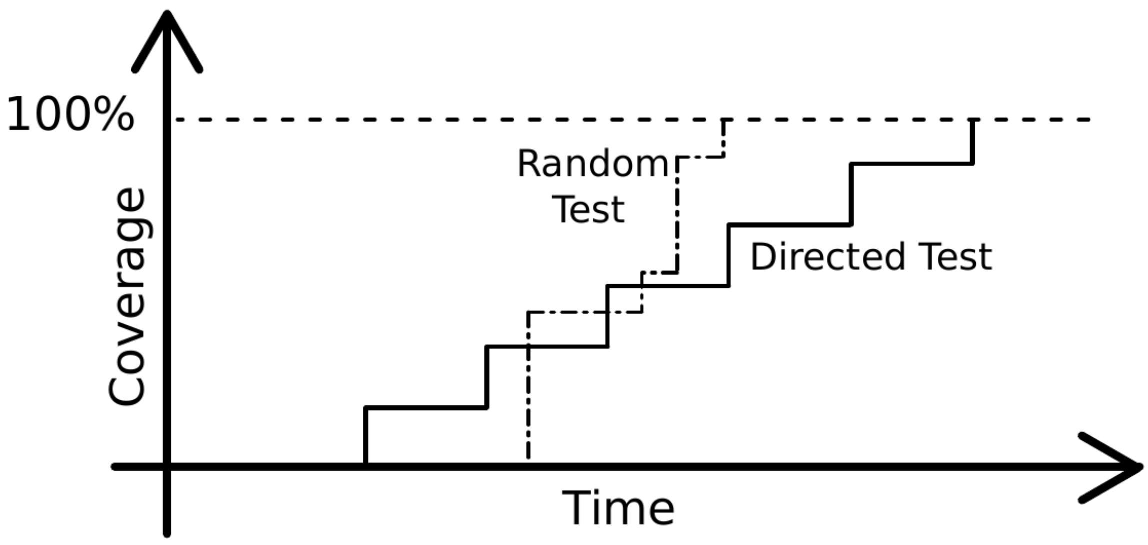


Fig 2.4 Random testing progress [5, p.8]

2.3 Observability and Controllability

Fundamental to the discussion of this chapter is the understanding of the concepts of controllability and observability[6,p4]. Informally, controllability refers to the ability to influence or activate an embedded finite state machine, structure, specific line of code, or behavior within the design by stimulating various input ports. Note that, while in theory a simulation testbench has high controllability of the design model's input ports during verification, it can have very low controllability of an internal structure within the model. In contrast, observability refers to the ability to observe the effects of a specific internal finite state machine, structure, or stimulated line of code.

Thus, a testbench generally has limited observability if it only observes the external ports of the design model (because the internal signals and structures are often indirectly hidden from the testbench).

To identify a design error using a simulation testbench approach, the following conditions must hold:

- The testbench must generate proper input stimulus to activate a design error.
- The testbench must generate proper input stimulus to propagate all effects resulting from the design error to an output port.
- The testbench must contain a monitor that can detect the design error that was first activated then propagated to a point for detection.

It is possible to set up a condition where the input stimulus activates a design error that does not propagate to an observable output port. In these cases, the first condition cited above applies; however, the second condition is absent, as illustrated in (Fig. 2.5).

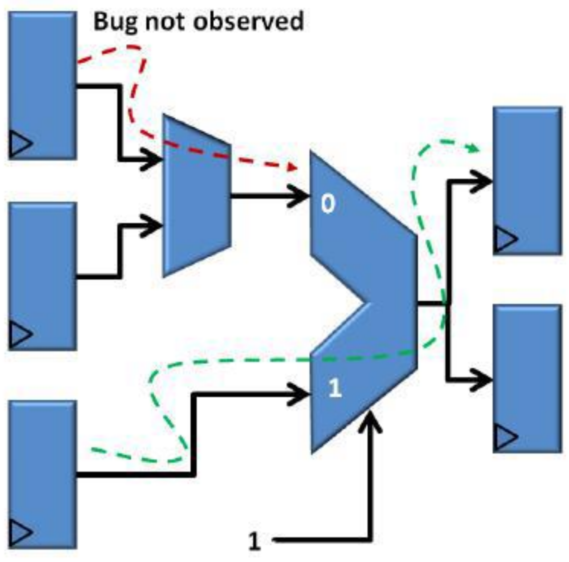


Fig 2.5 Bug not observed

2.4 Assertions

System Verilog Assertions or simply SVA [7] are checking conditions against one of the design specification. For example, we may want to write an assertion for a certain design specification that we don't want it to violate. If the assertion written is fired this mean there is a failure in this design specification.

For the following example we need when FRAME signal goes high, LDP signal goes low within two clock cycles.

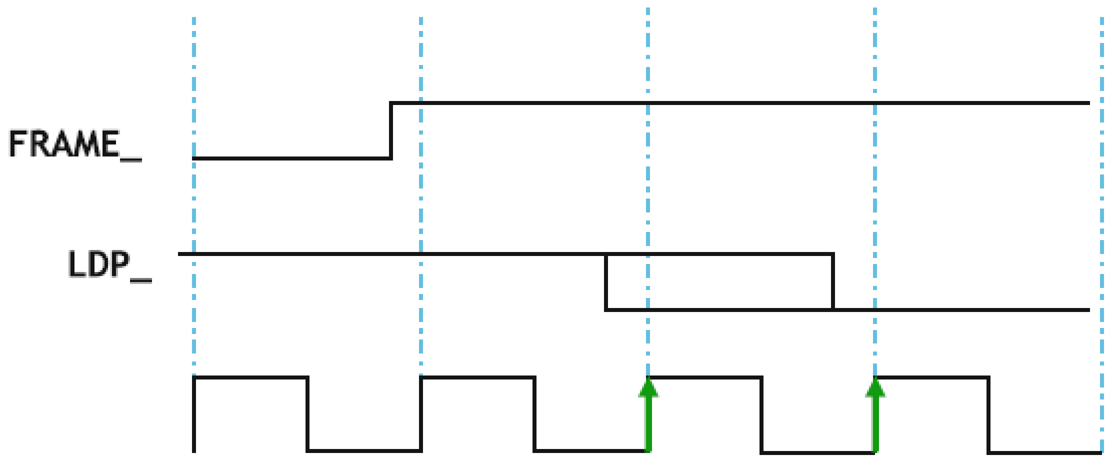


Fig 2.6 Protocol example

Such checking is essential to make sure that the design functions correctly.
The target of verification is to check that the design functions and works correctly according to the design specification and protocols. But we also want to increase the productivity in debugging, designing and simulating. System Verilog Assertions or simply SVA helps in those areas. SVA are easier to write than Verilog or System Verilog so they increase design productivity. They are easier to debug so they increase debugging productivity and they provide functional coverage and are faster to simulate.
Assertions will be discussed in details in the following sub sections but any assertion formula can be simplified on the following form

```
property NAME;
        Condition;
endproperty

assert property(NAME) else $display(msg);
```

Fig 2.7 Assertion General Form

As can be seen from the previous formula the assertion starts with property name, This defines a unique name for each assertion written. Next, A condition is written for the assertion and Finally the assertion ends with endproperty.
To use the written assertion we must use assert property and give it the assertion name that will be asserted. If the assertion failed a user written message can be displayed in terminal.

2.4.1 Advantages

- **shortens developing time:** As previously seen from the formula, Assertion writing is straight forward and self-explanatory. And as will be discussed in the following sections writing the condition for a certain design specification is simple and easy. This of course shortens the developing time as opposed to writing assertions using counterpart Verilog or system Verilog code that proves to be tedious.

The following example shows an assertion written using SVA and an assertion written using Verilog code for the case in (Fig. 2.7).

```
property CHECK;
        @(posedge clk) $rose(FRAME_) |-> ##[1:2] $fell(LDP_);
endproperty
assert property(CHECK) else $display("FAIL");
```

Fig 2.8 Assertion written using SVA

```
always @(posedge FRAME_)
begin : CHECK
          @(posedge clk);
          fork
          begin
                    @(negedge LDP_) disable CHECK;
          end
          begin
                    repeat(@posedge clk);
                    $display("FAIL");
                    disable CHECK;
          end
          join
end
```

Fig 2.9 Assertion written using Verilog code

From previous figures we can see that assertion using SVA is straight-forward. We begin with property name CHECK then a condition that fires only at the positive edge of the clock, then checks for the rise of FRAME_. The |-> means implies that the condition to the right is correct which is that LDP_ must fell within 2 clock cycles.

Compared to the simple SVA approach, the manual, tedious approach using Verilog code where we create two parallel condition using fork. The first one checks for the fall of LDP_ and disable the always block. The second waits for two clock cycles and if LDP_ doesn't fall by this time. It prints a fail message and disable the always block. Writing assertions using Verilog is not only hard and tedious but also prone to error and hard to debug.

- **Improves observability:** Assertions are used close to the logic meant to fire at. Which means that if the assertion failed we know exactly where it failed and we don't want to back trace to the source of the problem from the output. For our previous example in Fig The assertion is written to check on FRAME_ and LDP_ so if the assertion failed we know exactly where is the problem.

- There are other benefits for assertions like they can be used for coverage, Reusable, Has severity level, Global turning on and off and multi clock domain crossing.

2.4.2 Immediate Assertions

The first type of assertions is immediate assertions. As the name specifies those type of assertions checks immediately for the condition and does not wait for time as opposed to concurrent assertions. It must also be noted that this type of assertion is used in procedural blocks . The immediate assertion is written as following

```
always @(sensitivity_list)
begin
LABEL : assert (condition) $display (“Success MSG”);
        else $display(“Fail MSG”);
end
```

Fig 2.10 Immediate Assertion

Immediate assertion is written in always block. The “sensitivity_list” is any sensitivity and can be edge triggered of level sensitivity. “LABEL” is optional and can be removed. The condition is a logical condition like || or &&. Finally if assertion does not fail it will display a message and if it fails it will display another message, The display messages are optional and can be removed.

It can be noted that the immediate assertions works as if condition. Also, There must not be anything inside the assertion block related to design as the assertions are not synthesizable. Also it is illegal to use the assertions in continuous assignment.

```
assign a = assert(b||c); //Is Illegal.
```

2.4.3 Concurrent Assertions

Unlike Immediate Assertions that are completely combinational, Concurrent Assertions allows temporal domain sequences which allows sampling on clock. As can be seen in Fig, the assertion starts with property name, this defines a unique name for each assertion written. Next, a condition is written for the assertion and Finally the assertion ends with endproperty.

To use the written assertion we must use assert property and give it the assertion name that will be asserted. If the assertion failed a user written message can be displayed in terminal.

In the following figure, The condition samples on the positive edge of the clock. The assertion first checks for the part of the condition before of the implication sign |-> which is called the antecedent. If that part of the condition is true, the assertion checks for the part after the implication sign which is called the consequent and if this part is true the condition is successful and if the user defined a message to be displayed when asserting the property it will appear in the terminal. In the case of the assertion fails the message is displayed in the terminal. It can be noticed that the consequent can be defined alone in a sequence block which proves useful when the property consists of complex conditions.

```
property PROP;
        @(posedge clk) cStart  |-> req ##2 gnt;
endproperty
assert property(PROP)  $display(“SUCCESS”)
        else $display(“FAIL”);
```

Fig 2.11 Concurrent assertion example

```
Sequence SEQ;
        req ##2 gnt;
endsequence
property PROP;
        @(posedge clk)  |-> SEQ;
endproperty
assert property(PROP)  $display(“SUCCESS”)
        else $display(“FAIL”);
```

Fig 2.12 Sequence example

Since there is not a sampling condition in the consequent or in the defined sequence. They are sampled using the same sampling condition as in the property region.

One of the strong feature of Concurrent Assertion that they are executed in parallel with the design logic (Concurrent) and they are also multi-threaded. For the following figure, at the positive edge of the clock when cStart is high, the consequent begins to evaluate which implies that req is high at the same edge of cStart and gnt is high after two clock cycles from req. When cStart goes high again after one clock cycle like in the case of s2 and s3. Two assertions in different threads and checks for the condition at the same time.

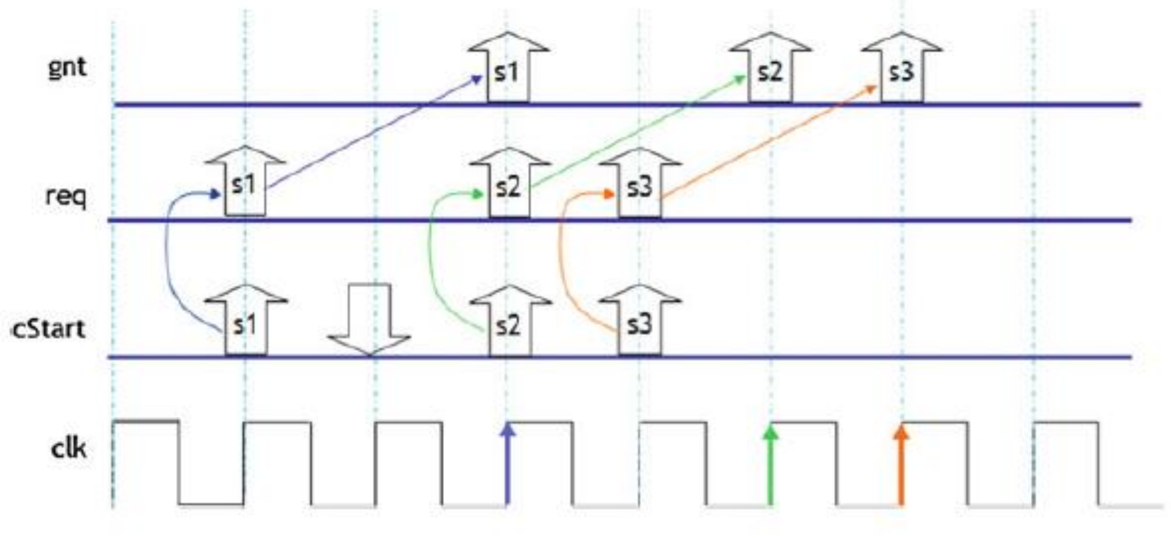


Fig 2.13 Assertion parallelism

As have been discussed in this section. Assertions are practically useful checker modules and provide simple way to write those checker compared to Verilog code. Also, Complex checkers can be divided into smaller blocks called sequences. There are other features for the assertions like parametrizing the assertion so they can be reused, Assertion's messages can be controlled using severity level and the whole assertion block can be disabled in a certain condition using "disable iff" feature.

2.5 Coverage

2.5.1 Introduction

Coverage is a metric we use to measure verification progress [6, p6]and completeness. Coverage metrics tells us what portion of the design has been activated during simulation (that is, the controllability quality of a testbench). Or more importantly, coverage metrics identify portions of the design that were never activated during simulation, which allows us to adjust our input stimulus to improve verification. There are two metrics of coverage, Code coverage and Functional Coverage.

No single metric is sufficient at completely characterizing the verification process. For example, we might achieve 100% code coverage during our simulation regressions. However, this would not mean that 100% of the functionality was verified. The reason for this is that code coverage does not measure the concurrent interaction of behavior within, or between multiple design blocks, nor does it measure the temporal sequences of functional events that occur within a design. Similarly, we might achieve 100% functional coverage, yet only achieve 90% code coverage.

This might indicate that there is either a problem with the fidelity in our functional coverage model (that is, an important behavior of the design was missing from the coverage model), or possibly some functionality was implemented that was never initially specified (for example, perhaps the specification and testplan needs to be updated with some late stage change in the requirements). Hence, to get a complete picture of a project's verification progress we often need multiple metrics.

An important thing is to know the classification of coverage In general, there are multiple ways in which we might classify coverage, but the two most common ways are to classify them by either their method of creation (such as, explicit versus implicit), or by their origin of source (such as, specification versus implementation). For instance, functional coverage is one example of an explicit coverage metric, which has been manually defined and then implemented by the engineer. In contrast, line coverage and expression coverage are two examples of an implicit coverage metric since its definition and implementation is automatically derived and extracted from the RTL representation.

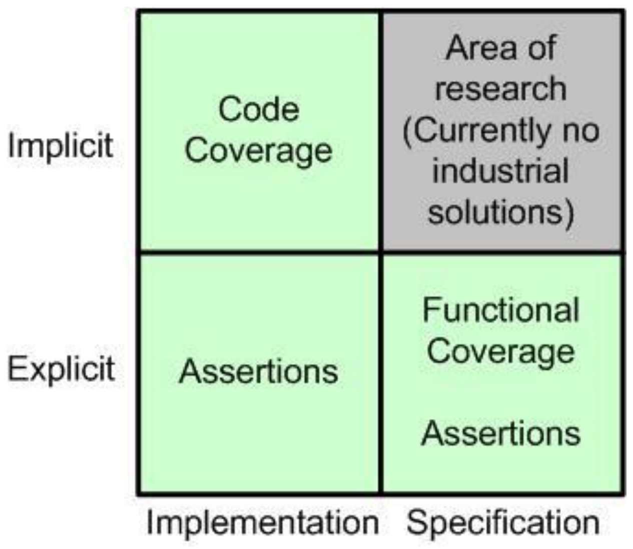


Fig 2.14 Coverage Classification

2.5.2 Code Coverage

Code coverage is a measurement of structures within the source code that have been activated during simulation. One limitation with code coverage metrics are that you might achieve 100% code coverage during your regression run, which means that your testbench provided stimulus that activated all structures within your RTL source code, yet there are still bugs in your design. For example, the input stimulus might have activated a line of code that contained a bug, yet the testbench did not generate the additional required stimulus that propagates the effects of the bug to some point in the testbench where it could be detected.

In fact, researchers have studied this problem and found cases where a testbench achieved 90% code coverage-yet, only 54% of the code was covered would be observable during a simulation run. That means that a bug could exist on a line of code that had been marked as covered—yet the bug was never detected due to insufficient input stimulus to propagate the bug to an observability point.

Another limitation of code coverage is that it does not provide an indication on exactly what functionality defined in the specification was actually tested. For example, you could run into a situation where you achieved 100% code coverage, and then assume you are done. Yet, there could be functionality defined in the specification that was never tested—or even functionality that had never been implemented! Code coverage metrics will not help you find these situations. Even with these limitations, the automatic aspect of code coverage makes it a relatively simple way to identify input stimulus deficiencies in your testbench. And is a great first choice for coverage metrics as you start to evolve your advanced verification process capabilities.

Code coverage has the following metrics that provides a great varieties

- Toggle coverage is a code coverage metric used to measure the number of times each bit of a register or wire has toggled its value. Although this is a relatively basic metric, many projects have a testing requirement that all ports and registers, at a minimum, must have experienced a zero-to-one and one-to-zero transition. In general, reviewing a toggle coverage analysis report can be overwhelming and of little value if not carefully focused. For example, toggle coverage is often used for basic connectivity checks between IP blocks. In addition, it can be useful to know that many control structures, such as a one-hot select bus, have been fully exercised.

- Line Coverage is a code coverage metric we use to identify which lines of our souce code have been executed during simulation. A line coverage metric report will have a count associated with each line of source code indicating the total number of times the line has executed. The line execution count value is not only useful for identifying lines of source code that have never executed, but also useful when the engineer feels that a minimum line execution threshold is required to achieve sufficient testing. Line coverage analysis will often reveal that a rare condition required to activate a line of code has not occurred due to missing input stimulus. Alternatively, line coverage analysis might reveal that the data and control flow of the source code prevented it either due to a bug in the code, or dead code that is not currently needed under certain IP configurations. For unused or dead code, you might choose to exclude or filter this code during the coverage recording and reporting steps, which allows you to focus only on the relavent code.

- Statement coverage is a code coverage metric we use to identify which statements within our source code have been executed during simulation. In general, most engineers find that statement coverage analysis is more useful than line coverage since a statement often spans multiple lines of source code-or multiple statements can occur on a single line of source code. A metrics report used for statement coverage analysis will have a count associated with each line of source code indicating the total number of times the statement has executed. This statement execution count value is not only useful for identifying lines of source code that have never executed, but also useful when the engineer feels that a minimum statement execution threshold is required to achieve sufficient testing.

- Block coverage is a variant on the statement coverage metric which identifies whether a block of code has been executed or not. A block is defined as a set of statements between conditional statements or within a procedural definition, the key point being that if the block is reached, all the lines within the block will be executed. This metric is used to avoid unscrupulous engineers from achieving a higher statement coverage by simply adding more statements to their code.

- Branch coverage is a code coverage metric that reports whether Boolean expressions tested in control structures (such as the if, case, while, repeat, forever, for and loop statements) evaluated to both true and false. The entire Boolean expression is considered one true-or-false predicate regardless of whether it contains logical-and or logical-or operators.

2.5.3 Functional Coverage

The objective of functional verification is to determine if the design requirements, as defined in our specification, are functioning as intended. Code coverage will not be useful for such thing. We will discuss the main aspects of functional coverage.
The functional coverage has two main metrics:

- **Covergroups:** it's a user defined type that sample all the variables included inside it at the same sampling edge
- **Coverpoint:** it's a user defined variable/expression that cover a certain design specification.

As in assertions, we follow some rules in writing functional coverage. In the following figure, We have a simple ALU [8] and we want to apply a coverage model for it. We want to check if we covered the different operations done by the ALU (op signal). First we declare a covergroup (Fig. 2.15) and here we can either define a sampling condition and the covergroup will sample automatically or we don't define a sampling condition and we must use the sample function of the covergroup. Inside the covergroup we declare a coverpoint with the name of the signal we want to cover. Finally, we create an object from the covergroup and inside the initial block we drive the stimulus and call the sample function.

```
bit [1:0] op
covergroup op_cov;
        coverpoint op;
endgroup

op_cov oc;
initial begin
    oc = new();
    //Drive stimulus here
    oc.sample()
```

Fig 2.15 Coverage form

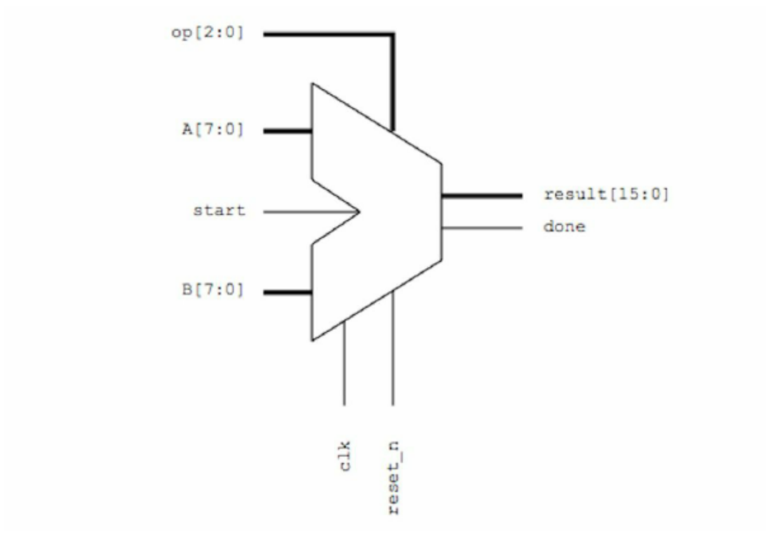


Fig 2.16 Simple ALU Module

One of the strongest features of the functional coverage is bins; Simply bins are something that collect coverage information. Instead of letting the tool to automatically view the values that the signal op_set takes (2’b00,2’b01,2’b10,2’b11) we can manually specify a bin and assign a value to it (Fig. 2.18). In this case, After the coverage is calculated op will has two coverage information, one that cover the value 3’b001 which we give the name “Add” and the other is 3’b010 which we give the name “And”.

```
bit [2:0] op
covergroup op_cov;
        coverpoint op{
                bins Add={‘h001};
                bins And={‘h010};
}
endgroup

op_cov oc;
initial begin
    oc = new();
    //Drive stimulus here
    oc.sample()
```

Fig 2.17 Coverpoint with bins

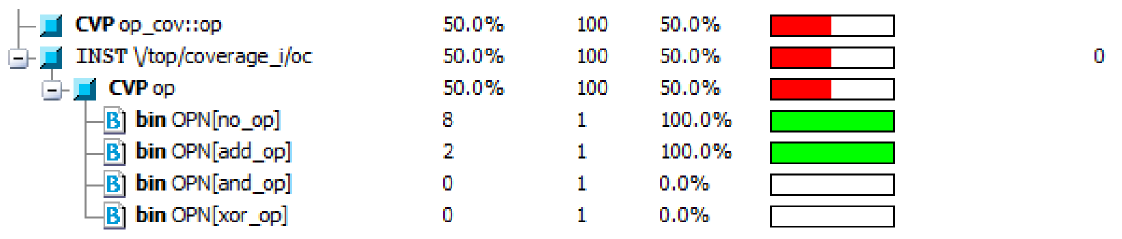
There are other features for coverage like cross coverage which is used to check coverage between two coverpoints at the same time, Also for the bins we can specify a transition that it can cover each cycle or use a bin type like wildcard to specify a value that has a don’t care condition.

3 Proposed Methodology

In this Project, we introduce a novel tool to calculate the coverage from System Verilog Assertions (SVA). In general, SVA provides a mean to use the property written for assertions to be used as a coverage as well as being used as a checker. However, the cover property in Questasim only provides how many times the assertions are exercised and does not give full information about coverage.

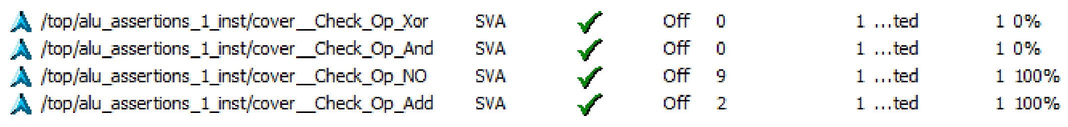
For example, Assertions are written for the purpose of coverage (will be discussed in details in the next chapters) for a simple ALU operation, Here we used the cover property of the assertion and compared it to the coverage output from functional coverage using coverage module.

As can be seen the cover property only offers how many times the assertion exercised but the coverage model provides a full detail about coverage like bins, percentage of coverage. Also, It should be noted that if several coverpoints exist it will be tedious to follow in the case of the cover property of assertion.



```
CVP op_cov::op              50.0%   100   50.0%
INST \/top/coverage_i/oc    50.0%   100   50.0%        0
  CVP op                    50.0%   100   50.0%
    bin OPN[no_op]          8       1     100.0%
    bin OPN[add_op]         2       1     100.0%
    bin OPN[and_op]         0       1     0.0%
    bin OPN[xor_op]         0       1     0.0%
```

Fig 3.1 Simple ALU coverage model



```
/top/alu_assertions_1_inst/cover__Check_Op_Xor   SVA  ✓  Off  0   1 ...ted   1  0%
/top/alu_assertions_1_inst/cover__Check_Op_And   SVA  ✓  Off  0   1 ...ted   1  0%
/top/alu_assertions_1_inst/cover__Check_Op_NO    SVA  ✓  Off  9   1 ...ted   1  100%
/top/alu_assertions_1_inst/cover__Check_Op_Add   SVA  ✓  Off  2   1 ...ted   1  100%
```

Fig 3.2 Simple ALU coverage model using assertions cover property

Our approach is to use the assert property of the assertions and make the assertion display a certain message that can be used to calculate the coverage. Using this approach provides a great flexibility in viewing the coverage to user in a simple way.

The question here, Why wouldn't we use the normal coverage property in systemVerilog as can be seen in (Fig. 3.1) and it will provide as with complete coverage details?

First, Assertions provides us with a strong concept which is sequences that can be used to cover complex design specification. Second, To view coverage we need to run the simulation again but as we will discuss later, our tool extract the coverage information from an output file and does not need the simulation to run again to calculate the coverage so the user can run our tool several time and view the same coverage without going back to Questa.

To summarize, the advantage of the proposed tool are the following:

- Our tool provides a clean , organized coverage data about the running test using assertions
- The tool can calculate coverage without returning to Questa or any simulation tool
- The tool has an advanced GUI section to generate the assertions automatically using user provided data.
- The tool can control the simulation tool like Questa and automatically runs tests several times until a certain coverage percentage is reached then stop the test (Future work)
- As the tool has the coverage data, it can predict the holes in the running test and generate a test case to cover those holes (Future work)

Compared to Questa our tool has the following advantages seen in TABLE 3.1

Point of comparison	**QuestaSim/ModelSim**	**Proposed Tool**
License	Licensed	Open Source
Support other tools by integration	No	Yes
Accumulated coverage	No	Yes
Generation for Coverage signals are automated	No	Yes

TABLE 3.1 Questa and Proposed Tool comparision

As can be seen the assertions provide a great mean to calculate coverage and the tool provides strong features to control over simulation and finally the tool can be extended with more and more features.

Our tool operation can be simplified in the following flow chart. First the tool reads the input files which are the assertion file written by the user and the output messages file from the assertions, Second it fetches the bins and coverpoint names from the assertion file and fetches the coverage data by calculating how many time each bin and coverpoint is exercised. Finally, it calculate the overall coverage, format and display the data.

In the Case of the automatically generated assertion the tool goes to an extra state which is Generate Assertion based coverage file.

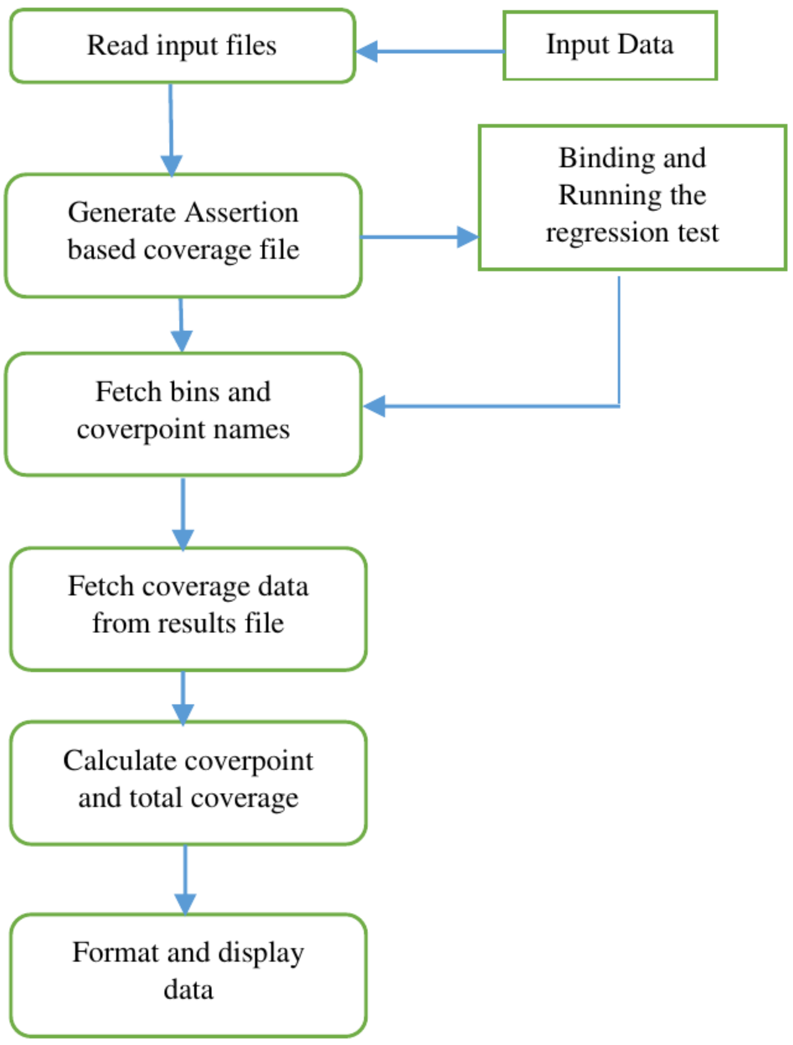

Read input files
Input Data
Generate Assertion based coverage file
Binding and Running the regression test
Fetch bins and coverpoint names
Fetch coverage data from results file
Calculate coverpoint and total coverage
Format and display data

Fig 3.3 Proposed Tool Flowchart

4 Case Study : SD Card

4.1 SD Card Memory

4.1.1 Introduction

SD(Secure Digital) Card is a type of non volatile memories[9] and is designed to provide high capacity, high performance, security in a small size for the requirements of the consumer electronic devices. SD Card are widely used in many application like digital cameras, audio players, mobile phones, etc...

There is another type of SD Card called SDIO or SD I/O Card . SDIO provides interfaces between I/O and SD hosts besides it's memory storage functionality. It must be also noted that SDIO Card are compatible with the mechanical, electrical , power ,signaling and software of the normal SD Card.

SD Card specification is divided into different documents (Fig. 4.1) but here we are interested in the SD Card physical layer that is related to the physical implementation of the device. The physical layer specifies the interface and the protocol used by the SD Card.

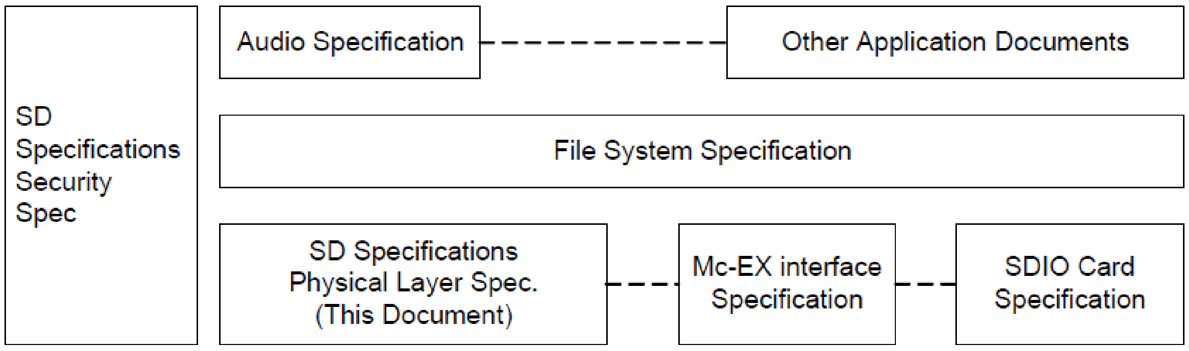

SD Specifications Security Spec
Audio Specification
Other Application Documents
File System Specification
SD Specifications Physical Layer Spec. (This Document)
Mc-EX interface Specification
SDIO Card Specification

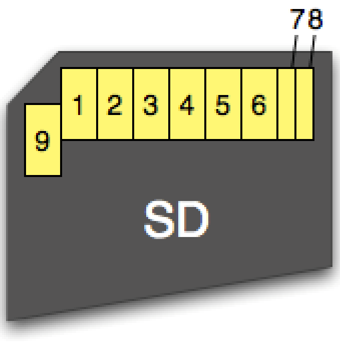
Fig. 4.1 SD Card specification

As there are a wide applications that requires the small size of memory, The SD Card has included many features that supports different application including:

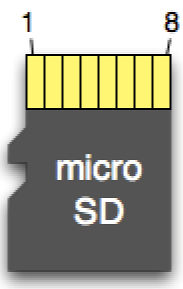
- Support high capacity up to 32GB.
- Support Read/Write operations and erase mechanism.
- Support two modes, default mode that can support up to 12.5MB/sec and high speed mode that can support up to 25MB/sec.
- Password protection for lock and unlock.
- High voltage and Low voltage support.
- Support stationary and portable applications.

4.1.2 Bus Protocol

The following figure shows the interface of the SD Card. The interface has 9 pins, 1x Clock (CLK), 1x Command (CMD), 4x Data (DAT0-DAT3) and 3x Power lines (VDD, VSS1 and VSS2).



Pin	SD	SPI
1	CD/DAT3	CS
2	CMD	DI
3	VSS1	VSS1
4	VDD	VDD
5	CLK	SCLK
6	VSS2	VSS2
7	DAT0	DO
8	DAT1	X
9	DAT2	X



Pin	SD	SPI
1	DAT2	X
2	CD/DAT3	CS
3	CMD	DI
4	VDD	VDD
5	CLK	SCLK
6	VSS	VSS
7	DAT0	DO
8	DAT1	X

Fig. 4.2 Pin interface

The SD Card transfer protocol is based on command and data serial bit stream where each stream has a start and an end bit, First The host send a command on the CMD line (Fig. 4.3) to a single SD Card or all SD Card connected to it, After the SD Card receives the command it can send a response on the CMD line to the host. A single SD Card can send the response to the host or all the connected cards.

Some commands are accompanied by data transfer like (Read and Write), The transfer is done in blocks (Fig. 4.4) and the protocol support single block and multiple block operations. For the multiple block operations the host can send a stop command to stop the transfer. Moreover, The host can select how many data lines are used for the transfer and in case of the write operation the SD Card sends a busy signal on DAT0 (Fig. 4.5) until the operation has finished.

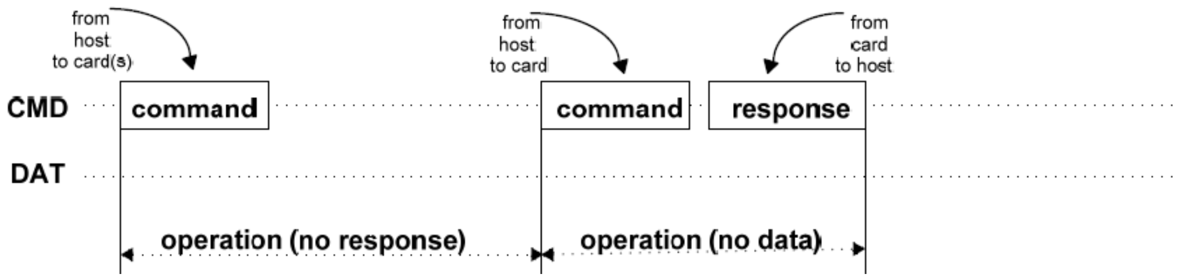


Fig. 4.3 Command Response operation

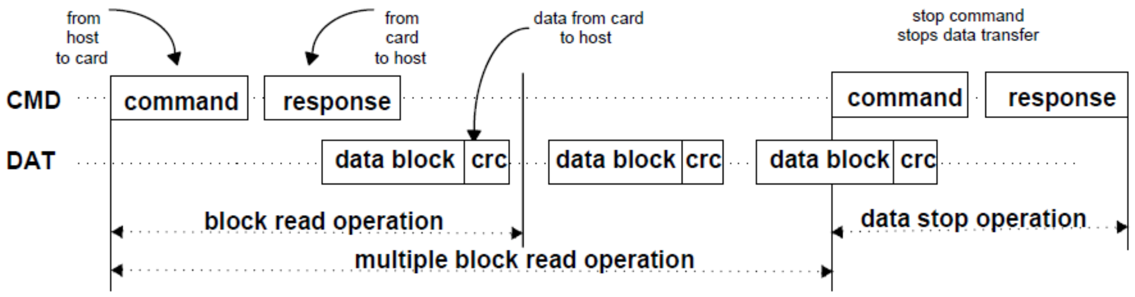


Fig. 4.4 Data transfer operation

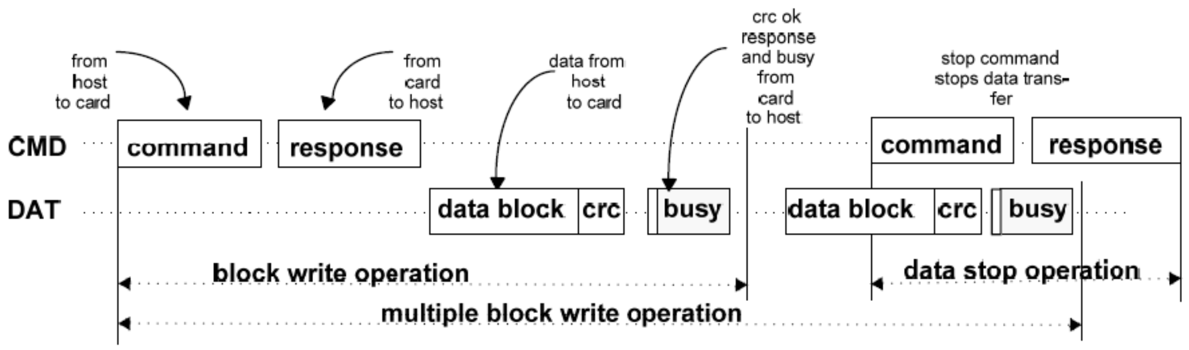


Fig. 4.5 Write operation

4.1.2.1 Command response operation

As previously described, The host sends a command to the SD Card on the CMD line, The command has a fixed format of 48-bits (Fig. 4.6) where the CMD line goes from high to low to indicate the start of command transfer (start bit) then the CMD line goes high again to indicate that the host is transferring the command and in case of the response by SD Card this bit becomes low. After that, The content of the command is sent followed by the CRC check and the CMD line goes high to indicate that the command has finished (stop bit).

In case of the response (Fig. 4.7) sent by SD Card, The same operation as sending the command is done but the format is larger as more data is sent to show the state of the SD Card.

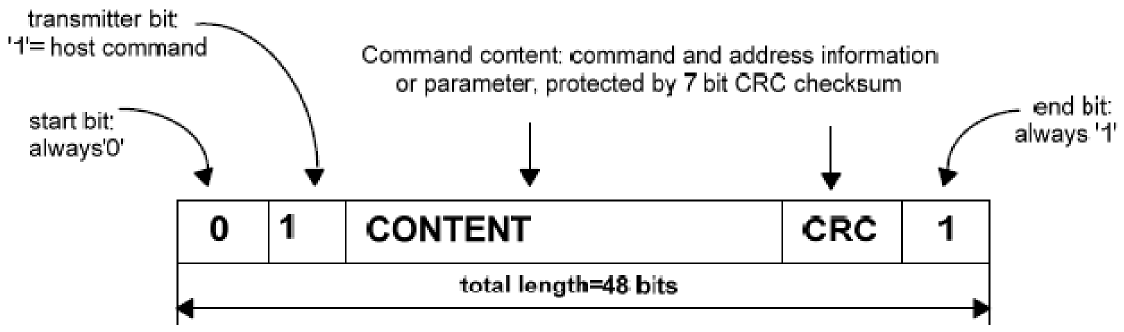


Fig. 4.6 Command format

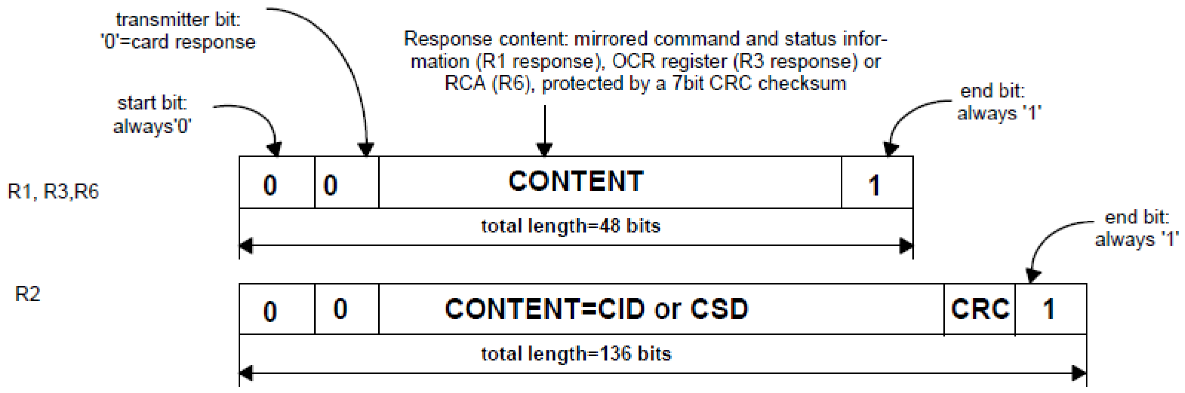


Fig. 4.7 Response format

4.1.2.2 Data operation

SD Card supports two data format. The first one is the usual 8-bit width data (Fig. 4.8) where the MSB bit is sent first and the LSB bit is sent last, SD-Card can use only DAT0 to send the data(Fig. 4.9) or all the four bit lines(Fig. 4.10).

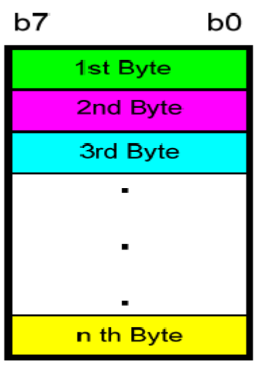


Fig. 4.8 8-bit width data

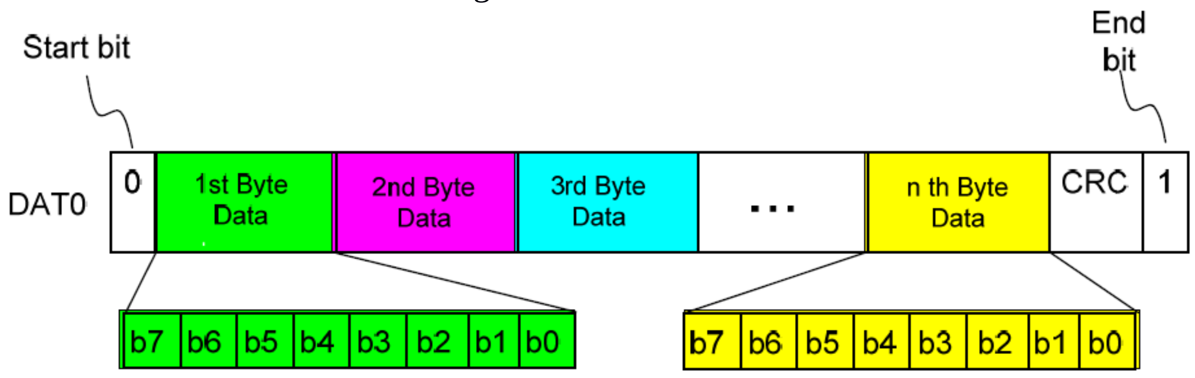


Fig. 4.9 only DAT0 is used

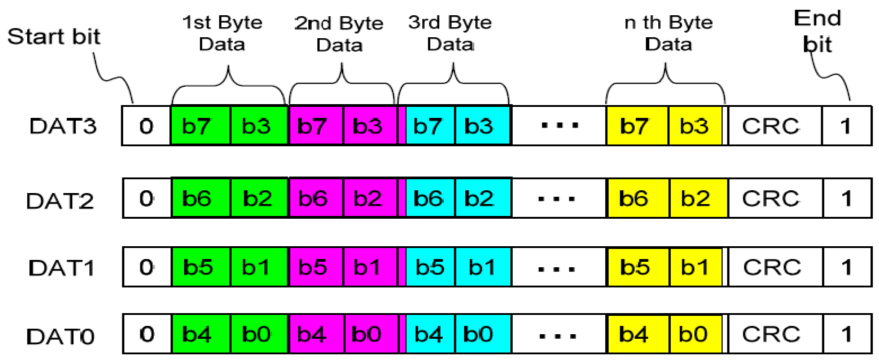

Start bit
1st Byte Data
2nd Byte Data
3rd Byte Data
n th Byte Data
End bit
DAT3 0 b7 b3 b7 b3 b7 b3 ... b7 b3 CRC 1
DAT2 0 b6 b2 b6 b2 b6 b2 ... b6 b2 CRC 1
DAT1 0 b5 b1 b5 b1 b5 b1 ... b5 b1 CRC 1
DAT0 0 b4 b0 b4 b0 b4 b0 ... b4 b0 CRC 1

Fig. 4.10 All the four Data lines are used

The second one is the wide data format where a single 512-bits burst is used to send the data, This format is useful when the SD-Card sends its status to the host. Like the usual 8-bit, The SD-Card sends the MSB first and the LSB last and the SD-Card can use only DAT0 signal or the whole four lines to send the data.

4.1.3 SD Card States

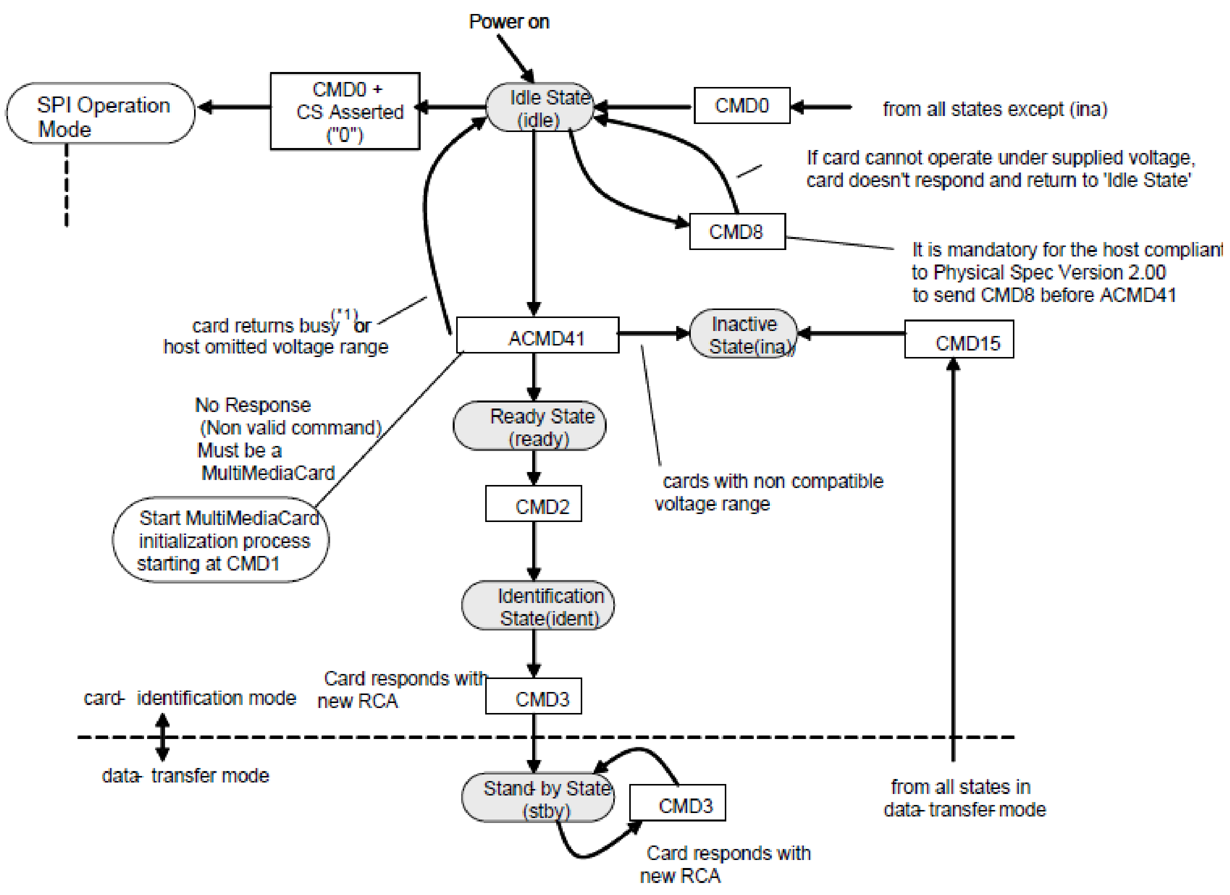
For the SD-Card to function correctly, It passes through several states including identification, transfer and disconnection from the host. All the SD-Card states are divided into three modes, Inactive mode where the SD-Card does not operate, Identification mode where the SD-Card starts to connect with the host and setup the communication and finally the data transfer mode where the SD-Card transfer the data.

4.1.3.1 Identification mode

In this mode (Fig. 4.11), the host resets all the SD-Card connected to it and checks the suitable operating voltage range and asks them to establish their addresses to communicate with them (Relative Card Address). All the operations done in this mode uses the command line (CMD) only. First, The host issues CMD0 to reset the SD Card and move it to idle state. Second, It checks the operation voltage as the SD Card does not the host supporting voltage range and the same with the host so the host uses CMD8 and ACMD41. Finally, The host issues CMD3 which asks the SD Card to respond with it's address, If the address is already assigned to one of the SD Card connected to the host, The host issues CMD3 again and the SD Card respond with a new address.

4.1.3.2 Data transfer mode

In this mode (Fig. 4.12), SD Card makes all it's data transfer with the host like reading, writing, etc.. Before issuing any data transfer commands the host needs to know the specification of the SD Card like block length, storage capacity and so on so it issues CMD9 and the SD Card respond with it's specification.
The host selects one SD Card for transfer mode by issuing CMD7 with the address of the SD Card. The SD Card that has the assigned address in the command is set for data transfer and the other cards are sent to standby mode which means it's the responsibility of the host to select the card needed for transfer.



Power on
SPI Operation Mode
CMD0 + CS Asserted ("0")
Idle State (idle)
CMD0
from all states except (ina)
If card cannot operate under supplied voltage, card doesn't respond and return to 'Idle State'
CMD8
It is mandatory for the host compliant to Physical Spec Version 2.00 to send CMD8 before ACMD41
card returns busy (*1) or host omitted voltage range
ACMD41
Inactive State(ina)
CMD15
No Response (Non valid command) Must be a MultiMediaCard
Ready State (ready)
cards with non compatible voltage range
Start MultiMediaCard initialization process starting at CMD1
CMD2
Identification State(ident)
card- identification mode
Card responds with new RCA
CMD3
data- transfer mode
Stand- by State (stby)
CMD3
from all states in data- transfer mode
Card responds with new RCA

Fig. 4.11 SD Card identification mode

All the commands and states in (Fig. 4.12) can be summarized as the following:

- All data read commands can be aborted any time by the stop command (CMD12). The data transfer will terminate and the card will return to the Transfer State. The read commands are: block read (CMD17), multiple block read (CMD18), send write protect (CMD30), send scr (ACMD51) and general command in read mode (CMD56).

- All data write commands can be aborted any time by the stop command (CMD12). The write commands shall be stopped prior to deselecting the card by CMD7. The write commands are: block write (CMD24 and CMD25), program CSD (CMD27), lock/unlock command (CMD42) and general command in write mode (CMD56).

- As soon as the data transfer is completed, the card will exit the data write state and move either to the Programming State (transfer is successful) or Transfer State (transfer failed).
- If a block write operation is stopped and the block length and CRC of the last block are valid, the data will be programmed.

- The card may provide buffering for block write. This means that the next block can be sent to the card while the previous is being programmed. If all write buffers are full, and as long as the card is in Programming State (see SD Memory Cardstate diagram Figure 4-3), the DAT0 line will be kept low (BUSY).

- There is no buffering option for write CSD, write protection and erase. This means that while the card is busy servicing any one of these commands, no other data transfer commands will be accepted. DAT0 line will be kept low as long as the card is busy and in the Programming State. Actually if the CMD and DAT0 lines of the cards are kept separated and the host keep the busy DAT0 line disconnected from the other DAT0 lines (of the other cards) the host may access the other cards while the card is in busy.
- Moving another card from Stand-by to Transfer State (using CMD7) will not terminate erase and programming operations. The card will switch to the Disconnect State and will release the DAT line.

- A card can be reselected while in the Disconnect State, using CMD7. In this case the card will move to the Programming State and reactivate the busy indication.
- Resetting a card (using CMD0 or CMD15) will terminate any pending or active programming operation. This may destroy the data contents on the card. It is the host’s responsibility to prevent this.

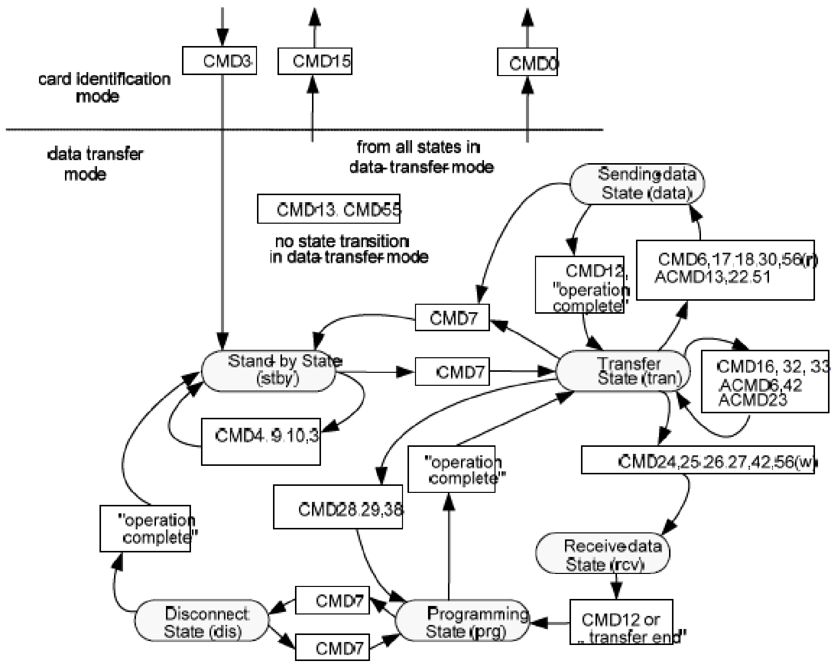


Fig. 4.12 SD Card data transfer mode

4.1.4 SD Highlighted Commands

4.1.4.1 Read

As any command sent to the SD Card on the CMD line has start bit and end bit, The data on the DAT lines has start bit by lowering the any of the DAT lines and an end bit by pulling up the DAT lines. If only on DAT line is used (DAT0) one start bit and one end bit are used, If all the four DAT lines are used (DAT0,DAT1,DAT2,DAT3) one start bit and one end bit are used for each line summing up to four start bits and four end bits.

In this thesis, It was mentioned that the data transfer in SD Card are block oriented, The block is the basic unit of transfer and has a maximum size of 512 bytes and the block length can be set using CMD16. At the end of each block a CRC check is appended for error detection. For a single block read CMD17 is used and for multiple block read CMD18 is used and the transmission can be stopped at any time using CMD12 in the case of the multiple block read. Also, The SD Card detect misalignment if a wrong address for reading is used and abort the transmission immediately.

The read operation of the SD Card can be interrupted by turning the power off safely as the SD Card ensure that the data will be safe and not damaged except for the write and the erase commands.

4.1.4.2 Write

Like the Read operations, The write operation is block oriented and the block length can be set using CMD16. However, In the write operation if the power is interrupted the data is not ensured to stay safe and not damaged.

4.1.4.3 Erase

Erase is used to remove the data from SD Card, To ensure maximum data throughput during the erase operation many block erase is advised to be used. For any erase operation the start block is specified by using CMD32 and the end block is specified by using CMD33. Finally, The erase operation starts using CMD38 and the SD Card will erase all the blocks between the start and end block inclusive.

It should be noted that if any protected sectors are included in the erase operation they are left untouched and only the non protected are removed.

4.2 SD Card Controller

4.2.1 Introduction

The main idea behind any memory controller is to unload low-level memory management from the host processor, freeing up resources. The main aim of the memory controller is to provide the most suitable interface and protocol between the host and the memories to efficiently handle data, maximizing transfer speed, data integrity and information retention. Designing memory controllers is challenging in terms of performance, area, power consumption and reliability. The memory controller can be integrated with the memory die on the same chip or can be on a separate chip.

In any memory controller, there are two sides one for card which handles sending and receiving data and manages the low level protocol and the other for the host. In our study the selected SD Card memory controller (Fig. 4.13) is connected to a wishbone interface at the host side[10].

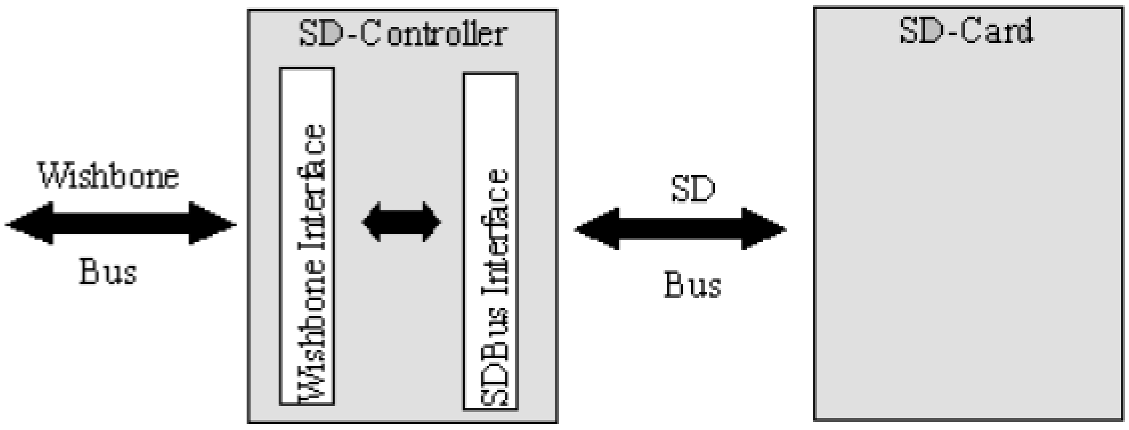


Fig. 4.13 SD Card Memory Controller

The selected controller has the following features:

- 32-bit Wishbone Interface
- DMA
- Buffer Descriptor
- Compliant with SD Host Controller Spec version 2.0
- Support SD 4-bit mode
- Interrupt-on-completion of Data and Command transmission
- Write/Read FIFO with variable size
- Internal implementation of CRC16 for data lines and CRC7 for command line

4.2.2 Architecture

The SD Card memory controller architecture (Fig. 4.14) consist of three main sections, The host interface which is responsible for communicating with the host using the wishbone interface, Physical interface which is responsible for sending the data, commands and receiving response on the SD Card pins using the protocol in the specification and the Card interface that prepares the data and commands, format the commands and add CRC checks. The main section consists of several modules, Summing up for seven modules.

4.2.2.1 Host Interface

- **SD Controller Top** the main module of host interface, it contains the master and slave wishbone interface to communicate with the host (rest of the system) through several defined register which are used by the controller to setup the SD Card.

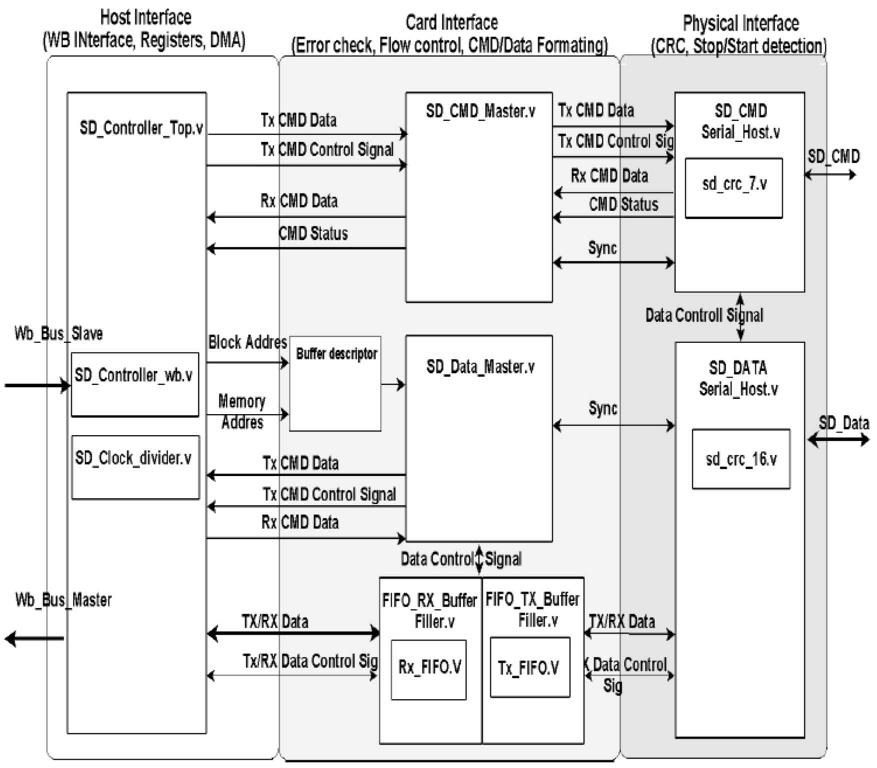


Fig. 4.14 SD Card Memory Controller architecture

4.2.2.2 Physical Interface

- **SD CMD Serial Host** The block is responsible for writing CMD to SD Card and reading responses, It can write command and does not receive the response from SD Card or it can write command and receive the corresponding response. It consists of FSM that manages the operation (Fig. 4.15).

- **SD Data Serial Host** Like the SD CMD Serial Host, This block is responsible to send and receive data from SD Card, It obtains the data from FIFOs in Card interface and store the data received from SD Card into the FIFOs again. The FSM (Fig. 4.16) describes it's operations.

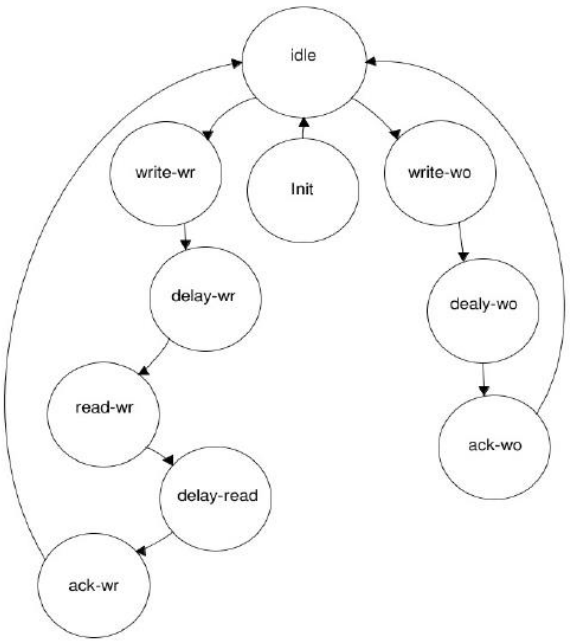


Fig. 4.15 SD CMD Serial Host FSM

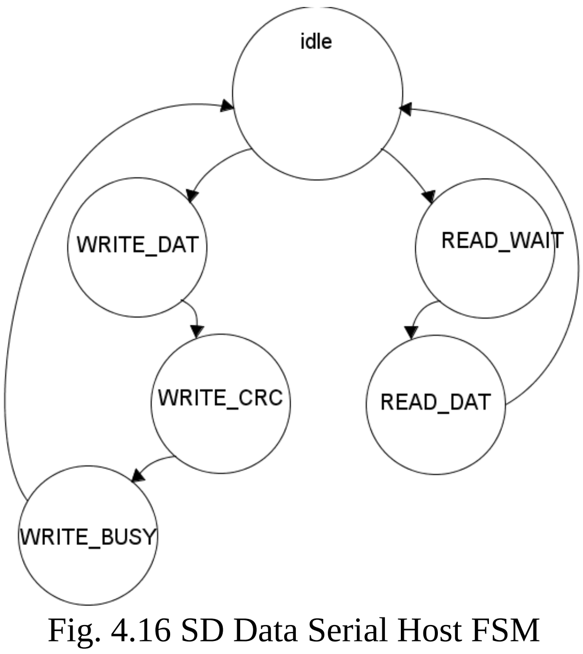


Fig. 4.16 SD Data Serial Host FSM

4.2.2.3 Card Interface

- **SD CMD Master** it's responsible to read the Response from SD CMD Serial Host and save it to the user accessible registers in host interface and reads the CMD from user accessible register in host interface and send it to SD CMD Serial Host. FSM (Fig. 4.17) describe it's operation

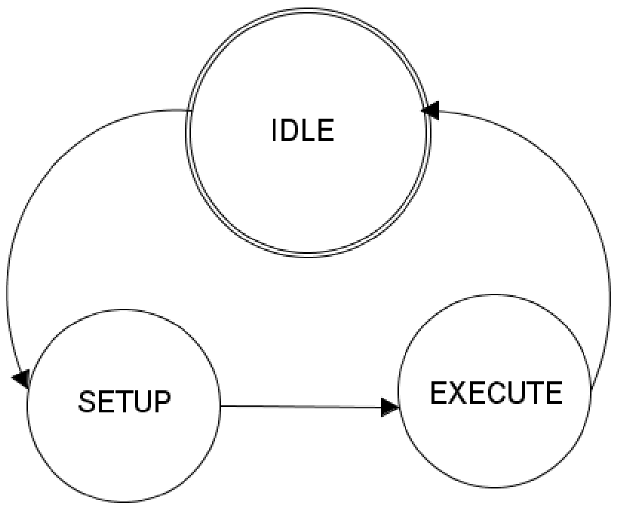


Fig. 4.17 SD CMD Master FSM

- **SD Data Master** it keeps track of the transmission operation and buffers overflow/underflow . If anything goes wrong during data transmission it issues reset CMD and restart the operation again. FSM(Fig 4.18) describes it's operation

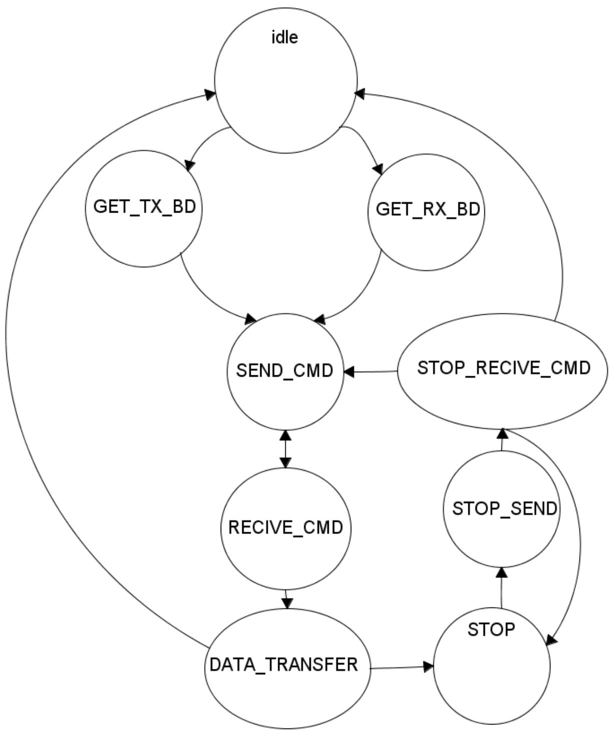


Fig. 4.18 SD Data Master FSM

5 Tool Implementation

In this chapter, we will discuss the proposed tool that would automate the whole assertions-based coverage process and make it at ease for even designers to cover their regression testing. The next sub-sections are first describing the standardization for the assertion code, to be able to do the coverage task. Then the details of the tool implementation are illustrated, as well as the GUI and the different features of the proposed tool.

Two design modules were used to test the tool implementation and checking its functionality. The first one is a simple ALU, where a coverage block and an assertion block were written to match the coverage plan of a basic ALU. The results from the two blocks were cross-checked. A more complex protocol -SD-Card controller- was used to test the tool. Again, a coverage and an assertion results were crosschecked with the coverage plan. The detailed results of running these two modules on our tool are showed in next chapter.

5.1 Standardization of the assertion Coding

As we discussed in Chapter 2, The ordinary style of assertions codes is as follows:

```
property PROPERTY_NAME;
      @(posedge clk ) (TRIGGER)  |->  (TEST_CASE);
endproperty

ASSERTION_NAME: assert property (PROPERTY_NAME)
            $display (“PROPERTY_NAME passed");
else
            $display (“PROPERTY_NAME failed");
```

This is the standard shape of SystemVerilog Assertions. In this code, a system message containing “PROPERTY_NAME passed” will show up in the transcript of the Tool (Questa for example), if the TEST_CASE is true, otherwise a message of “PROPERTY_NAME failed” will show up.

The most important note here is that in both cases if the assertion passed or failed, there will be a message with its state. This behavior is very desirable when dealing with assertions as a debugging tool, but for our case to use assertions as a coverage tool we will need some modifications.

In order to make industrial tool, parameters must be stated so that anyone can modify, change, and edit the running scripts. The standardization purpose is to unify the naming of SystemVerilog Assertions properties to be the same as the bins names in coverage and the output messages. On the other hand, the output must be standardized also, so that any user can modify manually each signal or uses the assertions file generated in other environments.

After observing the assertions codes and output styles, we noticed that our goal is achieved by applying the following modifications and standardization in writing SystemVerilog Assertions.

5.1.1 Standardization of the assertions property style

First, we know that the basic SystemVerilog Assertions property lines are generally:

```
property PROPERTY_NAME;
      @(posedge clk ) (TRIGGER)  |->  (TEST_CASE);
endproperty
```

Where TRIGGER is the event on which it enters the assertion, and TEST_CASE is the case on which the assertion passes or fails.

We need to modify the shape of assertions codes to meet our requirements for coverage, so we propose that the SVA code must be as the next lines of code:

```
property PROPERTY_NAME;
   @(posedge clk && TRIGGER)  (TEST_CASE)  |-> Always_True_Variable == 0;
endproperty
```

To use the assertions for coverage purpose, we need to test the TRIGGER event as the ordinary case, but as the assertions standard logs show both the pass and fail messages, this will confuse the Python script which calculate the coverage percentage (as discussed in the previous chapter). Therefore, we don't want the “fail” message to appear every time the assertion property does not fire as the ordinary assertion code do. Here we define our magical trick, the Always_True_Variable.

The Always_True_Variable is a dummy variable important for not displaying fail message. We can think of the new code form as: if the TEST_CASE did not happen, the program will not enter this assertion from the beginning so the fail messages will not appear, however it will enter it if and only if the TRIGGER is true and the TEST_CASE is true, in this case, the assertion will definitely pass. After this pass message is written to the external text file, the python script can start counting them to calculate the coverage.

5.1.2 Standardization of the output of assertions messages

As discussed earlier, the ordinary SystemVerilog Code is written as follows,

```
property PROPERTY_NAME;
      @(posedge clk ) (TRIGGER)  |->  (TEST_CASE);
endproperty

ASSERTION_NAME: assert property (PROPERTY_NAME)
            $display (“PROPERTY_NAME passed");
else
            $display (“PROPERTY_NAME failed");
```

Here, the $display messages will output both the pass and fail states of the assertion. In addition, this message will appear every time the TRIGGER condition is true. This problem is solved with the first modification in the previous section, but for the tool to find the coverage percentage, and to automatically know the Coverage pins, a way far modification is needed. This is the Output standardization.

First, to allow the tool to find the assertions output messages, we need to move them from the Questa Sim transcript to an external text file, so that the Core Python Code can find it. This output will be displayed in external text file using $fdisplay( ) function in SystemVerilog. Assertions output must be exactly like the following form:

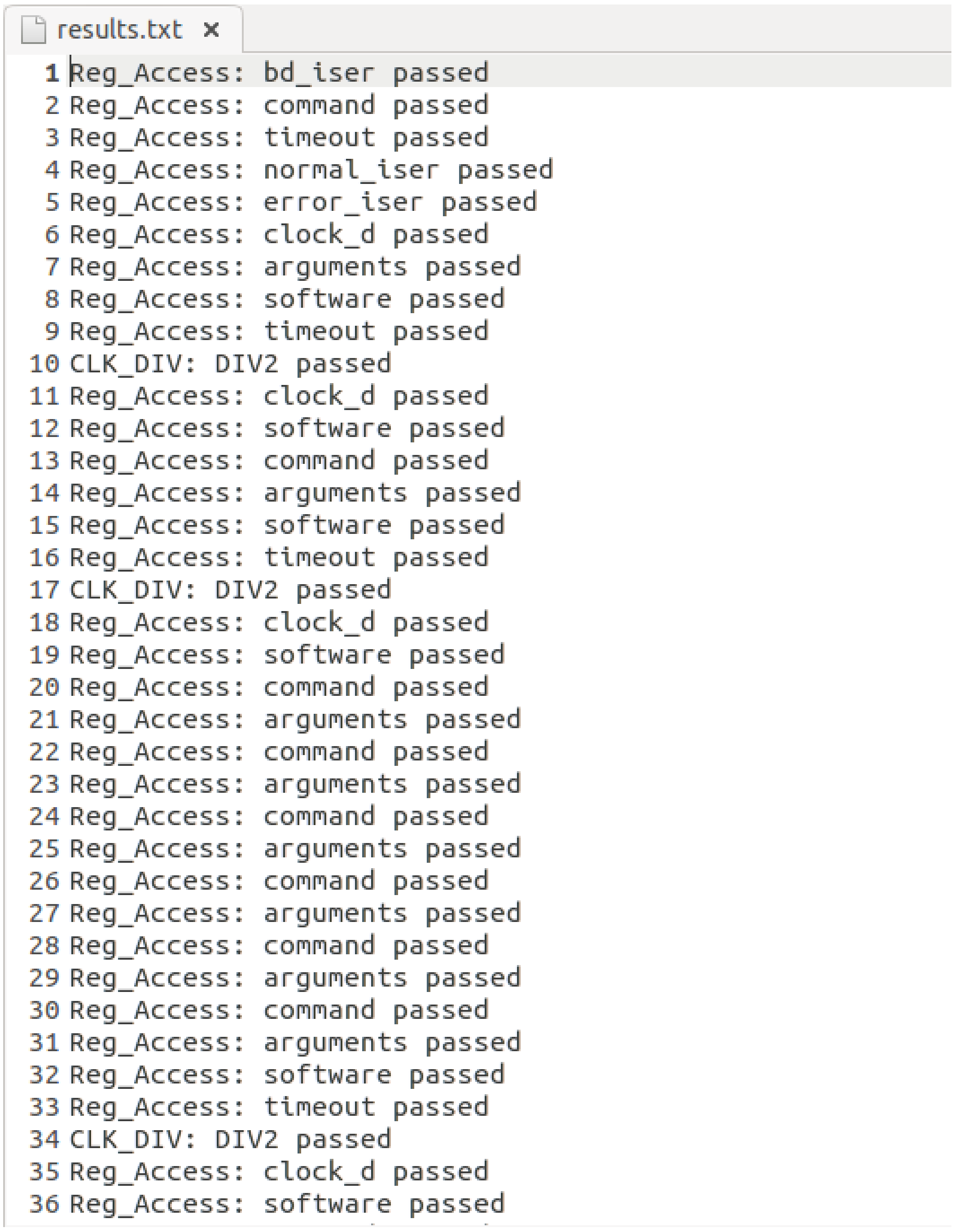
```
COVER_POINT_NAME : PROPERTY_NAME Passed
```

Where PROPERTY_NAME is the same name as the property of this assertion in the assertions code file, so that the tool can fully automate the process of naming the output signals and their corresponding coverage percentage in the GUI output. In reality, the tool will automatically search for all the properties names from the SystemVerilog Assertions code file, and then counts their appearance in the external file (the output file from running SystemVerilog Assertions).

The example in Figure 5.1 shows the output format of the assertions code, and Figure 5.2 shows a real example of the output text file from running the SVA code on the SD-Card module.

```
ASSERTION_NAME: assert property (PROPERTY_NAME)
          $fdisplay (file, " COVER_POINT_NAME: PROPERTY_NAME Passed");
```

Fig. 5.1 example of the assertions code, showing the standardization of the output



results.txt

```
Reg_Access: bd_iser passed
Reg_Access: command passed
Reg_Access: timeout passed
Reg_Access: normal_iser passed
Reg_Access: error_iser passed
Reg_Access: clock_d passed
Reg_Access: arguments passed
Reg_Access: software passed
Reg_Access: timeout passed
CLK_DIV: DIV2 passed
Reg_Access: clock_d passed
Reg_Access: software passed
Reg_Access: command passed
Reg_Access: arguments passed
Reg_Access: software passed
Reg_Access: timeout passed
CLK_DIV: DIV2 passed
Reg_Access: clock_d passed
Reg_Access: software passed
Reg_Access: command passed
Reg_Access: arguments passed
Reg_Access: command passed
Reg_Access: arguments passed
Reg_Access: command passed
Reg_Access: arguments passed
Reg_Access: command passed
Reg_Access: arguments passed
Reg_Access: command passed
Reg_Access: arguments passed
Reg_Access: command passed
Reg_Access: arguments passed
Reg_Access: software passed
Reg_Access: timeout passed
CLK_DIV: DIV2 passed
Reg_Access: clock_d passed
Reg_Access: software passed
```

Figure 5.2 a sample of the external text file generated by SystemVerilog Assertions for the SD-Card Controller.

In Figure 5.3, a real example for the assertions code from the SD-Card module is shown. Here the code is covering the wishbone interface of the SD-Card discussed in chapter 4, then displays the pass messages in the external text file defined in the top module of the testbench (sd_controller_top_tb). These messages are displayed in the external file as clearly shown in Fig.5.2.

```
//////////////////// Properties of covergroup WB_Interf /////////////////////
// coverpoint: Reg_Access


property arguments;
@(negedge clk && wb_we_i) (wb_adr_i == 8'h00) |-> z == 0;
endproperty

property command;
@(negedge clk && wb_we_i) (wb_adr_i == 8'h04) |-> z == 0;
endproperty

property block;
@(negedge clk && wb_we_i) (wb_adr_i == 8'h20) |-> z == 0;
endproperty

 property resp1;
 @(negedge clk && wb_we_i) (wb_adr_i == 8'h0c) |-> z == 0;
 endproperty

property software;
@(negedge clk && wb_we_i) (wb_adr_i == 8'h28) |-> z == 0;
endproperty

property timeout;
@(negedge clk && wb_we_i) (wb_adr_i == 8'h2c) |-> z == 0;
endproperty

property controller;
@(negedge clk && wb_we_i) (wb_adr_i == 8'h1c) |-> z == 0;
endproperty

property normal_iser;
@(negedge clk && wb_we_i) (wb_adr_i == 8'h38) |-> z == 0;
endproperty

property error_iser;
@(negedge clk && wb_we_i) (wb_adr_i == 8'h3c) |-> z == 0;
endproperty

property capa;
@(negedge clk && wb_we_i) (wb_adr_i == 8'h48) |-> z == 0;
endproperty

property bd_status;
@(negedge clk && wb_we_i) (wb_adr_i == 8'h50) |-> z == 0;
endproperty

property bd_isr;
@(negedge clk && wb_we_i) (wb_adr_i == 8'h54) |-> z == 0;
endproperty

property bd_iser;
@(negedge clk && wb_we_i) (wb_adr_i == 8'h58) |-> z == 0;
endproperty

property bd_rx;
@(negedge clk && wb_we_i) (wb_adr_i == 8'h60) |-> z == 0;
Endproperty
```

```
///////////////////// Assertions of covergroup WB_Interf //////////////////////
// coverpoint: Reg_Access


arguments_as: assert property (arguments)
  $fdisplay(sd_controller_top_tb.file, "Reg_Access: arguments passed") ;

command_as: assert property (command)
  fdisplay(sd_controller_top_tb.file,"Reg_Access: command passed") ;


block_as: assert property (block)
  $fdisplay(sd_controller_top_tb.file,"Reg_Access: block passed") ;

resp1_as : assert property (resp1 )
  $fdisplay(sd_controller_top_tb.file,"Reg_Access: resp1 passed") ;

software_as : assert property (software)
  $fdisplay(sd_controller_top_tb.file,"Reg_Access: software passed") ;

timeout_as : assert property (timeout)
 $fdisplay(sd_controller_top_tb.file,"Reg_Access: timeout passed") ;

controller_as : assert property (controller)
 $fdisplay(sd_controller_top_tb.file,"Reg_Access: controller passed") ;

normal_iser_as : assert property (normal_iser)
  $fdisplay(sd_controller_top_tb.file,"Reg_Access: normal_iser passed") ;

error_iser_as : assert property (error_iser)
$fdisplay(sd_controller_top_tb.file,"Reg_Access: error_iser passed") ;

capa_as : assert property (capa)
$fdisplay(sd_controller_top_tb.file,"Reg_Access: capa passed") ;

bd_status_as : assert property (bd_status)
 $fdisplay(sd_controller_top_tb.file,"Reg_Access: bd_status passed") ;

bd_isr_as : assert property (bd_isr)
 $fdisplay(sd_controller_top_tb.file,"Reg_Access: bd_isr passed") ;

bd_iser_as : assert property (bd_iser)
  $fdisplay(sd_controller_top_tb.file,"Reg_Access: bd_iser passed") ;

bd_rx_as : assert property (bd_rx)
  $fdisplay(sd_controller_top_tb.file,"Reg_Access: bd_rx passed") ;

bd_tx_as : assert property (bd_tx)
 $fdisplay(sd_controller_top_tb.file,"Reg_Access: bd_tx passed") ;
```

Figure 5.3 A real code snippet of signals from the SystemVerilog assertions code of the wishpone interface in the SD-Card module

5.2 Core Implementation of the Tool:

The tool is implemented using Python due to its ease of use when manipulating strings and output files, and when dealing with GUI. It is an open source language; its libraries are available and can be run on all environments with no problem.
The first step requires that the user writes the SystemVerilog Assertions code, compile and run it on any tool (for example: Questa or VCS), then write the paths of both the assertions file (.sv) and the external text file (.txt) in the entry boxes in the GUI of our tool.

Figure 5.4 shows the flowchart of the tool to find the coverage percentage from an assertion file. The tool first takes the paths of both SystemVerilog Assertions file and the result file from the GUI input. The core code then starts by opening the assertions file from the verification directory, then copying the code after removing the comments into an intermediate file, this step makes it easier for the tool to determine the bins and the coverpoints that the user need to be covered. The tool then extract the bins and coverpoints names from the intermediate file and delete it.

After that, the results file from the assertions code is opened and is used to find the coverage by parsing all the strings inside it, count the number of occurrences of every pin, and map them to their Coverpoints. Finally the tool calculate the coverage percentage for each coverpoint and the total coverage for all coverpoints, then formats and display the output in the terminal, in addition to writhing them on a text file to allow the GUI Code to read them and display this data in the colorful scheme.

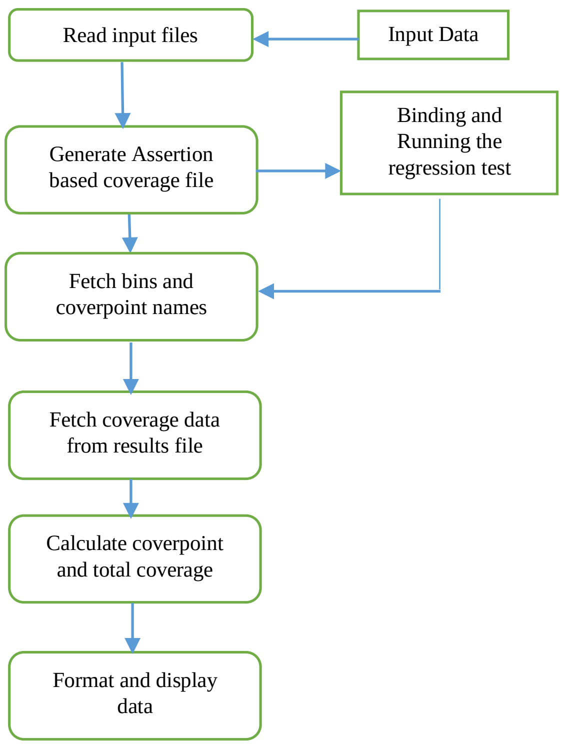

Read input files
Input Data
Generate Assertion based coverage file
Binding and Running the regression test
Fetch bins and coverpoint names
Fetch coverage data from results file
Calculate coverpoint and total coverage
Format and display data

Figure 5.4 The flowchart of the tool algorithm

Before discussing the GUI of the tool, we need to look at the core implementation of it. The core was implemented at first using Perl language but it was converted to python to be compatible and easy to use with the GUI part of the tool.
The core follows the flowchart of the tool, and its operation will be discussed in the following steps:

- The code takes the path of two files, Assertions and the output file by the assertion when the test is run, Here is called Result File.

```
AssertionPath=sys.argv[1]
ResultPath=sys.argv[2]
```

- The code then open those files for reading and another two files for writing, The first one is called Intermediate file and the purpose is to make another version of the assertion file but without comment so it will be easy for the rest of the code to calculate the coverage data efficiently. The second one is called output file which is used to write the useful data for the GUI part to display it.

```
AssertionFile=open(AssertionPath,"r")
IntermediateFile=open("Intermediate_Tool.sv","w+")
ResultFile=open(ResultPath,"r")
OutputFile=open("Output_File.txt","w+")
```

- The code then remove the comments from the assertion file and save the new data in the Intermediate file, The assertion file is closed and the Intermediate file is opened for reading

- The code start fetching the bins name, which are the property name of the assertion from the Intermediate file and create a hash table that is accessed by the name of the bin and each hash table entry has several entries, like the coverpoint name and how much time the bin is exercised.

```
for line in IntermediateFile:
        if(find_str(line,"assert property")!=-1):
                token=line[find_str(line,"(")+1:find_str(line,")")]
                Coverage_Bins[token.split(" ")[0]]=["None",0,1,1]
```

- The code makes another hash table for coverpoint and save the coverpoint names in it after fetching them from the Intermedate File. It then updates the bins hashtable and add the coverpoint name to it

```
for line in IntermediateFile:
        if(find_str(line,"fdisplay")!=-1 or find_str(line,"info")!=-1):
                for Key in Coverage_Bins:
                        token=line.split("")[1]
                        if (find_str(token,Key)!=-1):
                                Coverage_Bins[Key]=[token.split(":")[0],0,1,1]
                                CoverPoint_Coverage[token.split(":")[0]]=[0,0]
```

- In this part, The code calculates two thing, The first is how many times each bins is exercised and the second is to calculate the coverage of each coverpoint. The coverage of each coverpoint of course depends on the coverage of each bin.

```
for line in ResultFile:
        for Key in Coverage_Bins:
                token=re.split(':| ',line)
                if (token[1]==Key):


        Coverage_Bins[Key]=[Coverage_Bins[Key][0],Coverage_Bins[Key][1]+1,1,1]
                        if Coverage_Bins[Key][1]==1:
                                CoverPoint=Coverage_Bins[Key][0]

        CoverPoint_Coverage[CoverPoint]=[CoverPoint_Coverage[CoverPoint][0],Cover
Point_Coverage[CoverPoint][1]+1/CoverPoint_Coverage[CoverPoint][0]]
```

- The code then calculate the total coverage. It then displays the data on the terminal.
- Finally it writes the useful data to the output file to be used by the GUI.

5.3 GUI Implementation:

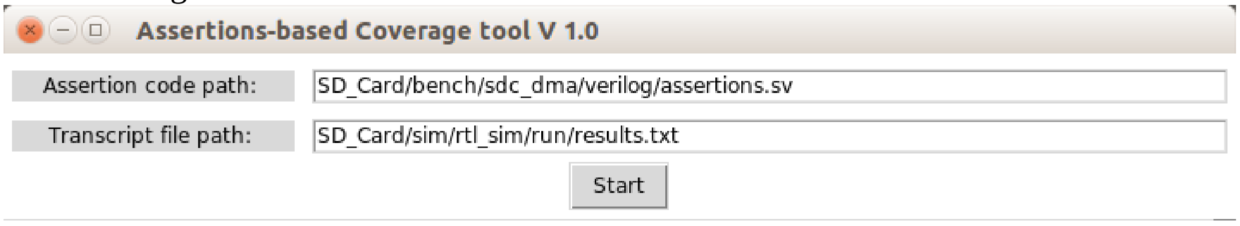
The GUI implementation was done by Python 2.7 tkinter package, as its ease of use, and compatibility with the core code of the tool.

The GUI Code has three functions; First, it forms the input interface for the tool with the entry boxes to enter the paths of assertions files and buttons to start the tool functionality. Second, It forms the output interface, which shows the coverage percentage in a colorful shape showing the percentage of each bin, in addition to each coverpoint. Finally, the GUI code is the master of all codes which are written as functions, so it controls the flow of the whole tool.

5.3.1 Version 1.0 of the Tool:

The input interface of version 1.0 is shown in Figure 5.3. It simply contains two entry boxes, and a button. When the user press the “Start” button, the tool will begin running the core code to calculate the coverage.

Figure 5.5 shows the input interface of the tool, the first Entry box is to enter the path of SystemVerilog Assertions file, while the second is to enter the path of the output text file from running the assertions.



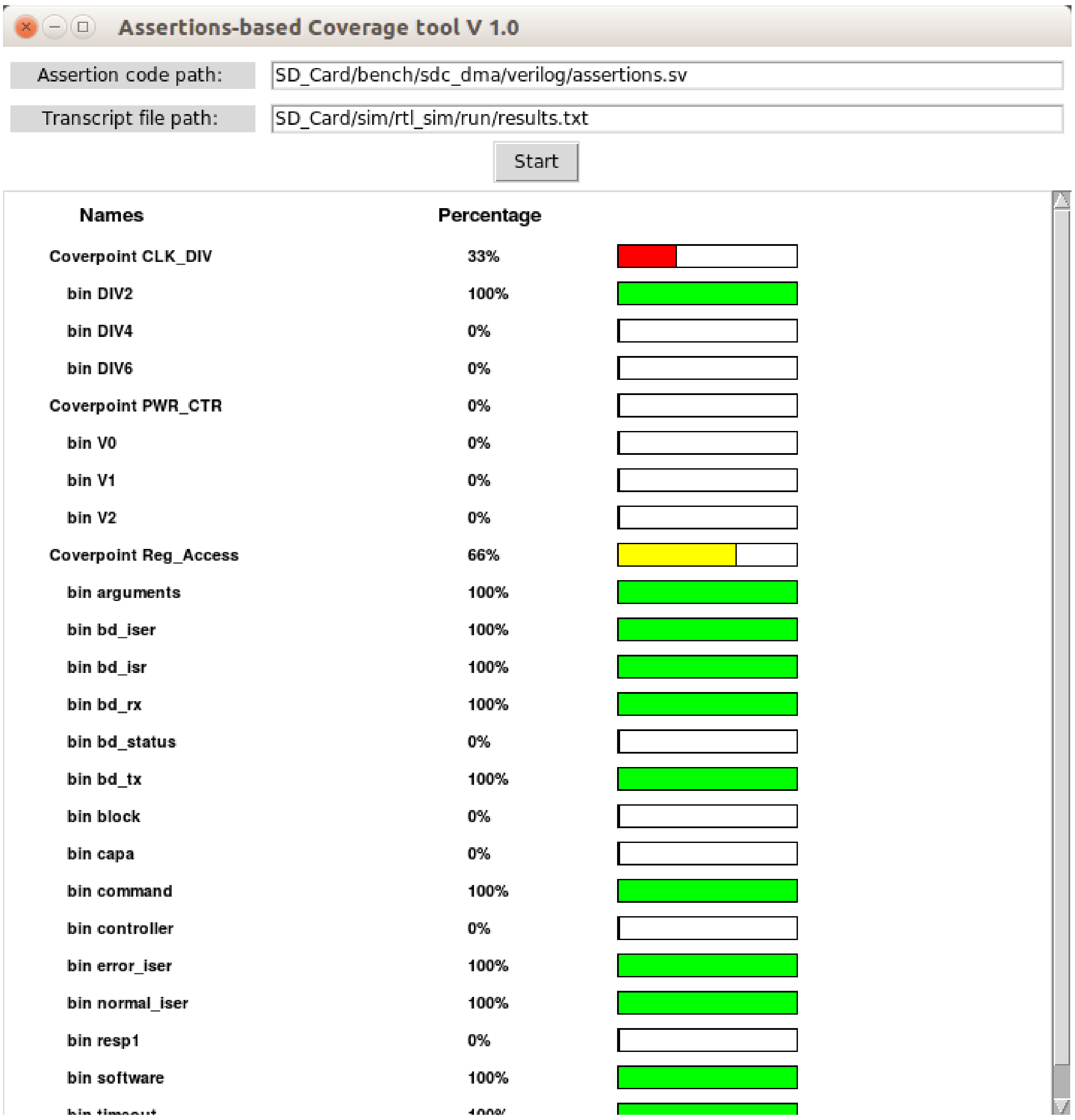
Assertion code path: SD_Card/bench/sdc_dma/verilog/assertions.sv

Transcript file path: SD_Card/sim/rtl_sim/run/results.txt

Figure 5.5 The input interface of the tool

After Clicking on “Start”, the tool will pass both inputs to the Core code, which will calculate the percentage of the coverage as explained in the previous chapter.

After calculating the coverage by the core code, the GUI reads the results from an external text file, then displays it as a colorful percentage boxes. Figure 5.6 shows the output for testing the wishbone interface of the SD-Card module with the code from figure 5.3.



Assertion code path: SD_Card/bench/sdc_dma/verilog/assertions.sv

Transcript file path: SD_Card/sim/rtl_sim/run/results.txt

Names	Percentage	
Coverpoint CLK_DIV	33%	🟥
bin DIV2	100%	🟩
bin DIV4	0%	☐
bin DIV6	0%	☐
Coverpoint PWR_CTR	0%	☐
bin V0	0%	☐
bin V1	0%	☐
bin V2	0%	☐
Coverpoint Reg_Access	66%	🟨
bin arguments	100%	🟩
bin bd_iser	100%	🟩
bin bd_isr	100%	🟩
bin bd_rx	100%	🟩
bin bd_status	0%	☐
bin bd_tx	100%	🟩
bin block	0%	☐
bin capa	0%	☐
bin command	100%	🟩
bin controller	0%	☐
bin error_iser	100%	🟩
bin normal_iser	100%	🟩
bin resp1	0%	☐
bin software	100%	🟩
bin timeout	100%	🟩

Figure 5.6 The Output of the GUI interface of the tool after running the code on the SD-Card controller.

6 Experimental Results

In this chapter, we will discuss the experimental results of running our tool, and compare it with the same code running on Questa Sim. The tool was tested with simple ALU, then the SD-Card Controller module discussed in chapter 4. Screenshots for running the ALU and the SD-Card Controller on the tool and Questa Sim are provided, in addition to the codes of both coverage and assertions.

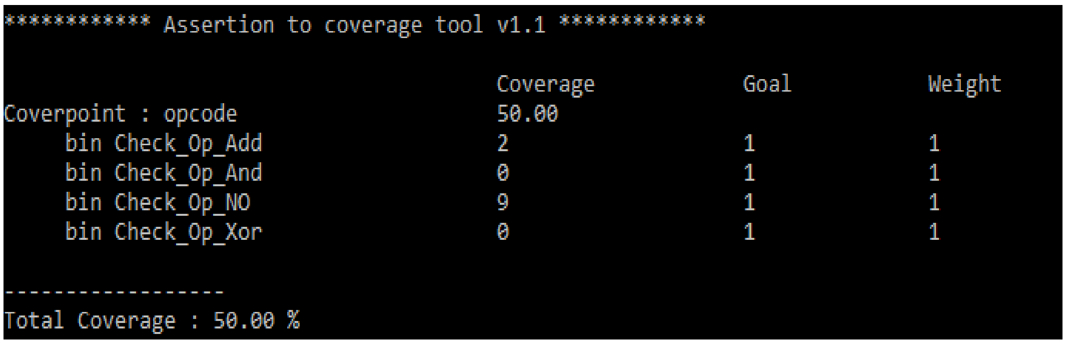
6.1 Tool Testing Methodology

To test the proposed tool and make sure that its results are the same as the results from other commercial tools, we wrote an ordinary coverage code and run it with Questa Sim, then, we wrote the equivalent assertion code to calculate the coverage of the same signals (bins) with our tool. After that, the results were compared, which show that the results are almost identical.

In the next subsections, all running codes of both coverage and assertions, and their corresponding output of the our tool and Questa Sim are shown.

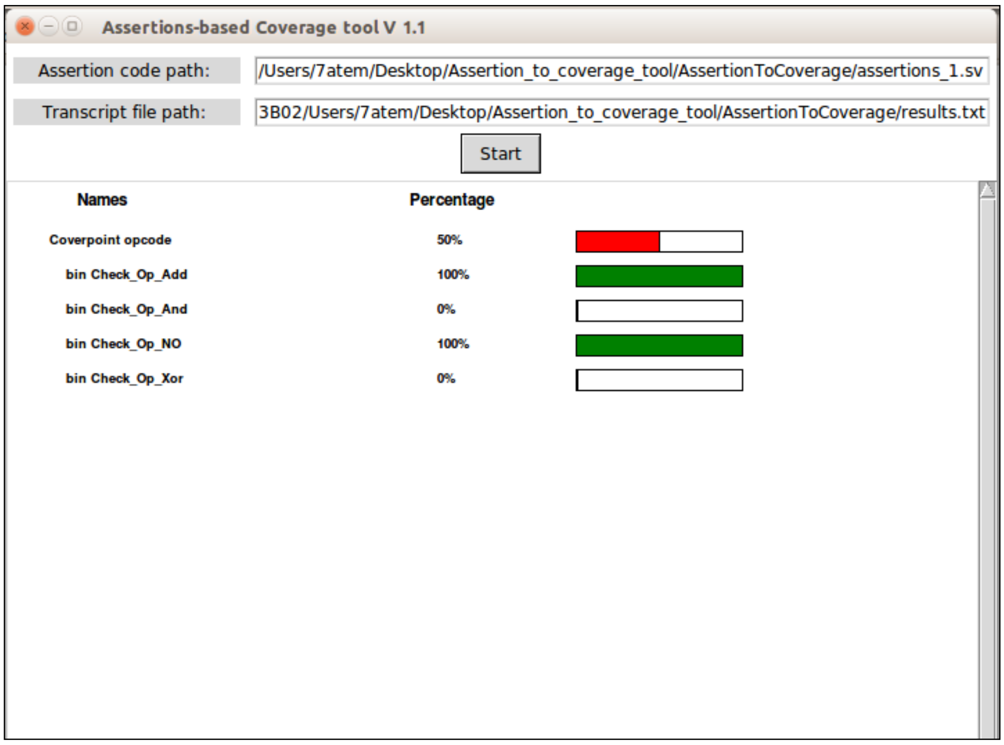
6.1.1 Simple ALU Testing

As a proof of concept, we first tested the tool with a simple ALU. The goal was to cover the control signal of it. The results of covering this module with the proposed tool is shown in figures 6.1 and 6.2.



```
************ Assertion to coverage tool v1.1 ************

                                         Coverage            Goal            Weight
Coverpoint : opcode                      50.00
     bin Check_Op_Add                    2                   1               1
     bin Check_Op_And                    0                   1               1
     bin Check_Op_NO                     9                   1               1
     bin Check_Op_Xor                    0                   1               1

------------------
Total Coverage : 50.00 %
```

Figure 6.1 The Output of the tool on the terminal (without GUI) after running the code on the simple ALU.



Assertions-based Coverage tool V 1.1

Assertion code path: /Users/7atem/Desktop/Assertion_to_coverage_tool/AssertionToCoverage/assertions_1.sv

Transcript file path: 3B02/Users/7atem/Desktop/Assertion_to_coverage_tool/AssertionToCoverage/results.txt

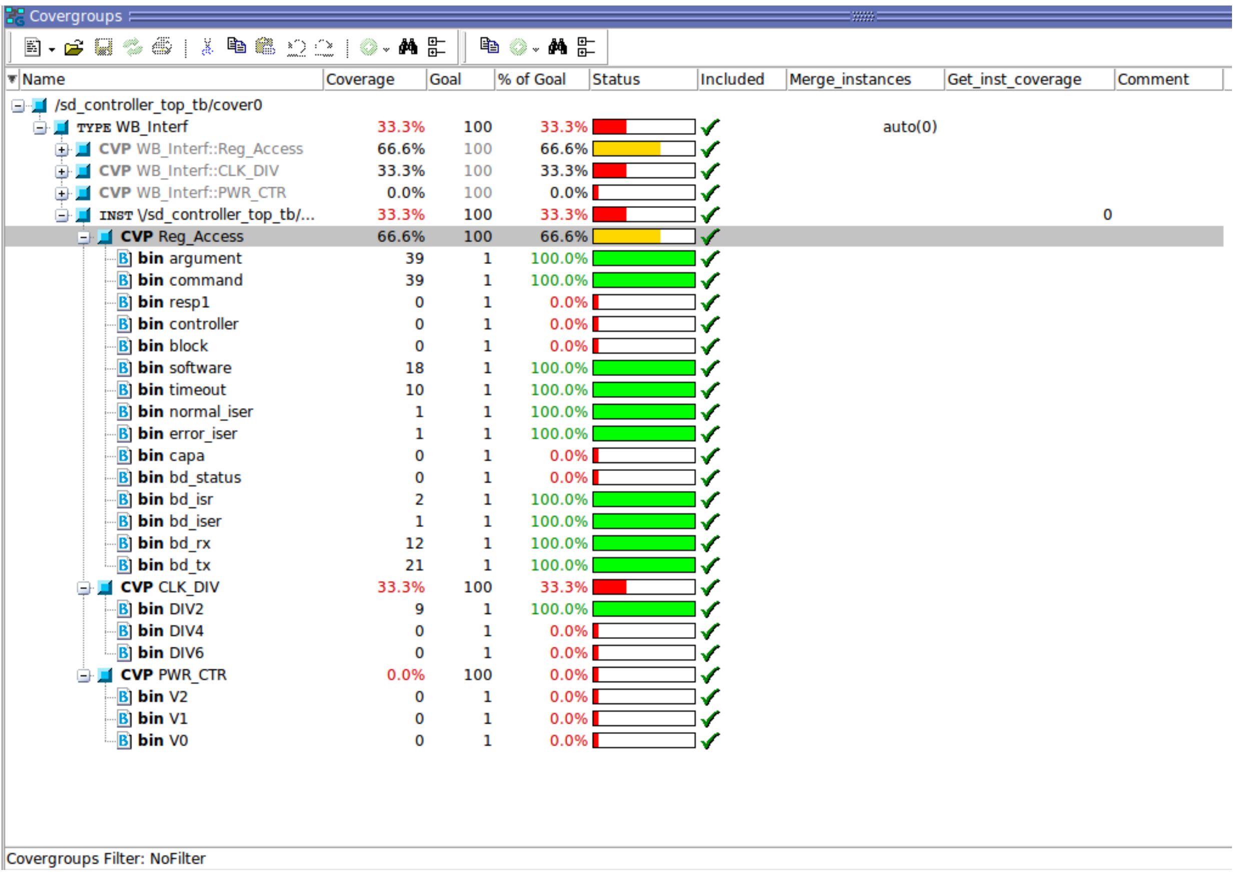
Start

Names	Percentage
Coverpoint opcode	50%
bin Check_Op_Add	100%
bin Check_Op_And	0%
bin Check_Op_NO	100%
bin Check_Op_Xor	0%

Figure 6.2 The Output of the GUI interface of the tool after running the code on the simple ALU.

6.1.2 SD-Card Controller Testing

For SD-Card Controller, we focused on the wishbone interface and tested many signals of it. We first wrote a coverage code for the Questa Sim, then assertions code for our tool.



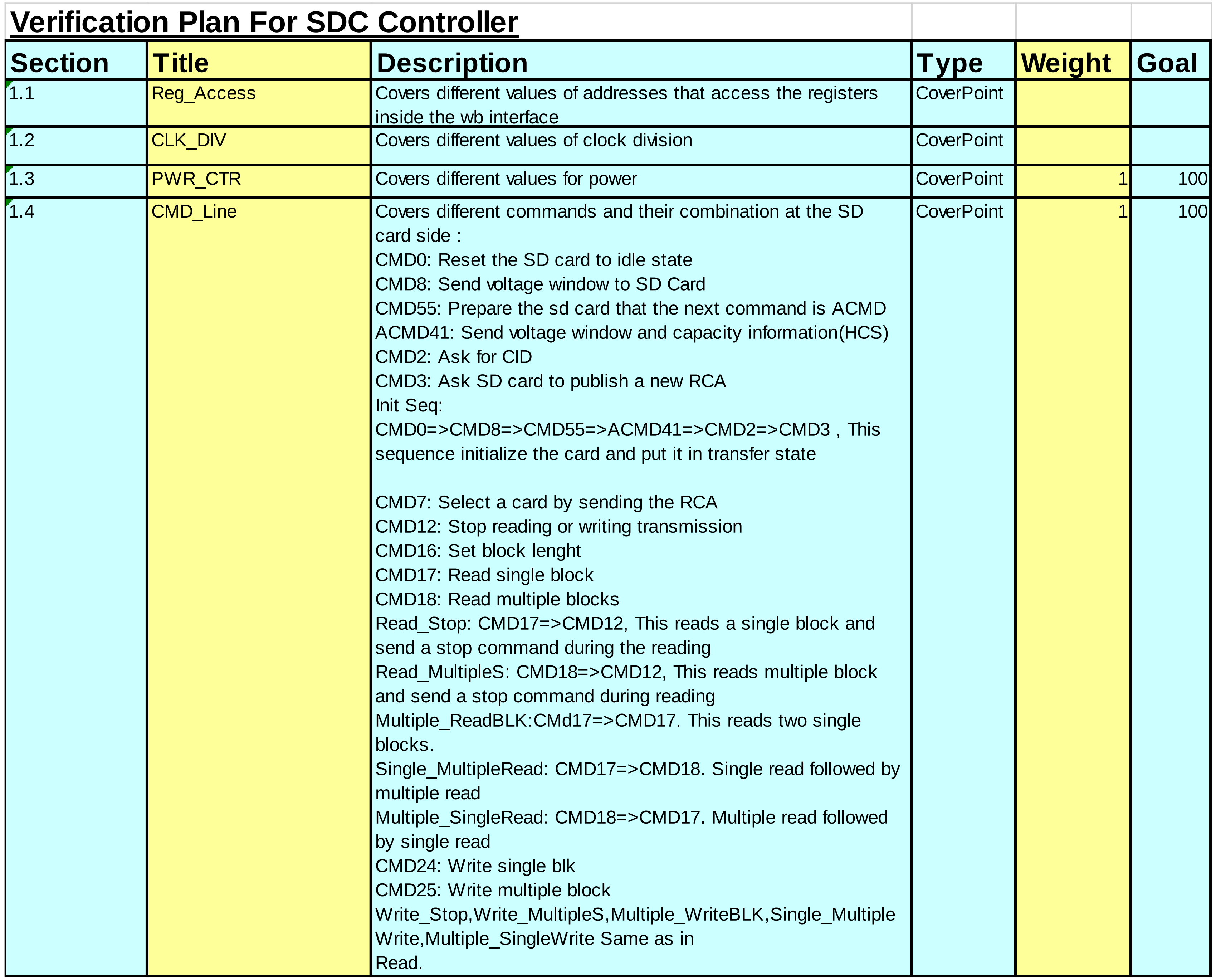
Covergroups

Name	Coverage	Goal	% of Goal	Status	Included	Merge_instances	Get_inst_coverage	Comment
/sd_controller_top_tb/cover0								
TYPE WB_Interf	33.3%	100	33.3%		✓	auto(0)		
CVP WB_Interf::Reg_Access	66.6%	100	66.6%		✓			
CVP WB_Interf::CLK_DIV	33.3%	100	33.3%		✓			
CVP WB_Interf::PWR_CTR	0.0%	100	0.0%		✓			
INST \/sd_controller_top_tb/...	33.3%	100	33.3%		✓		0	
CVP Reg_Access	66.6%	100	66.6%		✓			
bin argument	39	1	100.0%		✓			
bin command	39	1	100.0%		✓			
bin resp1	0	1	0.0%		✓			
bin controller	0	1	0.0%		✓			
bin block	0	1	0.0%		✓			
bin software	18	1	100.0%		✓			
bin timeout	10	1	100.0%		✓			
bin normal_iser	1	1	100.0%		✓			
bin error_iser	1	1	100.0%		✓			
bin capa	0	1	0.0%		✓			
bin bd_status	0	1	0.0%		✓			
bin bd_isr	2	1	100.0%		✓			
bin bd_iser	1	1	100.0%		✓			
bin bd_rx	12	1	100.0%		✓			
bin bd_tx	21	1	100.0%		✓			
CVP CLK_DIV	33.3%	100	33.3%		✓			
bin DIV2	9	1	100.0%		✓			
bin DIV4	0	1	0.0%		✓			
bin DIV6	0	1	0.0%		✓			
CVP PWR_CTR	0.0%	100	0.0%		✓			
bin V2	0	1	0.0%		✓			
bin V1	0	1	0.0%		✓			
bin V0	0	1	0.0%		✓			

Covergroups Filter: NoFilter

Figure 6.3 The Output of Questa Sim Coverage when testing the SD-Card Controller.

In figure 6.4, the coverage plan of wishbone interface in shown. In figure 6.5 the covergroup code is shown, which tests the wishbone interface of the SD-Card controller. The output of this coverage on Questa Sim is shown in Figure 6.3.



Verification Plan For SDC Controller

Section	Title	Description	Type	Weight	Goal
1.1	Reg_Access	Covers different values of addresses that access the registers inside the wb interface	CoverPoint		
1.2	CLK_DIV	Covers different values of clock division	CoverPoint		
1.3	PWR_CTR	Covers different values for power	CoverPoint	1	100
1.4	CMD_Line	Covers different commands and their combination at the SD card side : CMD0: Reset the SD card to idle state CMD8: Send voltage window to SD Card CMD55: Prepare the sd card that the next command is ACMD ACMD41: Send voltage window and capacity information(HCS) CMD2: Ask for CID CMD3: Ask SD card to publish a new RCA Init Seq: CMD0=>CMD8=>CMD55=>ACMD41=>CMD2=>CMD3 , This sequence initialize the card and put it in transfer state CMD7: Select a card by sending the RCA CMD12: Stop reading or writing transmission CMD16: Set block lenght CMD17: Read single block CMD18: Read multiple blocks Read_Stop: CMD17=>CMD12, This reads a single block and send a stop command during the reading Read_MultipleS: CMD18=>CMD12, This reads multiple block and send a stop command during reading Multiple_ReadBLK:CMd17=>CMD17. This reads two single blocks. Single_MultipleRead: CMD17=>CMD18. Single read followed by multiple read Multiple_SingleRead: CMD18=>CMD17. Multiple read followed by single read CMD24: Write single blk CMD25: Write multiple block Write_Stop,Write_MultipleS,Multiple_WriteBLK,Single_Multiple Write,Multiple_SingleWrite Same as in Read.	CoverPoint	1	100

Figure 6.4 The Coverage Plan of SD-Card wishbone interface.

```
covergroup WB_Interf;
     option.per_instance=1;

     Reg_Access : coverpoint wb_adr_i iff(wb_we_i)
     {
          bins argument={8'h00};
          bins command={8'h04};
          bins resp1={8'h0c};
          bins controller={8'h1c};
          bins block={8'h20};
          bins software={8'h28};
          bins timeout={8'h2c};
          bins normal_iser={8'h38};
          bins error_iser={8'h3c};
          bins capa={8'h48};
          bins bd_status={8'h50};
          bins bd_isr={8'h54};
          bins bd_iser={8'h58};
          bins bd_rx={8'h60};
          bins bd_tx={8'h80};
     }

     CLK_DIV: coverpoint clk_divider iff(wb_we_i&&(wb_adr_i==8'h4c))
     {
          bins DIV2={8'd0};
          bins DIV4={8'd1};
          bins DIV6={8'd2};
     }

     PWR_CTR: coverpoint power_control iff(wb_we_i&&(wb_adr_i==8'h24))
     {
          bins V2={8'd7};
          bins V1={8'd6};
          bins V0={8'd5};
     }

endgroup
```

Figure 6.5 The Coverage code of the SD-Card Controller.

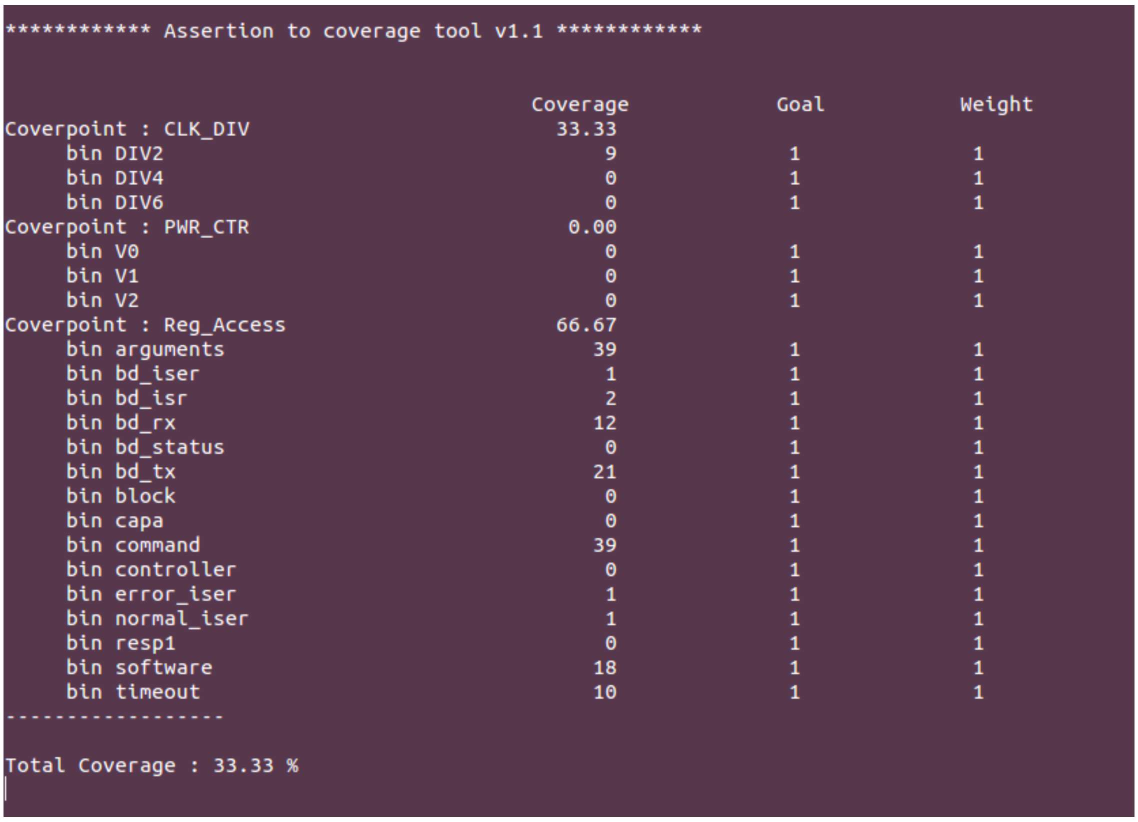
To test our tool, we needed to write an assertions code for the same signals as the coverage, then run our code and compare the results.

The assertions code is shown in figure 6.6, and the output of the tool on Linux terminal and on the tool GUI are shown in figure 6.7.

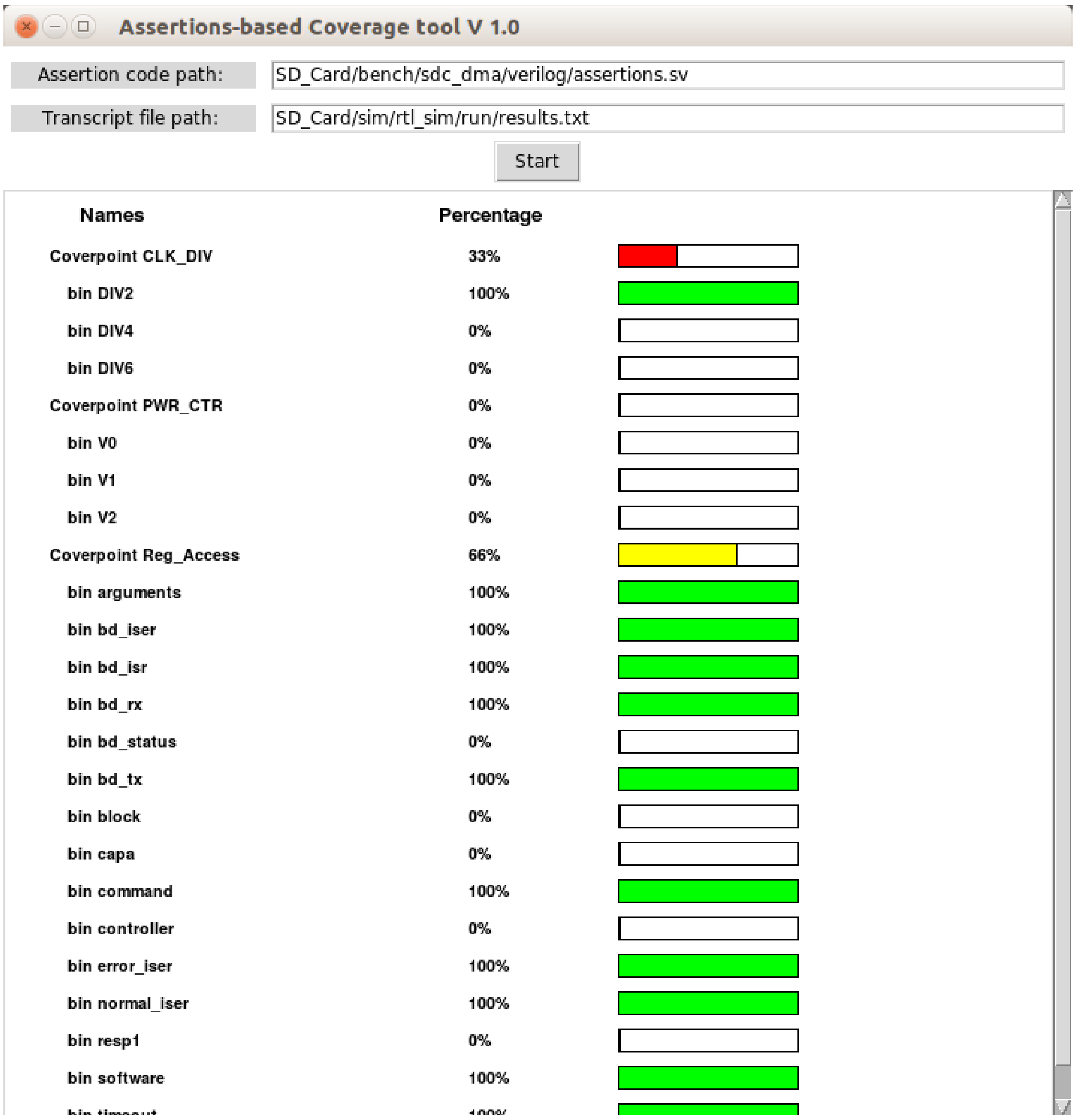
```
//////////////////// Properties of covergroup WB_Interf
//////////////////
// coverpoint: Reg_Access


property arguments;
@(negedge clk && wb_we_i) (wb_adr_i == 8'h00) |-> z == 0;
endproperty

property command;
@(negedge clk && wb_we_i) (wb_adr_i == 8'h04) |-> z == 0;
endproperty

property block;
@(negedge clk && wb_we_i) (wb_adr_i == 8'h20) |-> z == 0;
endproperty

 property resp1;
 @(negedge clk && wb_we_i) (wb_adr_i == 8'h0c) |-> z == 0;
 endproperty

property software;
@(negedge clk && wb_we_i) (wb_adr_i == 8'h28) |-> z == 0;
endproperty

property timeout;
@(negedge clk && wb_we_i) (wb_adr_i == 8'h2c) |-> z == 0;
endproperty

property controller;
@(negedge clk && wb_we_i) (wb_adr_i == 8'h1c) |-> z == 0;
endproperty

property normal_iser;
@(negedge clk && wb_we_i) (wb_adr_i == 8'h38) |-> z == 0;
endproperty

property error_iser;
@(negedge clk && wb_we_i) (wb_adr_i == 8'h3c) |-> z == 0;
endproperty

property capa;
@(negedge clk && wb_we_i) (wb_adr_i == 8'h48) |-> z == 0;
endproperty

property bd_status;
@(negedge clk && wb_we_i) (wb_adr_i == 8'h50) |-> z == 0;
endproperty

property bd_isr;
@(negedge clk && wb_we_i) (wb_adr_i == 8'h54) |-> z == 0;
endproperty

property bd_iser;
@(negedge clk && wb_we_i) (wb_adr_i == 8'h58) |-> z == 0;
endproperty

property bd_rx;
@(negedge clk && wb_we_i) (wb_adr_i == 8'h60) |-> z == 0;
Endproperty
```

```
///////////////// Assertions of covergroup WB_Interf ////////////////////
// coverpoint: Reg_Access


arguments_as: assert property (arguments)
 $fdisplay(sd_controller_top_tb.file, "Reg_Access: arguments passed") ;

command_as: assert property (command)
 $fdisplay(sd_controller_top_tb.file,"Reg_Access: command passed") ;

block_as: assert property (block)
 $fdisplay(sd_controller_top_tb.file,"Reg_Access: block passed") ;

resp1_as : assert property (resp1 )
 $fdisplay(sd_controller_top_tb.file,"Reg_Access: resp1 passed") ;

software_as : assert property (software)
 $fdisplay(sd_controller_top_tb.file,"Reg_Access: software passed") ;

timeout_as : assert property (timeout)
 $fdisplay(sd_controller_top_tb.file,"Reg_Access: timeout passed") ;

controller_as : assert property (controller)
 $fdisplay(sd_controller_top_tb.file,"Reg_Access: controller passed") ;

normal_iser_as : assert property (normal_iser)
 $fdisplay(sd_controller_top_tb.file,"Reg_Access: normal_iser passed") ;

error_iser_as : assert property (error_iser)
 $fdisplay(sd_controller_top_tb.file,"Reg_Access: error_iser passed") ;

capa_as : assert property (capa)
 $fdisplay(sd_controller_top_tb.file,"Reg_Access: capa passed") ;

bd_status_as : assert property (bd_status)
 $fdisplay(sd_controller_top_tb.file,"Reg_Access: bd_status passed") ;

bd_isr_as : assert property (bd_isr)
 $fdisplay(sd_controller_top_tb.file,"Reg_Access: bd_isr passed") ;

bd_iser_as : assert property (bd_iser)
 $fdisplay(sd_controller_top_tb.file,"Reg_Access: bd_iser passed") ;

bd_rx_as : assert property (bd_rx)
 $fdisplay(sd_controller_top_tb.file,"Reg_Access: bd_rx passed") ;

bd_tx_as : assert property (bd_tx)
 $fdisplay(sd_controller_top_tb.file,"Reg_Access: bd_tx passed") ;
```

Figure 6.6 A real code snippet of signals from the SystemVerilog assertions code of the wishpone interface in the SD-Card module, this code shows the code of coverpoint “Ref_Access” from the coverage code.



```
************ Assertion to coverage tool v1.1 ************


                                           Coverage            Goal            Weight
Coverpoint : CLK_DIV                         33.33
     bin DIV2                                    9               1                1
     bin DIV4                                    0               1                1
     bin DIV6                                    0               1                1
Coverpoint : PWR_CTR                          0.00
     bin V0                                      0               1                1
     bin V1                                      0               1                1
     bin V2                                      0               1                1
Coverpoint : Reg_Access                      66.67
     bin arguments                              39               1                1
     bin bd_iser                                 1               1                1
     bin bd_isr                                  2               1                1
     bin bd_rx                                  12               1                1
     bin bd_status                               0               1                1
     bin bd_tx                                  21               1                1
     bin block                                   0               1                1
     bin capa                                    0               1                1
     bin command                                39               1                1
     bin controller                              0               1                1
     bin error_iser                              1               1                1
     bin normal_iser                             1               1                1
     bin resp1                                   0               1                1
     bin software                               18               1                1
     bin timeout                                10               1                1
------------------

Total Coverage : 33.33 %
```



Assertions-based Coverage tool V 1.0

Assertion code path: SD_Card/bench/sdc_dma/verilog/assertions.sv

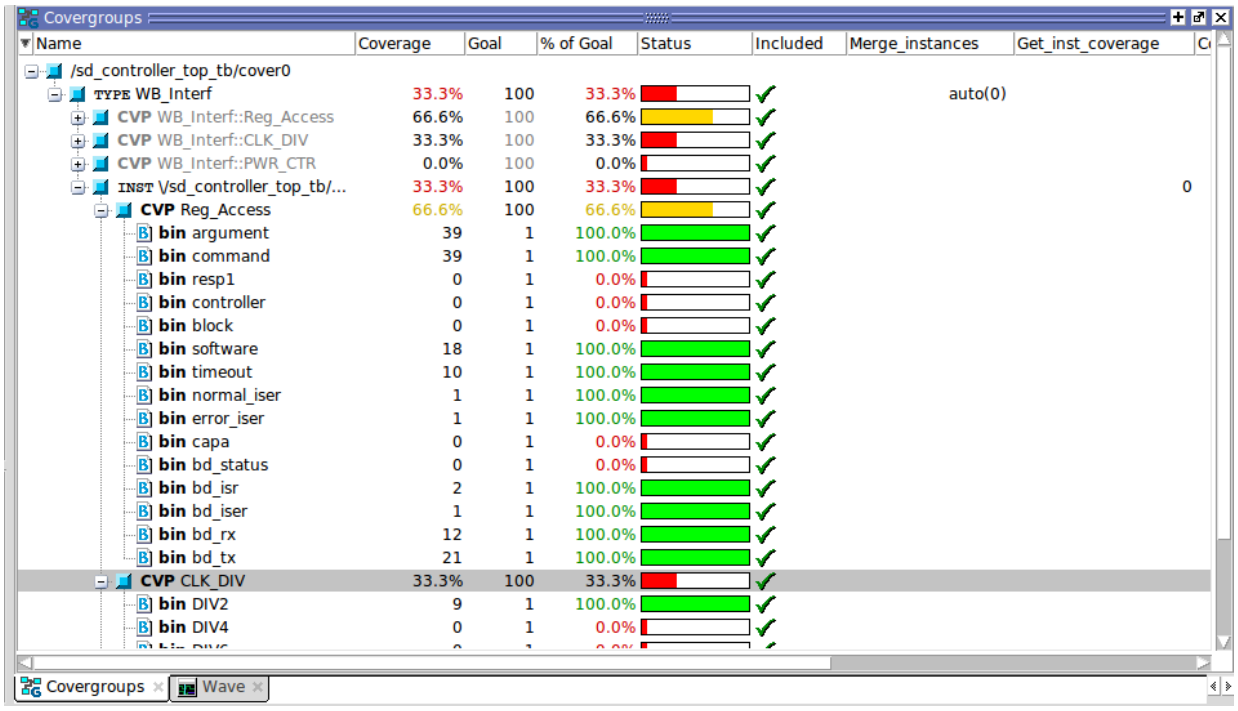
Transcript file path: SD_Card/sim/rtl_sim/run/results.txt

Start

Names	Percentage	
Coverpoint CLK_DIV	33%	🟥
bin DIV2	100%	🟩
bin DIV4	0%	
bin DIV6	0%	
Coverpoint PWR_CTR	0%	
bin V0	0%	
bin V1	0%	
bin V2	0%	
Coverpoint Reg_Access	66%	🟨
bin arguments	100%	🟩
bin bd_iser	100%	🟩
bin bd_isr	100%	🟩
bin bd_rx	100%	🟩
bin bd_status	0%	
bin bd_tx	100%	🟩
bin block	0%	
bin capa	0%	
bin command	100%	🟩
bin controller	0%	
bin error_iser	100%	🟩
bin normal_iser	100%	🟩
bin resp1	0%	
bin software	100%	🟩

Figure 6.7 The output of our tool in GUI, and Linux terminal.

To take a closer look and focus on certain signals, Let's take the first coverpoint from the coverage code, then see its output on both Questa Sim and our tool. The first coverpoint is named Reg_Access, and Figure 6.8 shows its bins on Questa Sim, while Figure 6.9 shows its bins on our tool. As obvious from the figures, the results of Reg_Access covering from both are identical.

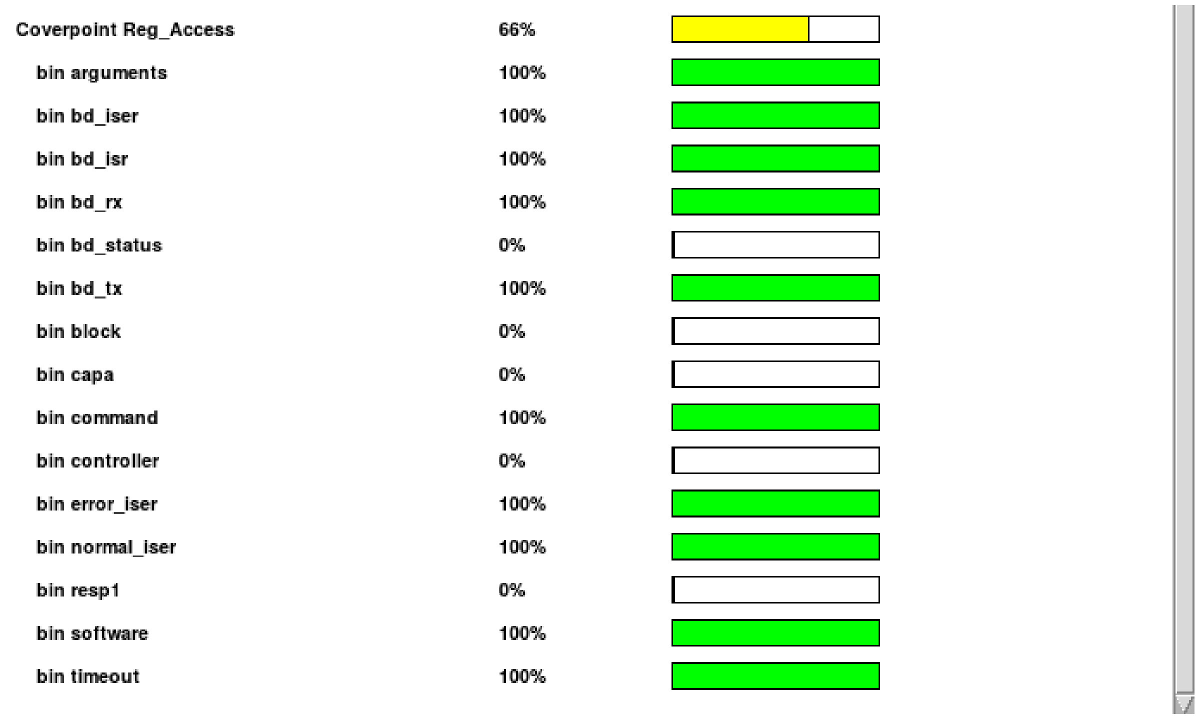


Covergroups

Name	Coverage	Goal	% of Goal	Status	Included	Merge_instances	Get_inst_coverage
/sd_controller_top_tb/cover0							
TYPE WB_Interf	33.3%	100	33.3%	🟥	✓	auto(0)	
CVP WB_Interf::Reg_Access	66.6%	100	66.6%	🟨	✓		
CVP WB_Interf::CLK_DIV	33.3%	100	33.3%	🟥	✓		
CVP WB_Interf::PWR_CTR	0.0%	100	0.0%	🟥	✓		
INST \/sd_controller_top_tb/...	33.3%	100	33.3%	🟥	✓		0
CVP Reg_Access	66.6%	100	66.6%	🟨	✓		
bin argument	39	1	100.0%	🟩	✓		
bin command	39	1	100.0%	🟩	✓		
bin resp1	0	1	0.0%	🟥	✓		
bin controller	0	1	0.0%	🟥	✓		
bin block	0	1	0.0%	🟥	✓		
bin software	18	1	100.0%	🟩	✓		
bin timeout	10	1	100.0%	🟩	✓		
bin normal_iser	1	1	100.0%	🟩	✓		
bin error_iser	1	1	100.0%	🟩	✓		
bin capa	0	1	0.0%	🟥	✓		
bin bd_status	0	1	0.0%	🟥	✓		
bin bd_isr	2	1	100.0%	🟩	✓		
bin bd_iser	1	1	100.0%	🟩	✓		
bin bd_rx	12	1	100.0%	🟩	✓		
bin bd_tx	21	1	100.0%	🟩	✓		
CVP CLK_DIV	33.3%	100	33.3%	🟥	✓		
bin DIV2	9	1	100.0%	🟩	✓		
bin DIV4	0	1	0.0%	🟥	✓		

Covergroups | Wave

Figure 6.8 shows the coverage of Reg_Access coverpoint on Questa sim



Coverpoint Reg_Access	66%	🟨
bin arguments	100%	🟩
bin bd_iser	100%	🟩
bin bd_isr	100%	🟩
bin bd_rx	100%	🟩
bin bd_status	0%	⬜
bin bd_tx	100%	🟩
bin block	0%	⬜
bin capa	0%	⬜
bin command	100%	🟩
bin controller	0%	⬜
bin error_iser	100%	🟩
bin normal_iser	100%	🟩
bin resp1	0%	⬜
bin software	100%	🟩
bin timeout	100%	🟩

Figure 6.9 shows the coverage of Reg_Access coverpoint on the proposed tool.

7 Future Work and Conclusion

7.1 Future Work

Our Future work is a more mature version of the tool that might be able to “predict” the needed tests that would fill the gap in the coverage results, which would represent a great leap in the proposed verification environment.

Another important modification is the automatic mode of the tool. In the new version it will be able to generate the assertions for the users by allowing them to input the coverpoints and bins names, then it will automatically generate the assertions for them. It would be able to generate complex sequences, that allows complex protocol checking and cover more complex scenarios.

The general goal of all these modifications is to allow the tool to automatically achieve high coverage ratio and be able to find the signals that need to be covered. More about the general ideas to enhance the tool capabilities is explained in Fig 7.1

7.1.1 Automatic Test Cases Generation

Our tool is far from reaching its full potential. The tool can be extended to have more and more features and one of the strongest features that can be added is the Tool can generate automatic test cases for covering the holes in coverage.

The tool already have the coverage data and the tool can be made to detect the holes in this coverage data and generate test cases to cover those holes. Figure 7.1 shows the extended flow chart for the tool after adding the new feature

7.1.2 Automatic Assertions Generation

Another promising modification is the automatic assertions generation. As the verification engineers are used to use Coverage codes, it may be hard at the beginning to use assertions as coverage, so we would propose an easier interface to automatically generate the code for them.

Figure 7.2 shows the proposed input interface, in which the user is almost completing a coverage code.

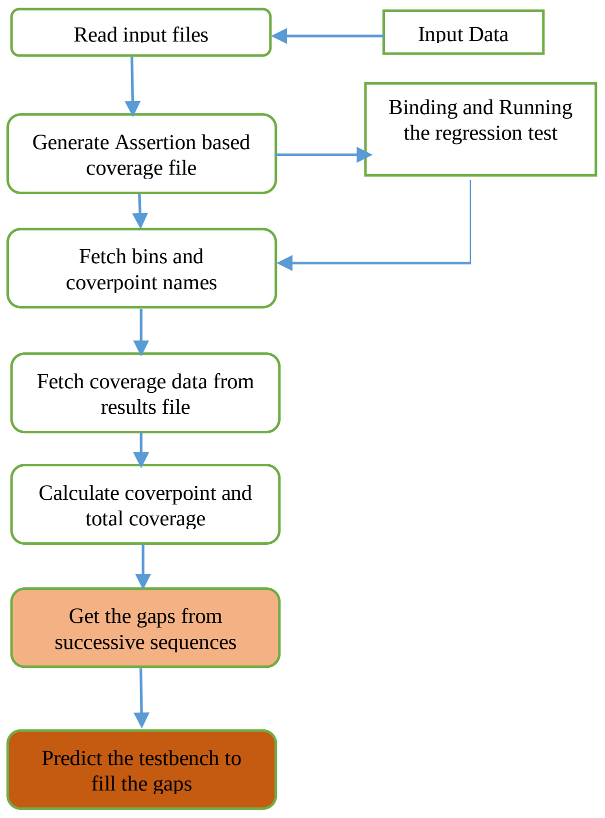
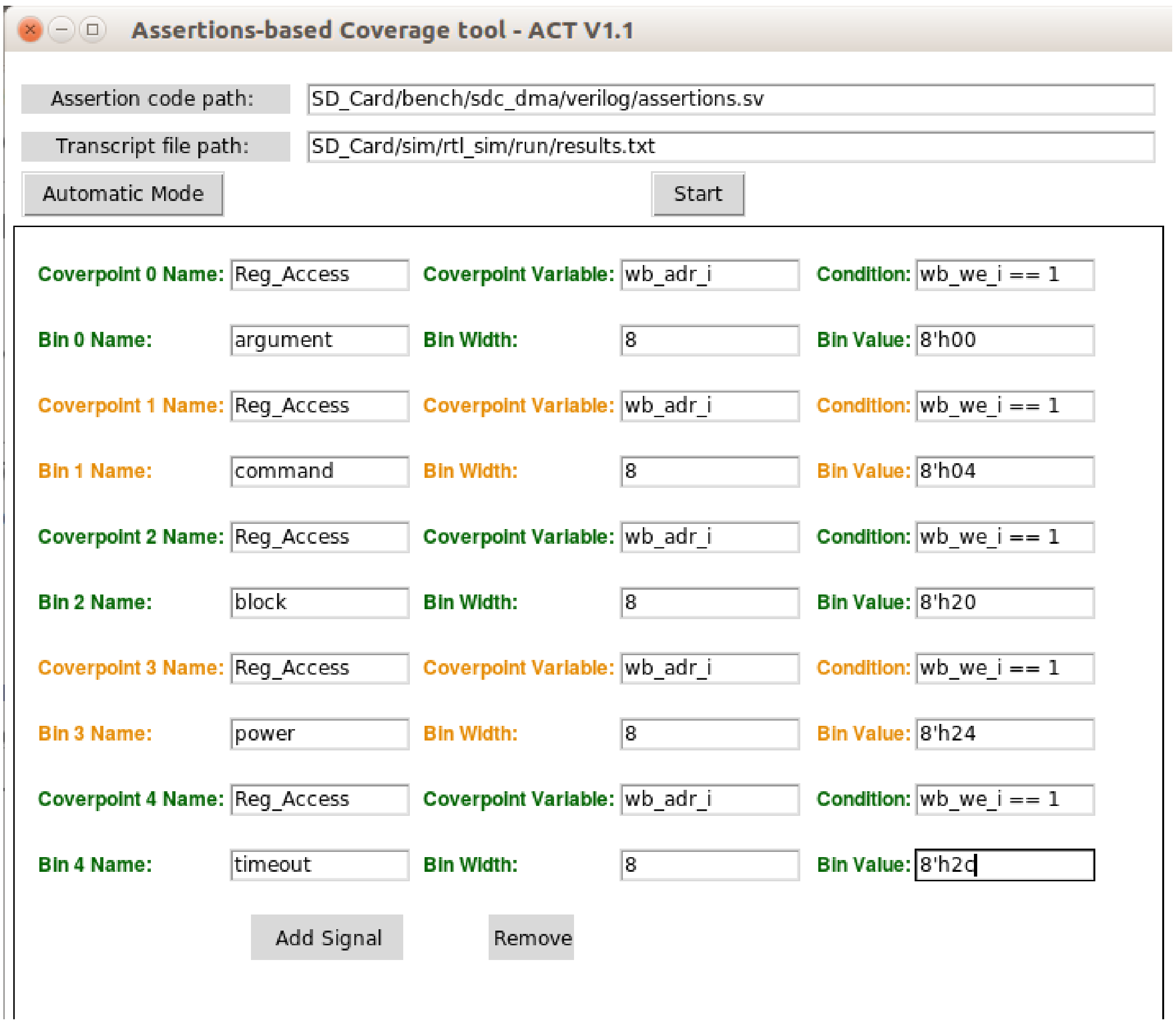

Read input files
Input Data
Generate Assertion based coverage file
Binding and Running the regression test
Fetch bins and coverpoint names
Fetch coverage data from results file
Calculate coverpoint and total coverage
Get the gaps from successive sequences
Predict the testbench to fill the gaps

Figure 7.1 our next step for an Auto-coverage Complete Tool.



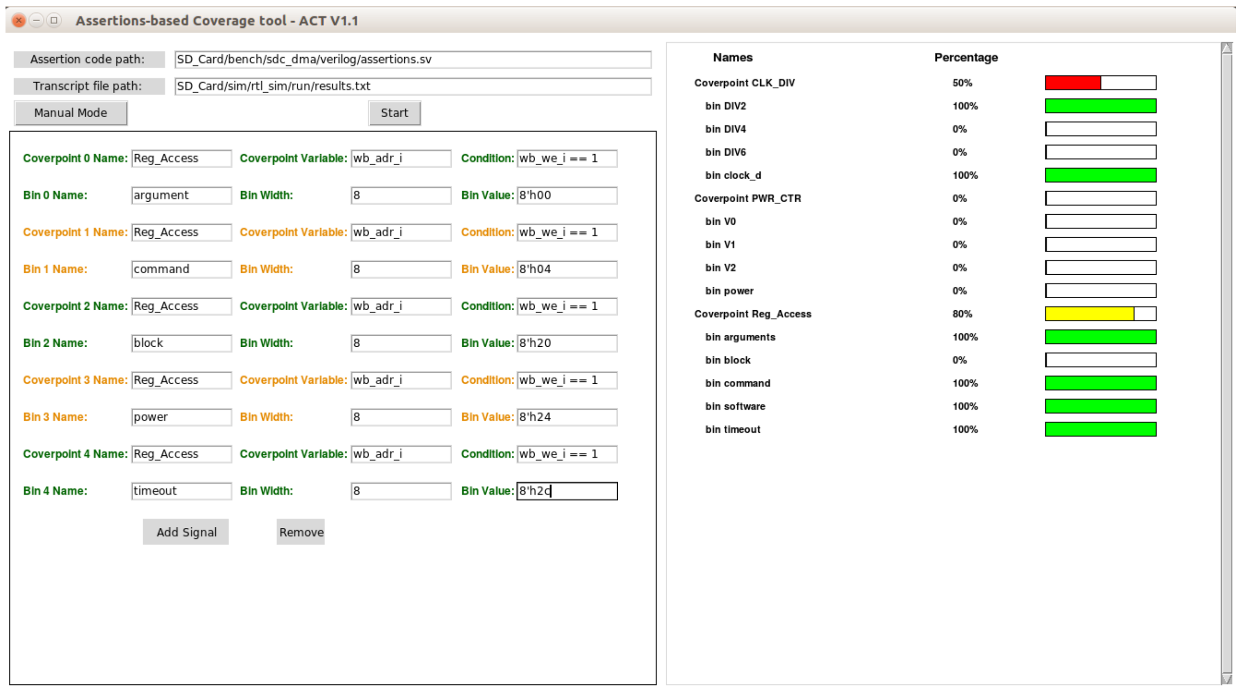
Assertion code path:	SD_Card/bench/sdc_dma/verilog/assertions.sv				
Transcript file path:	SD_Card/sim/rtl_sim/run/results.txt				
Coverpoint 0 Name:	Reg_Access	Coverpoint Variable:	wb_adr_i	Condition:	wb_we_i == 1
Bin 0 Name:	argument	Bin Width:	8	Bin Value:	8'h00
Coverpoint 1 Name:	Reg_Access	Coverpoint Variable:	wb_adr_i	Condition:	wb_we_i == 1
Bin 1 Name:	command	Bin Width:	8	Bin Value:	8'h04
Coverpoint 2 Name:	Reg_Access	Coverpoint Variable:	wb_adr_i	Condition:	wb_we_i == 1
Bin 2 Name:	block	Bin Width:	8	Bin Value:	8'h20
Coverpoint 3 Name:	Reg_Access	Coverpoint Variable:	wb_adr_i	Condition:	wb_we_i == 1
Bin 3 Name:	power	Bin Width:	8	Bin Value:	8'h24
Coverpoint 4 Name:	Reg_Access	Coverpoint Variable:	wb_adr_i	Condition:	wb_we_i == 1
Bin 4 Name:	timeout	Bin Width:	8	Bin Value:	8'h2c

Figure 7.2 Input interface of the automatic assertion generation modification.

After the new modification, The tool has two modes, the manual mode and Automatic mode. In manual mode the user writes the SystemVerilog Assertions code, compile and run it on any tool (for example: Questa or VCS), then write the paths of both the assertions file (.sv) and the external text file (.txt) in the entry boxes in our tool.

After that, the GUI code pass them to the core function which parses the assertions file to get the properties names (representing cover pins), and then the external text file to count the signals names. Finally, it calculates the coverage percentage for every pin; sum them to calculate the coverage of each cover point, and then outputs all of them to the terminal, as well as to another text file; the GUI code reads this file and displays the names of the cover points and pins with a colorful output of their percentage.

The second mode is the Automatic mode. In this mode, the user enters the names, widths, and values of the properties and signals to be covered, and then our tool will automatically generate the assertions code, run it using Questa Sim in patch mode through the python system task, then use the generated assertions and text files and continue the same steps as in the Manual mode. Figure 7.3 shows the complete modified tool after running it in Manual mode.


Assertions-based Coverage tool - ACT V1.1

Assertion code path:	SD_Card/bench/sdc_dma/verilog/assertions.sv
Transcript file path:	SD_Card/sim/rtl_sim/run/results.txt

Manual Mode

Start

Coverpoint 0 Name:	Reg_Access	Coverpoint Variable:	wb_adr_i	Condition:	wb_we_i == 1
Bin 0 Name:	argument	Bin Width:	8	Bin Value:	8'h00
Coverpoint 1 Name:	Reg_Access	Coverpoint Variable:	wb_adr_i	Condition:	wb_we_i == 1
Bin 1 Name:	command	Bin Width:	8	Bin Value:	8'h04
Coverpoint 2 Name:	Reg_Access	Coverpoint Variable:	wb_adr_i	Condition:	wb_we_i == 1
Bin 2 Name:	block	Bin Width:	8	Bin Value:	8'h20
Coverpoint 3 Name:	Reg_Access	Coverpoint Variable:	wb_adr_i	Condition:	wb_we_i == 1
Bin 3 Name:	power	Bin Width:	8	Bin Value:	8'h24
Coverpoint 4 Name:	Reg_Access	Coverpoint Variable:	wb_adr_i	Condition:	wb_we_i == 1
Bin 4 Name:	timeout	Bin Width:	8	Bin Value:	8'h2c

Add Signal

Remove

Names	Percentage	
Coverpoint CLK_DIV	50%	🟥
bin DIV2	100%	🟩
bin DIV4	0%	
bin DIV6	0%	
bin clock_d	100%	🟩
Coverpoint PWR_CTR	0%	
bin V0	0%	
bin V1	0%	
bin V2	0%	
bin power	0%	
Coverpoint Reg_Access	80%	🟨
bin arguments	100%	🟩
bin block	0%	
bin command	100%	🟩
bin software	100%	🟩
bin timeout	100%	🟩

Figure 7.3 Input interface of the automatic assertion generation modification, with the output of Manual mode.

7.2 Conclusion

A novel method to calculate functional coverage using SystemVerilog Assertions is proposed, with a complete open source CAD tool that exploits the capabilities of the proposed method.

The tool was designed using Python, and its output is shown in the terminal, in addition to a handy colorful GUI. The output completely shows the coverage of the DUT with all bins details.

The tool is tested by a simple ALU, and an open source SD-Card Controller. The tool results were also compared to the coverage tool of Questa Sim, which yielded the same results. In addition, all features of the proposed tool with the aid of GUI is explained.

References

[1] Bruce Wile, John Goss, Wolfgang Roesner, Comprehensive Functional Verification, The Complete Industry Cycle, 2005

[2] G. E. Moore, "Cramming More Components Onto Integrated Circuits," in ***Proceedings of the IEEE***, vol. 86, no. 1, pp. 82-85, Jan. 1998.

[3] Pedro Araujo, (2014) Development of a reconfigurable multi-protocol verification environment using UVM methodology, Master's thesis, Universidade do porto.

[4] Janick Bergeron. Writing testbenches: functional verification of HDL models. Kluwer Academic Publishers, 2003.

[5] Chris Spears. SystemVerilog for Verification: A Guide to Learning the Testbench Language Features. Springer, 2007.

[6] Mentor Graphics, Coverage Cookbook Complete Verification Academy

[7] Ashok B. Mehta SystemVerilog Assertions and Functional Coverage, Guide to Language, Methodology and Applications, Springer Verlag New York (2014) NoRestriction.

[8] Ray Salemi, The UVM Primer an introduction to universal verification methodology.

[9] SD committee, Simplified_Physical_Layer_Spec-1

[10] Opencores SD-Card protocol.