KTH - Royal Institute of Technology

Methodical Validation and verification of FPGA-code

Master’s Thesis at ABB Robotics

Author: Babak Khodayari
Examinor professor: Dr. Ahmed Hemani
Supervisor: Stefan Westberg

December 22 , 2014

TRITA-ICT-EX-2015:3

This Page is Intentionally Left Blank

Abstract

Proper verification of FPGA-code requires knowledge, skills, tools and resources. ABB Robotics uses FPGA technology in their robot control system. The increasing complexity of their FPGA designs requires increasingly more advanced verification methods.

This provides an appropriate research on methodologies, languages and tools for more detailed evaluation based on ABB Robotics requirements and possibilities. The thesis demonstrates the chosen methods, languages and tools for verification of a FPGA-design by verification methods that are state of the art.

The verification environment such as functional verification, open source VHDL verification methodology (OSVVM), and universal verification methodology (UVM) were investigated in practical tests followed by an evaluation of advantages and disadvantages of the tests according the company requirements. This provides the verification teams with different test environments and presents available options for verification development and future work.

Acknowledgements

Stefan Westberg deserves the great thanks for his supportive supervision of my thesis. It was working with him that made writing this thesis a fascinating experience.
I appreciate the time he spend mentoring this thesis during every steps of this work and reviewing the final report.

I owe special thanks to my professor Dr.Ahmed Hemani, who encouraged me during this research. He generously shared his knowledge of verification with me during my studies at Royal Institute of Technology KTH.

I would like to express my deep gratitude to Per V. Carlsson Manager Controller HW at ABB Robotics for the unique opportunity that he gave to me while doing this thesis and providing me with all requisite.

I express my warm thanks to Tiberiu Seceleanu for his guidance at ABB Corporate Research and Jang Gao at ABB Robotics China.

Babak Khodayari

Table of Contents

Table of figures

Chapter 1

Introduction

FPGA devices have been developed rapidly in the last decade. It became very complex for provide higher speed and larger memory in the industry. They require advanced verification technology for test and debugging. To conduct an advanced verification, it needs a research in this area.
ABB Robotics delivers the Robots with high-quality using complex FPGA designs that need verification with advanced methodologies.

1.1 Background

As much as FPGA designs develop in the electronic industries, validation and verification of the designs depend more on the methodologies, languages and tools.

1.2 Problem

Every methodology, tool and language for validation and verification has its advantages and disadvantages. Some of them are designed for small project and some are appropriate for large and complex projects.
ABB Robotics does not have a proper method for validation of the complex FPGA projects. They need a wide study in the area to find an appropriate validation method for their projects.

1.3 Purpose

ABB Robotics uses FPGA in their robot control system. The FPGA designs needs advanced verification methods.
The purpose of the thesis is to research and investigate the different methodologies, languages and tools for verification based on ABB Robotics requirements and possibilities.

1.4 Goal, Benefits, Ethics and Sustainability

The goal is to get an appropriate list of techniques for more detailed evaluation and report the evaluation of the list of the techniques. Chosen technique must be demonstrated practically to find a proper way of verification of FPGA code
It will be done by following tasks:

- To show that what methods are state of the art.
- Simulation
- Writing the report
- Presentation

By conducting this thesis ABB Robotics will have different test environment with advanced verification methodologies. All the thesis work at ABB must follow ABBs code of conduct.

1.5 Methodology / Methods

Different methodologies will be chosen for doing the research and thesis work. According to the following reasons different methods that are appropriate to this thesis were studied and examined.
The company has its abilities and limitations for cost and resources. The technique, tools and languages for verification of FPGA also have its advantages and disadvantages. The techniques that are going to be used must fulfil the company’s requirement and match its possibilities and limitations.
Methodologies are studied in theory and were used in the implementation part practically. The outcome of the thesis is the comparison of the features of different methodologies beside languages and tools.

1.6 Outline

This report is structured as follows. The second chapter is covers theory of verification languages methodologies. Tools are given with a short review for each of them.
Furthermore constraint random generation, functional verification, Open source VHDL verification, Universal verification methodology were studied in this chapter.
Chapter three is describes the features of Design Under Verification (DUV) for this thesis work.
Chapter four provides information about practical work for verification of the design in different tests. The implementation details are shown for each test using a methodology, tool and language. Simulation of the design are shown in the end of each test.
Finally, in chapter five the conclusion and future work are discussed.

Chapter 2

Theory, verification language, methodologies and tools

This chapter will research and discuss the methods, languages and tools for advanced verification methodology for ABB Robotics. Field studies were conducted externally and internally ABB on Language, Methodology and Tool. It is needed to research the requirement to achieve practical performance of the advanced verification and validation.

2.1 Important elements

The important elements to achieve high quality of verification must be considered. They are very important to take any decision.
Methods, language and tools must be chosen in a way that they could have potential to be used in future projects. Methods, tools and languages that are globally used have advantages to be reuse in the future developments. Time of the verification, Cost of the tools, human resources and required skills are key elements of a successful verification.

2.2 Field study

Field study conducted externally and internally in the company.

In the External Study method, languages and tools for verification were studied. Webinar seminars and tutorial videos used to study verification methodologies.

Internal Study is to discover that what other groups work on verification. ABB Corporate Research and ABB Robotic China and other departments in ABB are target groups for questionnaire, meeting and cooperation.

2.3 Need of verification language

The purpose of verification is different from design. In the verification higher level languages with very advanced tools can be used but in the design, language and methods are strongly limited by synthesis tools. In the design process languages and methods must be understood by the synthesis tool but in verification there is no such limitation. For this reason some languages have been designed specifically to fulfil the requirements of the verification methods.

Design languages have been developed to satisfy design and some of them for verification too.

2.4 Verification languages

In this part different languages, methods and tools based on their features, maturity and design complexity is studied.
There are differences between learning a language and how to use it in a methodical way by verification tools. The following languages is studied to find their features and abilities:

- VHDL IEEE 1076 ™
- Verilog IEEE 1364 ™
- e IEEE 1647 ™
- Open Vera
- System Verilog IEEE 1800 ™
- SystemC
- PSL (property Specific language)

2.4.1 VHDL IEEE 1076 ™

VHDL stands for VHSIC Hardware Description Language. VHSIC is abbreviation of Very High Speed Integrated Circuit. VHDL describes the behavioural and structure of digital circuit design. It is standardized as IEEE 1076. VHDL is used for both simulation and synthesis. [1]
Using VHDL has many advantages. It is a synthesizable RTL language that is chosen by many design teams all over the world. VHDL emphasis is in the compile time checking. It is a portable language between design tools and members in the project groups. Technology independent design i.e. functionality separated from implementation. VHDL has a large support by many design groups and automation tool vendors. [2]

2.4.2 Verilog IEEE 1364™

Verilog standardized as IEEE 1364, is one of the most famous languages used for design and verification of digital circuits at RTL level since 1980s.
There is broad support for Verilog by Electronic Design Automation (EDA) vendors. Verilog is the basis of the most of the Hardware Description Languages (HDL). [3]
Verilog defined the Programing Language Interface (PLI). PLI is a collection of routines which give authority to bidirectional interface for Verilog and other languages such as C. [4]

2.4.3 e-language IEEE 164™

The e language is an IEEE 164 used only for verification and it must be used together with a design language such as VHDL or Verilog for design and verification. It can verify SystemC™/C++, RTL model or gate-level model. One

feature of e-language is that it is very flexible language. It is known as advanced verification language for its Assertion and coverage. IEEE 164 supports random and constrained-random stimulus generation.
The e Reuse Methodology (eRM) is the basis for the Open Verification Methodology (OVM) and Universal Verification Methodology (UVM) [5]
The disadvantage of this language is that it is not widely used and just few tools such as Specman and Cadence can support it. [6]

2.4.4 Open Vera

Open Vera (TM) is an interoperable, open hardware verification language. This language is the basis of the advanced verification methods in the System Verilog. Open Vera is a widely developed verification language. OpenVera is an easy to learn language for new verification engineers. It combines the similarity and advantages of HDLs, C++ and Java, with purpose to functional verification. For this reason it is an appropriate language for using in testbenches, assertions and properties. Open Vera is appropriate language for verification of complex System on chips. It is widely used and supported by many vendor companies.
Similar to e language, Open Vera also has a disadvantage, it is used for only verification and most be used together with a design language such as VHDL, Verilog. [7]

2.4.5 System Verilog IEEE 1800™ -2012

System Verilog is rooted in the Verilog language. It is merged of Std 1364™ -2005 Verilog and Std 1800™ -2005 System Verilog. The last version is updated to Std 1800™ -2012.
The productivity boost of system Verilog is in both design and verification. This language covers design, simulation, validation and Assertion Based Verification (ABV).
System Verilog has object-oriented constructs. The powerful features such as random constrained stimuli generation, assertion and functional coverage makes system Verilog one of the most popular languages wide-world. But it has less emphasis on compile time checking in compare with VHDL language. [2]
It supports Open verification methodology (OVM) and Universal verification methodology (UVM) that are advanced methodology for verification. System Verilog has a great Multi-vendor support. There are new abstractions to adapt Cadence (eRM) to System Verilog. [8]

2.4.6 SystemC IEEE 1666-2005 ™

SystemC is approved by IEEE standard IEEE 1666-2005 ™. It is a set of C++ classes and macros. It provides an event driven simulation interface in C++. Comparing with other HDL languages, systemC is a higher level language. SystemC can be synthesized by tools and supports constraint randomization. It is ideal for transaction-level modeling (TLM). It has high-performance reference modelling.
One of the advantages of using SystemC is that it needs only a C++ compiler to run. It is flexible with platforms and licenses. [9] [10]

2.4.7 Property Specific Language (PSL) IEEE 1850-2010™

PSL stand for Property Specification Language developed by Accelera. It is a language based on properties to verify a designs. Properties are used to create assertions. It is used with design written in VHDL or Verilog. PSL can be embedded within the VHDL or Verilog code as comments or written in a single file. To prove or refuse a PSL routine hold on a design, a formal verification tool must be used. Model checking is a common used formal verification tool for this purpose.
PSL used for monitoring for testing the state of design under verification dynamically. PSL deliver legal sequence of inputs as constraint for verification of the design. It has functional coverage methodology for verification too. [11] [12]

2.5 Methodologies

Using methodologies provides possibilities with high level of confidence to achieve the best quality of a design and verification in a specific time and resources. (Reference) [13]

To reach to this goal a study on following methods is performed:

- Assertion Based Verification (ABV)
- Random Constraint
- Functional coverage
- Open source VHDL verification Methodology (OSVVM)
- Universal Verification Methodology (UVM)

2.5.1 Assertion Based Verification (ABV)

Assertion based verification is one of the powerful method for increasing observability and decreasing the debugging time. In the normal action an "info" and in failure action, an "error" will be reported.

By assertion based verification methodology a design intent is captured.
Assertions used in simulation, formal verification or emulation verifies the captured design intent to be implemented appropriately.
Assertions continuously check the state of a signal. If the state of signal is true, there is no action as result and monitoring will be continued until the state of signal change to a false state. When assertion detect the false state it executes an Error, Fatal or a warning. [14]
Using assertion based verification gives some advantages for verification. It increases observability. Observability helps for decreasing debugging time

during verification. Bugs can be find by assertions in significantly shorter time. [15]
Assertions are supported by PSL, VHDL and System Verilog language. System Verilog Assertions is an independent language for itself.
There are two type assertions, immediate assertions and concurrent assertions for checking signals.

Immediate Assertion: Immediate assertion is a single assert with a name that if the state of signal is false at that moment it will be monitored. Immediate assertion can be deactivated by the user. The difference of this kind of assertion with an "if" is that immediate assertion can be added to the RTL code and it is auto-ignored during synthesizing but "if" cannot be ignored by tools. [16]

Concurrent Assertion: Concurrent assertion do the complex checks through single or multi cycle assertion. It can check single or more signals in the same or different time.

In the property it can be a single condition or implication which has two conditions.

The property has different outcomes. The outcomes of the concurrent assertion in normal action is an "aborted", "failed" or "matched". For failure action the outcome is "fatal", "error", "warning" or "info". There are some example in following.

Concurrent assertion for implication of signal "a" and "b" for multi cycle:

```
    assert property ( a |-> ##1 b) …
```

Possible outcomes in normal action:

```
    aborted, failed, matched
```

Possible outcomes in failure Action:

```
    assert property (…) else    $fatal / $error / $warning / $info("text") ;
```

Sampled function: There are some functions for sampling in the assert property. They provide different methods to sample a signal. For instance signal a can be sampled by rose(a), fell(a), stable(a), past(a) and past(a,2).

Rose function checks if signal a changes from low to high. Fell sample from high to low. Stable(a) samples if signal a keep its previous value. Past(a) samples the previous value of signal a. Past(a,2) samples the value of signal a two cycle ago. [16]

Sequenced: Sampling can be performed by a sequence of signals in multi cycle assertion. The example below shows a sequenced.

```
assert property (a ##1 !a ##1 a |-> ##1 b ##1 !b)
```

Stand-alone sequenced: Sequences can be defined in separated stand-alone sequences with a name. Later in the assertion the state of the each stand-alone sequence can be sampled. [16]

2.5.2 Random Constraint Generation

This is one of the powerful methodology for random constraint stimuli generation to find any bugs. Random constraint generation targets every corner of the design under verification.

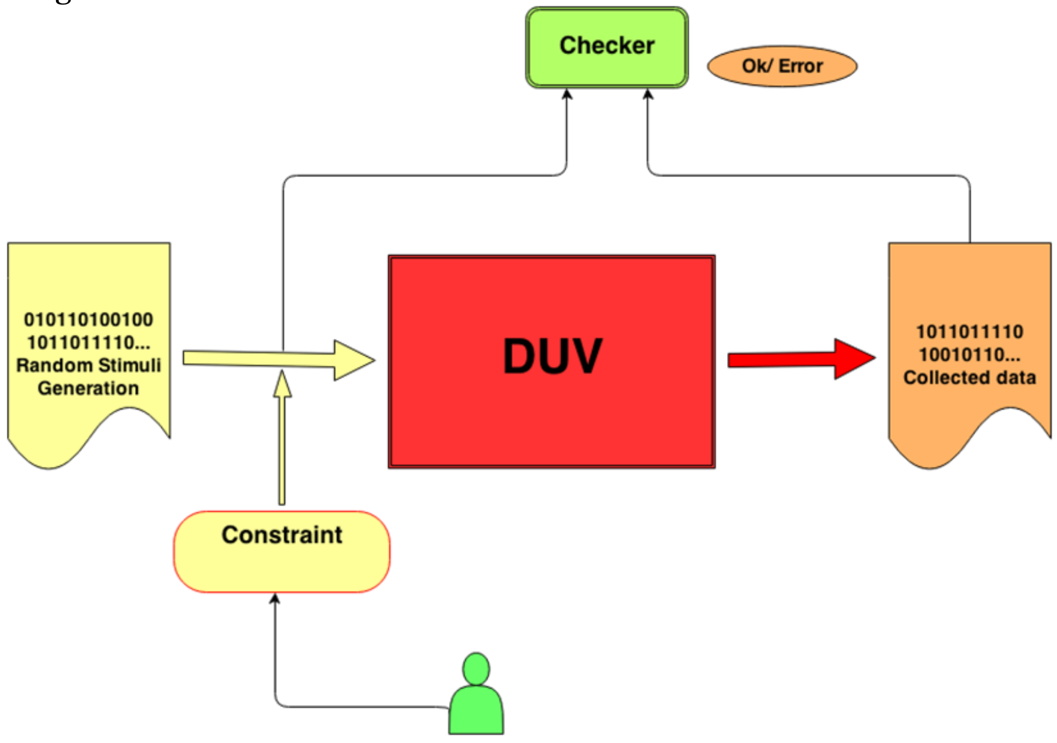


Figure 1 Random Constraint Generation

Random stimuli generation: Engineers are able to find bugs from random points of the design. It decreases human faults during verification. In system Verilog objects are defined in the classes. Outside the classes a randomization function is called to take the random value.

Constraint: By applying design constraint to the randomization and comparing the expected data with the collected data by a checker, potential bugs can be detected easily. There are some constraint methods that are applicable to the random stimulation by the user. The user defines the range of the randomization to match with design under test. The main constraint in System Verilog are, Inside, Distribution, Implication and Solving-order.
In the implementation part test 2 in chapter 4, constraints are described in more details. [16]

2.5.3 Functional coverage

Functional coverage is a methodology that proves that all the identified features in a design are verified. By using this methodology there is no requirement to re-check waveforms every time in the simulation. Functional coverage checks if every corner is tested. If there is some point that are not tested the verification is continued by changing constraint to reach the 100 percent of coverage. Functional coverage in VHDL and System Verilog are very similar. Main part of this methodology in system Verilog are listed below. [13]

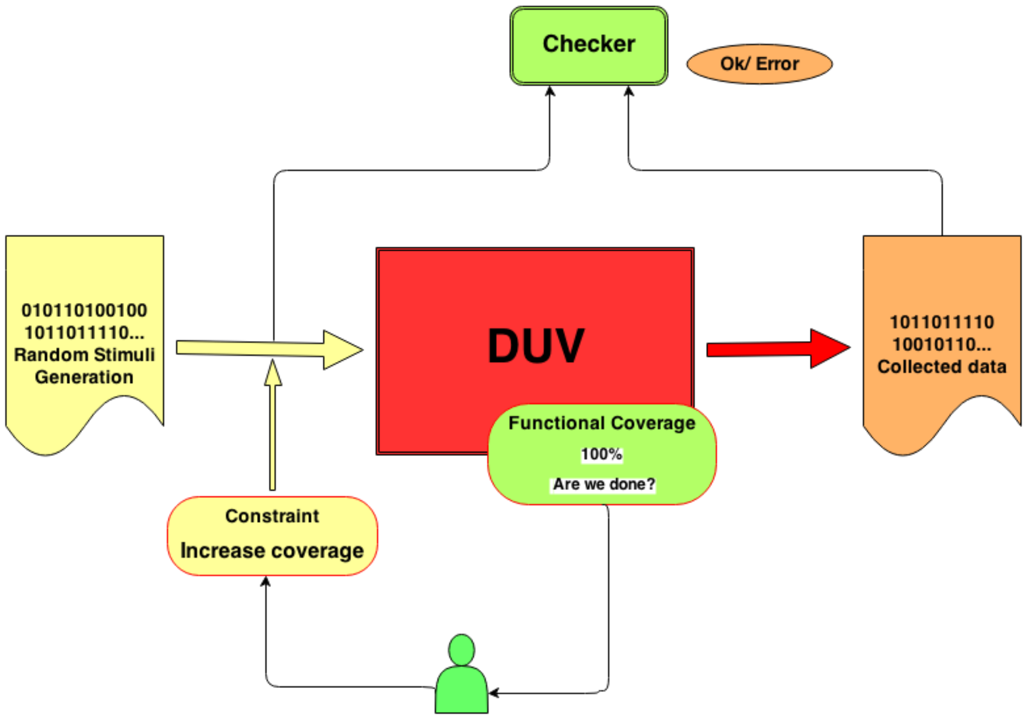


Figure 2 Functional coverage

Coverage group: A covergroup encapsulates the specification of a coverage model.
Coverpoints: Each random variable can have a coverpoint with different bins. Coverpoint can be an integer variable or an expression.
Coverbins: There are many different coverbins. User defined, Transition, Transition repetition, Automatic, Wildcard (sampled value X or Z), Ignored (excluded from coverage) and illegal (excluded from coverage and Error) are bins that can take values to be counted for functional coverage.
Cross coverage: Cross coverage is used for crossing between different coverpoints.
Sampling: Sampling is conducted in a task, function or procedure. [17] [18]

2.5.4 Open Source VHDL Verification Methodology (OSVVM)

Open Source VHDL Verification Methodology is called by its abbreviation OSVVM. Engineers may find more bugs due to unexpected combinations of inputs. This methodology do the same function as System Verilog. Since VHDL is a popular language for designers, it has been developed to be able to do more advanced verification.
OSVVM has developed VHDL libraries with constraint random generation and functional converge.
OSVVM has libraries for the popular tools such as Questasim. The main advantage of this methodology is that companies and designers can save more cost and resource. Using the same language used in the designs that are written in VHDL, make this methodology more popular.
For using this methodology it is required to install libraries in the tool and write the verification components in the testbench. The components are listed in following.

- Declaration of variables of type
- Stimuli process
- Constraints and distributions
- Coverpoint, bins and coverage process
- ICover and sampling process
- Write-bins and cover report process [19] [20] [21]

2.5.5 Universal Verification Methodology

The Universal Verification Methodology represents the latest advanced techniques to the verification. It provide testbenches for Verilog, System Verilog, VHDL and SystemC designs. It follows Accelleras standard and is open source under Apache license.
Universal Verification Methodology provides a higher quality and standard approach to building System Verilog testbenches.
UVM is developed based on the Open Verification Methodology (OVM) and e Reuse Methodology (eRM). UVM can be run in any System Verilog simulator. It is required to add UVM libraries to the tool. The main advantage UVM is reuse ability of the testbench and verification IP components. Skill reuse is also a key feature of UVM.

Universal Verification Methodology uses modular architecture to enable the reuse of components. There are two groups of components. The first group is environment and another one is agent. Any sub-components are located under environment or agent.
Tests can be called by user to be run for specific purpose. Every test can build its own components group automatically.
In the implementation part in chapter 4 there are two tests representing UVM in basic and advanced level. [22] [23]

2.6 Tools

Every tool supports some methodologies and languages. Based on methodology and language for design an appropriate tool must be chosen.
There are several tools supported by companies. Every tool has its own advantages and disadvantages. Some of them will be study and based on the company rolls and cost, appropriate tools are used for practical representation.

2.6.1 Modelsim

Modelsim is a popular tool that has been used by many companies. Modelsim is the common used tool for design and traditional verification designed by Mentor Graphic Company.
Modelsim is an HDL simulation tool. It has many libraries and Support Code coverage techniques. Recently it support open source VHDL verification methodology (OSVVM). [24] [25]

2.6.2 Questasim

It is an advanced verification tool given by Mentor Graphic. The environment is very similar to Modelsim. For this reason users of Modelsim do not need any effort to learn feature of Questasim. Questasim supports many methodologies such as Random Constraint, Functional coverage, Code coverage, Assertion Based Verification (ABV), Open Source VHDL Verification Methodology (OSVVM), Open Verification Methodology (OVM) and Universal Verification Methodology (UVM). [26]

2.6.3 Incisive Enterprise Verifier

Incisive Enterprise Verifier is an advance verification tool given by Cadence. It supports Random Constraint, Functional coverage, Code coverage, Assertion Based Verification (ABV), Open Source VHDL Verification Methodology (OSVVM), Open Verification Methodology (OVM) and Universal Verification Methodology (UVM). [27]

2.6.4 Synopsis VCS

Synopsys VCS® is a powerful tool provided by Synopsys. It is a tool for design, test, verification, coverage and debugging. It provide broad System Verilog support, verification planning, coverage and debug environment.
VCS has support for many design and verification languages such as Verilog, VHDL, System Verilog, Open Vera, and SystemC. It support latest methodologies such as assertions, functional coverage, OVM, and UVM methodologies. [28]

2.6.5 Riviera-pro

Riviera-pro is an advanced verification tool given by ALDEC Company. ALDEC Riviera-pro provide simulation in VHDL, Verilog, System Verilog, SystemC,

and mixed-language simulations. It has advanced debugging for multi-language debug environment.
ALDEC Riviera-pro provide support Random Constraint, code-functional coverage, Assertion-Based Verification (SVA and PSL)
, OSVVM, OVM and UVM. [29]

2.6.6 Incisive Specman Elite

Cadence® Incisive Specman Elite is a verification tool for e language. It can perform an automate testbench generation. Simulation of e testbenches with designs written in VHDL or Verilog, it require an HDL-simulation such as Synopsys VCS or Mentor Questa to be run with Specman in the same time. Specman provide support for e Reuse Methodology (eRM), Open Verification Methodology (OVM) and Universal Verification Methodology (UVM). [30]

2.7 Comparison of languages

There are some trade-offs for industry that plays a big role for choosing appropriate tools and methodologies. For big companies long term achievement is more intelligent than a short term reaching by using low cost tools and technology. But still cost and time are considered as important factors in verification development. In verification technology there are not a unique language, tool and method that can be used for multi-purpose and requirement. Table 1 shows the compression of languages in different aspect. VHDL, System Verilog are the languages that can fulfil the requirement for FPGA verification. They are developed to be advanced verification language used widely. There are great vendor support for both. They both have functional verification and assertion and can be used in both design and verification of FPGA.

Verification features	Verilog	VHDL	PSL	Vera	e	System verilog	SystemC
Advanced verification	☒	☑	☑	☑	☑	☑	☑
Widely used	☑	☑	☒	☒	☒	☑	☒
Multi-vendor support	☑	☑	☒	☒	☒	☑	☑
Functional coverage	☒	☑	☑	☑	☑	☑	☑
Assertion	☑	☑	☑	☑	☑	☑	☑
Using for both design and verification	☑	☑	☒	☒	☒	☑	☑
Using for FPGA verification	☑	☑	☒	☒	☒	☑	☒
Can be used as main language in UVM	☒	☒	☒	☒	☑	☑	☑

Table 1 Comparison of languages

Chapter 3

Design Under Verification

This chapter focuses on the Design Under Verification (DUV) and explain the details and behaviour of the DUV and each specific part of it. The DUV that this thesis verifies is an FPGA called R15.

3.1 The FPGA design R15

R15 is a communication and timing controller unit. It is a component of the in the servo computer that is a part of ABB IRC5 robot controllers.

R15 consist of two main components, R10 sub-system and PPC Local bus.

- R10 sub-system: It handles measurement loops, monitor for safety purpose and communication board status.
- PPC Local bus interface: provide communication link to the master unit PPC.

The thesis focuses on functional verification and testing of R15 register set and PPC local bus interface. For doing this it is required to know more about communication ports and type of registers and how access to every bit of registers.

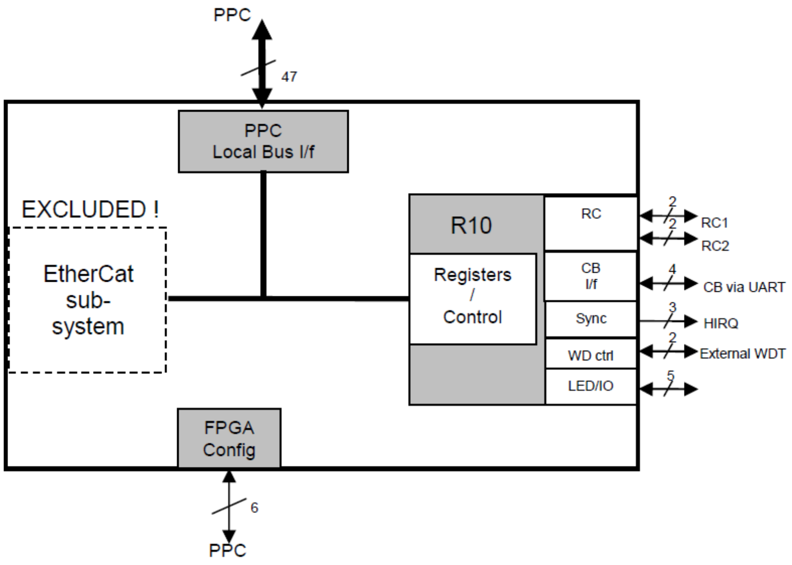


Figure 3 R15 Design under verification

3.2 Local bus interface ports

R15 is connected to the Power PC through the Local bus interface. The features of the ports of local bus interface are described below:

- Address (32 bits): Module has 32 bits address port. Bit 0 and bit 1 are for byte access that are permanently hard coded to zero and byte accesses cannot be executed on Local Bus. In addition, ADDRESS bit4-bit2 are used for burst access for maximum 8 burst accesses. In every single access it is allowed to access word address. But byte accesses cannot be performed in the local bus. However, to make use of max burst length, 32 bits addresses is obliged. LAD is 32 bit that is used for sending and receiving both data and address. There is a LATCH that connects the address [26:5] to the LAD. It has an enable signal named LALE.

- Data-in and Data-out (32 bits): Data in and out use 32 bits data. They are connected to bi-directional LAD in PPC. Since LAD is bi-directional, I-Buffer and O-Buffer control the connection between LAD and DATA-OUT or DATA-IN. LBCTL is a control signal to enable buffers. When LBTCL is 0, I-BUFFER connect the LAD to DATA_OUT and when it is 1, O-BUFFER connect the LAD to DATA-IN.

- R/nW port is used for commanding to the CPU to read or Write.

- nCS[1:0]: this port is used for selecting the

- Clock and controlling signals

- Not used signals: LUPWAIT is a dedicated wait signal which is not used in R15. LGPL0 is a configurable signal and it is also unused.

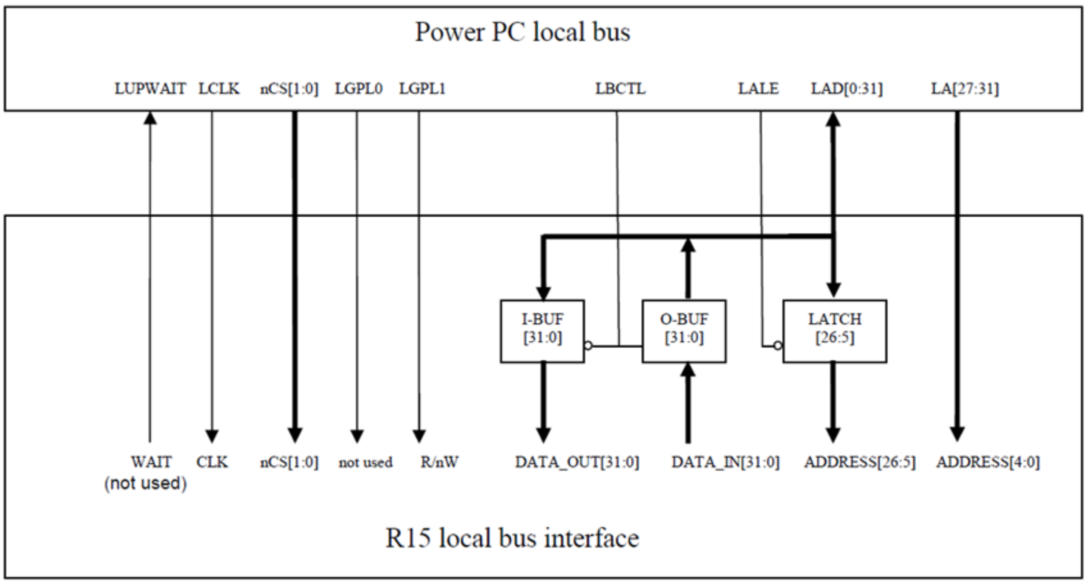


Figure 4 Communication ports for R15 local bus interface

Register bits in the R15 are accessible in different way but some of register bits are not fully accessible. They are described in below.

- R/W register: The data can be read and written normally. First step of the verification is to verify the reset values in the registers. Normal write and read data is performed in the next step in a verification.
- R/- bit register: Write process cannot be apply to this kind of registers. They can be read at every time. The reset value also is predefined.

- -/W register: this kind of registers can be written but there is no access to read. Since they are not read accessible, there is no way to test their value by PPC Local Bus.

For testing the registers it must be considered that some of the bit addresses are not implemented in the design or not used. In every case they have special condition.

- Implemented: Can take 0 or 1
- Not implemented: Not accessible and their value consider to be 0
- Not used: Implemented and can take 0 or 1 but not used in the design

Chapter 4

Implementation

A traditional testbench inherited by the FPGA designer that consist of some separate individual test with several packages.
There is a main testbench calling tb_server and several test_cases for specific purposes. Testbenches and packages for developing in next tests are described below.

- **Test_case_1:** Testbench for testing PPC local bus
- **Tb-server**: Handling of communication between testcases, surrounding components and DUV.
- **Test_case_package**: Time measurement, control and printing out to result file.
- **Test_func_package**: Different functions to handle data, sample CB data, compare actual
- **Server_control_package**: Defining types and functions for use in test cases server functions.
- **Local_bus_rw**: Functions to read and write local bus.
- **Data_log**: Functions for test I/O. printing the result.log.

4.0 Test 0: Simplifying traditional testbenches

The traditional test has been used for simplifying test_case_1 and including other necessary packages inside it. It needs to minimize the code for developing in the next test with new methodology. In this test demonstrate code coverage without adding any new methodology. For that reason it named test_0.

Excluding unnecessary parts will reduce the developing time to focus on the functional verification and new methodologies.

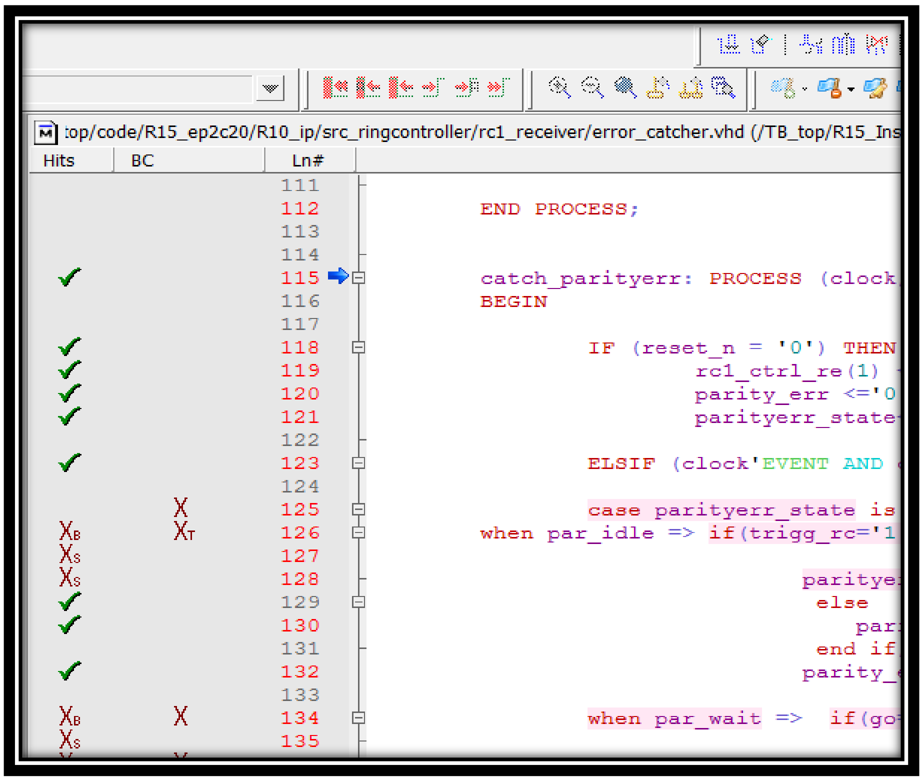
4.0.1 Code coverage

Code Coverage is a tool feature. It shows that if all of the code has been tested
In Mentor Modelsim an X in the Hits column indicates:

- Missed Statement is shown as XS
- Missed Branch is shown as XB
- Missed Condition is shown as XC

An X in the BC column indicates:

- Missed true is shown as XT
- Missed False is shown as X



```
top/code/R15_ep2c20/R10_ip/src_ringcontroller/rc1_receiver/error_catcher.vhd (/TB_top/R15_Ins
Hits    BC      Ln#
                111
                112          END PROCESS;
                113
                114
✓               115          catch_parityerr: PROCESS (clock
                116          BEGIN
                117
✓               118                  IF (reset_n = '0') THEN
✓               119                          rc1_ctrl_re(1)
✓               120                          parity_err <='0
✓               121                          parityerr_state
                122
✓               123                  ELSIF (clock'EVENT AND
                124
        X       125                  case parityerr_state is
XB      XT      126          when par_idle => if(trigg_rc='1
XS              127
XS              128                                    parityer
✓               129                                     else
✓               130                                        par
                131                                     end if
✓               132                                    parity_
                133
XB      X       134                  when par_wait =>  if(go
XS              135
```

Figure 5 Code coverage in Modelsim

4.1 Test 1: Open Source VHDL Verification Methodology (OSVVM)

OSVVM methodology will find more possible bugs in the design due to unexpected combination of data and address. By testing all corner of design with any possible data, there is no chance for bugs to be hidden in the design. This can be done by using constraint random generation method and 100% functional coverage of the design.

4.1.1 Need of verification tool

Since modelsim cannot support constraint random or other advanced features of OSVVM Cadence and Questa can fulfill the requirement of this test. A trial version of Questasim has been used for this test.

4.1.2 OSVVM Libraries in the Questasim

Last version of Questasim has an empty folder for OSVVM files. The OSVVM files must be compiled into OSVVM library. The files are consist of following packages. [19]

```
SortListPkg_int.vhd
RandomBasePkg.vhd
RandomPkg.vhd
CoveragePkg.vhd
MessagePkg.all
```

For using OSVVM methodology in the code and make the appropriate packages visible, library and packages must be added.

```
library osvvm;
use osvvm.sortlistPkg_int.all;
use osvvm.RandomPkg.all;
use osvvm.CoveragePkg.all;
use osvvm.randombasePkg.all;
use osvvm.messagePkg.all;
```

4.1.3 Declaration of variables of type

For OSVVM randomization and functional coverage it is needed to declare variables of different type that have been defined in OSVVM library.
RandomPType is a protected type declared in the RandomPkg to represent random value. Rnd_a and Rnd_b are RandomPType variables.

If they are declared in the architecture outside the process, they must be declared as shared variable.

```
shared variable Rnd_a : RandomPType;
shared variable Rnd_d : RandomPType;
```

For collecting the data for functional coverage to be measured it uses covType. It is a coverpoint type declared in the coveragePkg to collect data values into bins.

```
shared variable cp_data : covPType;
shared variable cp_addr : covPType;

shared variable cp_addr_data_1 : covPType;
shared variable cp_addr_data_2 : covPType;
```

4.1.4 Stimuli process

In stimuli process, constraint random generation will be used to generate the random data and addresses that is written to the registers.

The written data in the addresses will be read and compare to the golden model which is the original random data. If the data mismatch an error will arose.

By calling functions or procedures that is part of the protected type in OSVVM Library the constraint random values is generated.

```
Rnd_d.InitSeed (Rnd_d'instance_name);
```

This function uses a string to set an initial seed to produce initial values in constraint random generation. The seed will created when the attribute 'instance name used for returning a string.

4.1.5 Constraints and Distributions

```
data_rand <= Rnd_d.Randslv ( min_dat, max_dat, 32);                    (1)

data_rand <= Rnd_d.Randslv (X”00000000”, X”FFFFFFFF”, 32);             (2)

addr_rand <= Rnd_a.Randslv ( (add_1,
                              add_2,
                              add_3,
                              add_4,
                              add_5,
                              add_6,
                              add_7,
                              add_8,
                              add_9,
                              add_10,
                              add_11,
                              add_12,
                              add_13,
                              add_14,
                              add_15,
                              add_16,
                              add_17,
                              add_18,
                              add_19,
                              add_20,
                              add_21), 32);                              (3)

data_rand <= Rnd_d.Randslv( X”00000000”, X”FFFFFFFF”,
     (X”00000001”, X”00000002”, X”00000003”,…) ,32);                    (4)
```

(1) It creates a random std_logic vector with length of 32 by using the minimum and maximum range of data. Min_dat and max_dat were declared already as std_logic vector in the declaration part. Min_dat is has the value of X”00000000” and Max_dat is X”FFFFFFFF”.
(2) It creates a random std_logic vector with length of 32 by using the direct values of minimum and maximum range of data.
(3) It creates a random std_logic vector with length of 32. Distribution of the random generator will be according the variables that contain the address of each specific value.
(4) It creates a random std_logic vector with length 32 by using the direct values of minimum and maximum range of data and inside the prentices the illegal

values are excluded from the randomization. Since illegal addresses are more than the legal addresses, this way is not an appropriate solution.

4.1.6 Coverpoint, bins and coverage process

Using functional coverage, it is known which specific points have not been covered or which parts of the design are covered. It helps to cover all the corner of the design to be verified.
The data items that are measuring are known as coverpoints. Coverpoints have some feature to collect data into bins to be measured.

Coverpoint of data, consisting of 16 bins for range of data values produced.

```
cp_data.AddBins(GenBin(min_dat,max_dat,16));
```

Coverpoint of different addresses, each specific address has 1 bin.

```
cp_addr.AddBins(GenBin(add_1,add_1));
```

Coverpoint for every combination of Address and data, 16 bins.

```
cp_addr_data_1.AddCross(GenBin(add_1,add_1), GenBin(min_dat, max_dat,16));
```

4.1.7 ICover and sample process

A process is required for sampling of the data. Coverage is sampled in the sample process. When the sampling function is executed, values are scanned and the bins are updated.

```
cp_addr.ICover (TO_INTEGER(UNSIGNED(addr_rand)));
cp_data.ICover (TO_INTEGER(UNSIGNED(data_rand)));

cp_addr_data_1.ICover(TO_INTEGER(UNSIGNED(addr_rand)),TO_INTEGER(UNSI
GNED(data_rand))) );
```

4.1.8 Write bins and cover report process

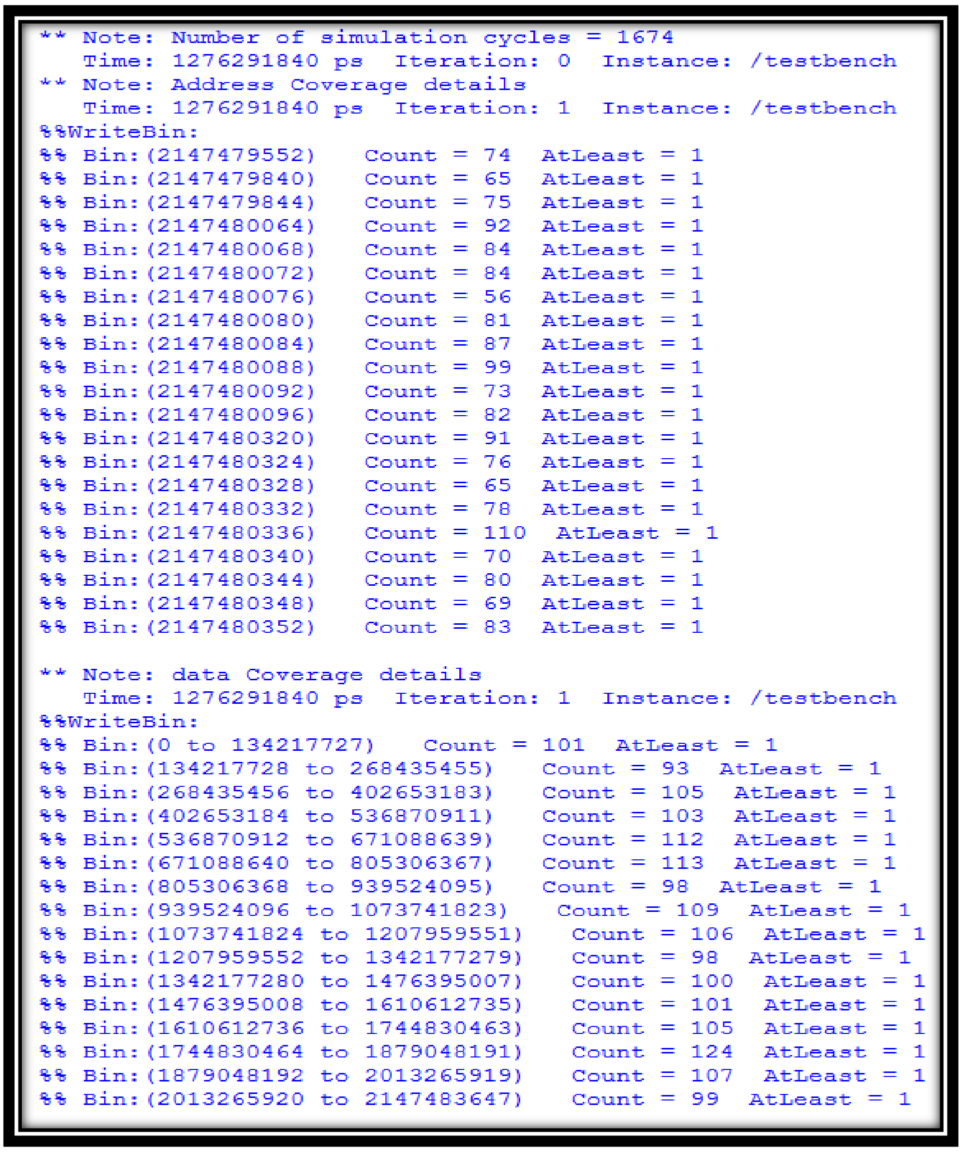
When simulation is terminated at the end of the program, the coverage report will print out all the data and address and cross coverage of both.

```
cp_addr.WriteBin;
cp_data.WriteBin;
cp_addr_data_1.writebin;
report "coverage holes” & to_string(cp_addr_data_1.CountCovHoles);
```

4.1.9 Simulation and Coverage report for address and data

At the end of simulation a report is printed to the console. It contains information about the number of simulation cycles, time of simulation, iteration and coverage detail for address and data.

Figure 6 OSVVM functional coverageFigure 6 shows more details about functional coverage. Address range is from xFFFFF000 to xFFFFF230 which target specific registers and data range is from x00000000 to xFFFFFFFF.

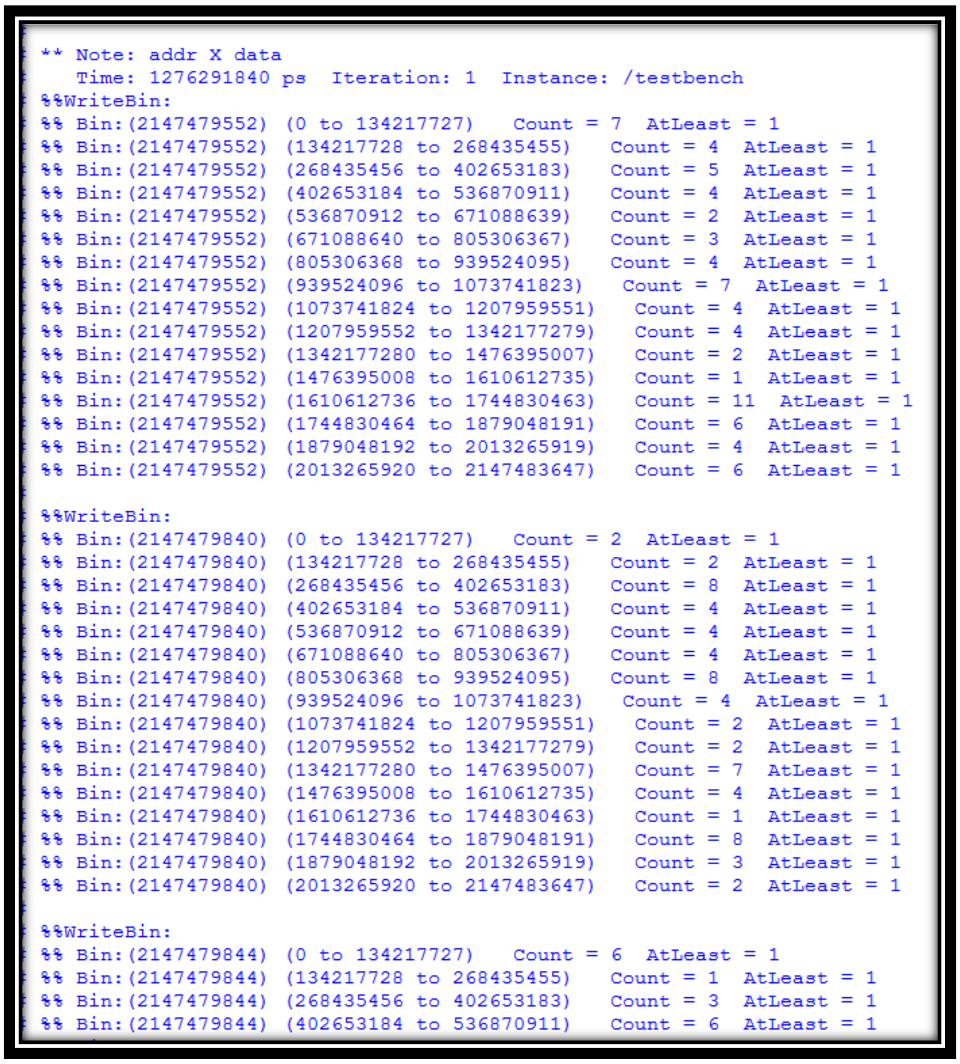


```
** Note: Number of simulation cycles = 1674
   Time: 1276291840 ps  Iteration: 0  Instance: /testbench
** Note: Address Coverage details
   Time: 1276291840 ps  Iteration: 1  Instance: /testbench
%%WriteBin:
%% Bin:(2147479552)   Count = 74  AtLeast  = 1
%% Bin:(2147479840)   Count = 65  AtLeast  = 1
%% Bin:(2147479844)   Count = 75  AtLeast  = 1
%% Bin:(2147480064)   Count = 92  AtLeast  = 1
%% Bin:(2147480068)   Count = 84  AtLeast  = 1
%% Bin:(2147480072)   Count = 84  AtLeast  = 1
%% Bin:(2147480076)   Count = 56  AtLeast  = 1
%% Bin:(2147480080)   Count = 81  AtLeast  = 1
%% Bin:(2147480084)   Count = 87  AtLeast  = 1
%% Bin:(2147480088)   Count = 99  AtLeast  = 1
%% Bin:(2147480092)   Count = 73  AtLeast  = 1
%% Bin:(2147480096)   Count = 82  AtLeast  = 1
%% Bin:(2147480320)   Count = 91  AtLeast  = 1
%% Bin:(2147480324)   Count = 76  AtLeast  = 1
%% Bin:(2147480328)   Count = 65  AtLeast  = 1
%% Bin:(2147480332)   Count = 78  AtLeast  = 1
%% Bin:(2147480336)   Count = 110  AtLeast  = 1
%% Bin:(2147480340)   Count = 70  AtLeast  = 1
%% Bin:(2147480344)   Count = 80  AtLeast  = 1
%% Bin:(2147480348)   Count = 69  AtLeast  = 1
%% Bin:(2147480352)   Count = 83  AtLeast  = 1

** Note: data Coverage details
   Time: 1276291840 ps  Iteration: 1  Instance: /testbench
%%WriteBin:
%% Bin:(0 to 134217727)    Count = 101  AtLeast = 1
%% Bin:(134217728 to 268435455)   Count = 93  AtLeast = 1
%% Bin:(268435456 to 402653183)   Count = 105  AtLeast = 1
%% Bin:(402653184 to 536870911)   Count = 103  AtLeast = 1
%% Bin:(536870912 to 671088639)   Count = 112  AtLeast = 1
%% Bin:(671088640 to 805306367)   Count = 113  AtLeast = 1
%% Bin:(805306368 to 939524095)   Count = 98  AtLeast = 1
%% Bin:(939524096 to 1073741823)   Count = 109  AtLeast = 1
%% Bin:(1073741824 to 1207959551)   Count = 106  AtLeast = 1
%% Bin:(1207959552 to 1342177279)   Count = 98  AtLeast = 1
%% Bin:(1342177280 to 1476395007)   Count = 100  AtLeast = 1
%% Bin:(1476395008 to 1610612735)   Count = 101  AtLeast = 1
%% Bin:(1610612736 to 1744830463)   Count = 105  AtLeast = 1
%% Bin:(1744830464 to 1879048191)   Count = 124  AtLeast = 1
%% Bin:(1879048192 to 2013265919)   Count = 107  AtLeast = 1
%% Bin:(2013265920 to 2147483647)   Count = 99  AtLeast = 1
```

Figure 6 OSVVM functional coverage

Figure 7 shows details of cross coverage report for some register with their Address and Data range from x00000000 to x0FFFFFFF.



```
** Note: addr X data
   Time: 1276291840 ps  Iteration: 1  Instance: /testbench
%%WriteBin:
%% Bin:(2147479552) (0 to 134217727)    Count = 7  AtLeast = 1
%% Bin:(2147479552) (134217728 to 268435455)   Count = 4  AtLeast = 1
%% Bin:(2147479552) (268435456 to 402653183)   Count = 5  AtLeast = 1
%% Bin:(2147479552) (402653184 to 536870911)   Count = 4  AtLeast = 1
%% Bin:(2147479552) (536870912 to 671088639)   Count = 2  AtLeast = 1
%% Bin:(2147479552) (671088640 to 805306367)   Count = 3  AtLeast = 1
%% Bin:(2147479552) (805306368 to 939524095)   Count = 4  AtLeast = 1
%% Bin:(2147479552) (939524096 to 1073741823)   Count = 7  AtLeast = 1
%% Bin:(2147479552) (1073741824 to 1207959551)   Count = 4  AtLeast = 1
%% Bin:(2147479552) (1207959552 to 1342177279)   Count = 4  AtLeast = 1
%% Bin:(2147479552) (1342177280 to 1476395007)   Count = 2  AtLeast = 1
%% Bin:(2147479552) (1476395008 to 1610612735)   Count = 1  AtLeast = 1
%% Bin:(2147479552) (1610612736 to 1744830463)   Count = 11  AtLeast = 1
%% Bin:(2147479552) (1744830464 to 1879048191)   Count = 6  AtLeast = 1
%% Bin:(2147479552) (1879048192 to 2013265919)   Count = 4  AtLeast = 1
%% Bin:(2147479552) (2013265920 to 2147483647)   Count = 6  AtLeast = 1

%%WriteBin:
%% Bin:(2147479840) (0 to 134217727)    Count = 2  AtLeast = 1
%% Bin:(2147479840) (134217728 to 268435455)   Count = 2  AtLeast = 1
%% Bin:(2147479840) (268435456 to 402653183)   Count = 8  AtLeast = 1
%% Bin:(2147479840) (402653184 to 536870911)   Count = 4  AtLeast = 1
%% Bin:(2147479840) (536870912 to 671088639)   Count = 4  AtLeast = 1
%% Bin:(2147479840) (671088640 to 805306367)   Count = 4  AtLeast = 1
%% Bin:(2147479840) (805306368 to 939524095)   Count = 8  AtLeast = 1
%% Bin:(2147479840) (939524096 to 1073741823)   Count = 4  AtLeast = 1
%% Bin:(2147479840) (1073741824 to 1207959551)   Count = 2  AtLeast = 1
%% Bin:(2147479840) (1207959552 to 1342177279)   Count = 2  AtLeast = 1
%% Bin:(2147479840) (1342177280 to 1476395007)   Count = 7  AtLeast = 1
%% Bin:(2147479840) (1476395008 to 1610612735)   Count = 4  AtLeast = 1
%% Bin:(2147479840) (1610612736 to 1744830463)   Count = 1  AtLeast = 1
%% Bin:(2147479840) (1744830464 to 1879048191)   Count = 8  AtLeast = 1
%% Bin:(2147479840) (1879048192 to 2013265919)   Count = 3  AtLeast = 1
%% Bin:(2147479840) (2013265920 to 2147483647)   Count = 2  AtLeast = 1

%%WriteBin:
%% Bin:(2147479844) (0 to 134217727)    Count = 6  AtLeast = 1
%% Bin:(2147479844) (134217728 to 268435455)   Count = 1  AtLeast = 1
%% Bin:(2147479844) (268435456 to 402653183)   Count = 3  AtLeast = 1
%% Bin:(2147479844) (402653184 to 536870911)   Count = 6  AtLeast = 1
```

Figure 7 OSVVM functional cross coverage

4.2 Test_2: Functional verification with System Verilog and Questasim

This test is done in System Verilog and Questasim. System Verilog is a mature language that is made for verification. It is an object oriented language and easy to learn.

4.2.1 Need of verification tool

Since Modelsim cannot support constraint random or other advanced feature of System Verilog Cadence and Questa can fulfill the requirement of this test. A trial version of Questasim has been used for this test too.

4.2.2 Test structure

The test is performed by writing a top testbench and a test program. In the top testbench there is a clock generator that is providing the clock to other parts. An interface has been used for instantiation of the Design Under Verification (DUV). Connecting signals also declared in the top testbench.

The main part of the code is written inside the test case in a program. The test program contains the all other component in the test case.

The default clock uses the positive edge of the clock signal provided by clock generator to drive the ports. The default skew is set to 100 ps.

Reset signal work as active low. It is low in the time of start and will be high after 10 clock cycle.

There is a packet generator that provide the randomization of a class with constraints.

Program starting with testing of the initial or reset values in the registers and will check if there is mismatch or possible errors. A function saves the reset values in a queue.

There are driver tasks for communicating with the FPGA to read and write the local bus.
The data in the registers is checked for any possible errors by checkers.

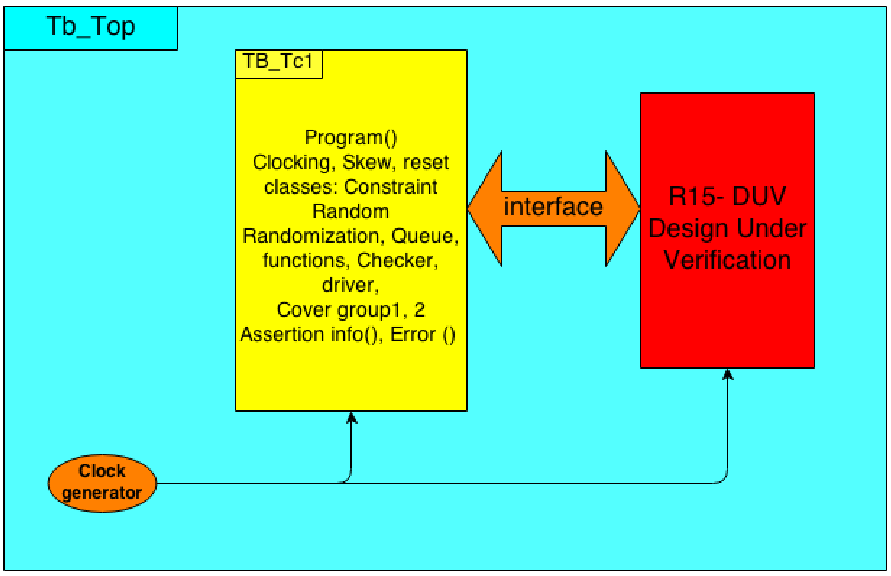


Figure 8 Testbench structure for Functional verification with System Verilog

4.2.3 Constraints random generation

Constraint random generation is used to randomize the data values to random addresses. This methodology uses a packet class to constraint the legal values. There are two 32 bit variables for address and data. They are declared by rand keyword inside the class. There are different keywords that has been used for constraint the randomization that is described in following.

- **Inside**: By this keyword a legal value among the given values to each specific address can be determined. The specific address here is read_reg
  Constraint read_reg {addr_rand inside { 'hFFFFF000,'hFFFFF10C…};}

- **Distribution**: When weight based is required in System Verilog, distribution expression is useful. Below the dist keyword return true when the expression is contained. The second address will be generated two time more than the firs address.
  Constraint addr_dist {addr_rand dist {'hFFFFF000 := 1, 'hFFFFF10C := 2 };};

- **Implication**: The implication keyword always has two conditions. If first condition is true then next condition must be true. In the following example it shows that when the randomized address is 'hFFFFF120 which is the address of LEGAL_IO register then the legal range of randomized data must be in a range that bit 0 to 15 and bit 27 to 31 must be zero. Other bits can take 0 or 1.
  Constraint legal_IO { addr_rand=='hFFFFF120 -> data_rand [15:0]==0 && data_rand [31:27]==0;}

- **Solving Order**: it will assure that all combination of the legal randomization has been uniformed by given constraint. Constraint randomization for address will be done with deactivating the constraint for data. Address will be fixed value then the constraint for data will be turn on and randomized.

```
Constraint order_1  {solve addr_rand before data_rand;}
```

By randomization of a class one of the all possible combination will be selected randomly and all variables must satisfying all constraints.

4.2.4 Functional Coverage

There are two covergroup for functional coverage that are described below.

4.2.4.1 Covergroup for address, data, crossing and repeating

In this covergroup there are four coverpoints.

- The first coverpoint is for address. `a:  coverpoint addr_rand`
  Every address has a coverbin. For example the bin for test register is written as: `a01_TEST= {8'hFFFFF000};`
- Coverpoints for data: an automatic bin can be associated to the coverpoint of the data using `d:  coverpoint data_rand.` But since it is required to check every single bit and excluding some of the unused bits or not implemented bits in some of the registers, a separated coverpoint for every single bit is instantiated. For example for bit zero of data a coverpoint is defined: `data0:  coverpoint data_rand[0];`
- Crossing data and address: for each coverpoint of data and address a cross coverpoint is defined and ignore bits are listed. For example IO and WDT are two registers that their third bit (bit [2]) is not implemented. By defining the ignore bit, the coverage will exclude the statistics of that bit. If Ignore bit is not defined, the coverage will include the “not implemented bits” in the functional coverage and it will never reach the 100 % coverage.

  ```
  regXdata2: cross data2, address
  {
  ignore_bins regXdata2= binsof (data2)&&( binsof (address.IO)|| binsof
  (address.WDT));
  }
  ```

- Repeatation of address: this coverpoint is used to gather statistics if all the registers have been tested two times continuously.
  `reg_repeat: coverpoint addr_rand`
  Bins for every address will be defined in this way:

  ```
  bins        TEST_2          = ('hFFFFF000[* 2]);
  ```

4.2.4.2 Covergroup for Local bus, transition and crossing

- Coverpoint with transition bins for communication ports: This coverpoint there are communication ports that are using for coverage. For example when LALE is zero and in the next positive edge of clock, it changes to one and remain at one for at least one clock and maximum 100 clock cycle and take zero again.

```
 lale:   coverpoint lale
    {
    bins lale_up1           = ( 0 => 1[*1:100] => 0);
     }
```

- Cross signals for communication ports: crossing of communication ports are defined in this type of coverpoint. There is an example below.

```
nCS_0xlbctl: cross nCS_0, lbctl;
```

4.2.5 System Verilog Assertions (SVA)

System Verilog assertion is one of the powerful method for increasing observability and decreasing the debugging time. In this test the normal action an “info” and in failure action, an “error” will be reported.

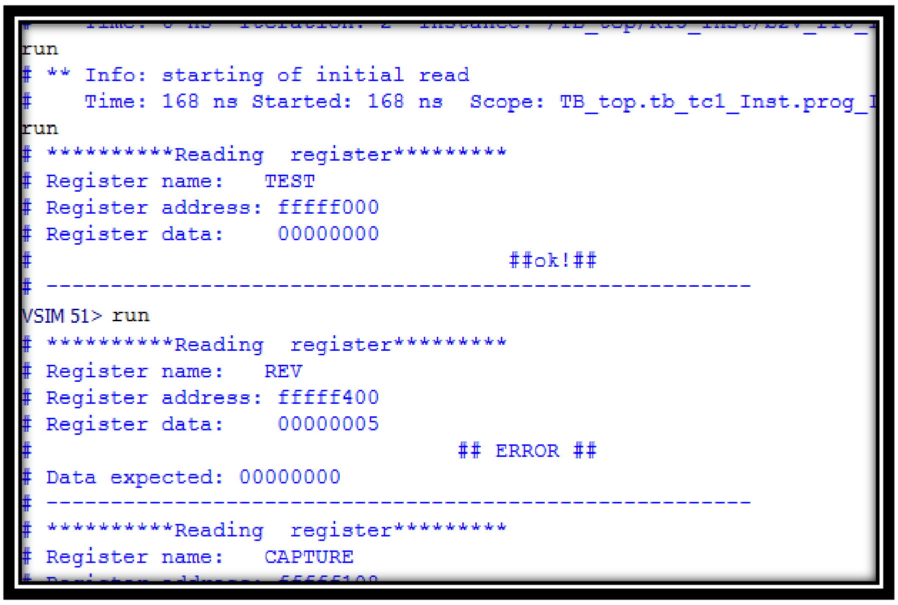
- **Info:** This action is used for showing the start of executing specific part of the code such as reading the reset values, writing random data and reading random data. For this part an immediate assertion has been used.

```
prog_read_start: assert property(@(negedge lclk) $rose(read_start) |->1 )
$info("starting reading r data ");
```

- **Error:** Error actions have been written to monitor the communication ports for several sequences during communication with FPGA. It observes the ports and will detect any potential failures and generate an error. Concurrent assertion will have a complex check across multi cycle assertion and using sequenced assertion.

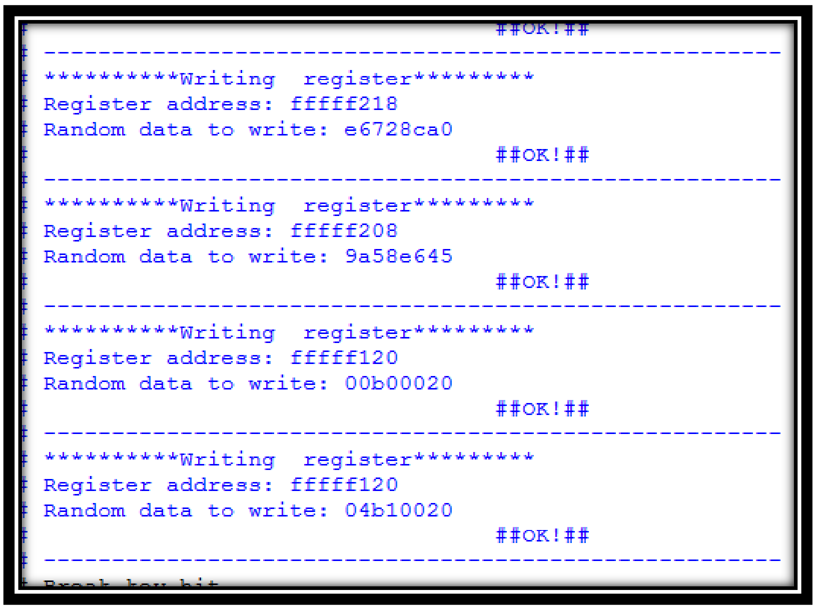
```
Sequence addrlatch;
    ((reset) ##1 $rose(lale) ##1 lale ##1 !$fell(lale));
Endsequence

REG_data_direction_for_write : assert property(@(negedge lclk) addrlatch and lbctl |-
> $fell(lgpl)) else $error("Data direction and address latch for Writing ");
```



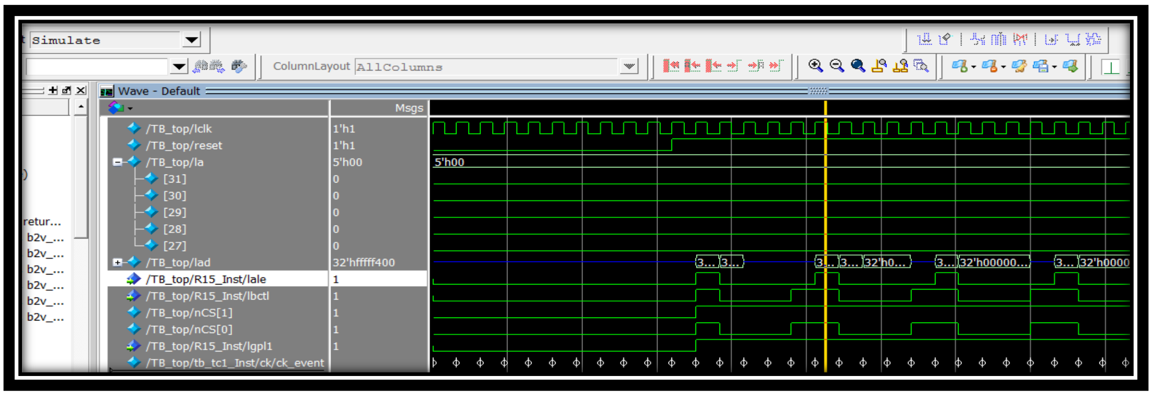
```
#    Time: 0 ns  Iteration: 2  Instance: /TB_top/RTS_Inst/b2v_ifo_I
run
# ** Info: starting of initial read
#    Time: 168 ns Started: 168 ns  Scope: TB_top.tb_tc1_Inst.prog_I
run
# **********Reading  register*********
# Register name:    TEST
# Register address: ffffff000
# Register data:     00000000
#                                     ##ok!##
# -------------------------------------------------------
VSIM 51> run
# **********Reading  register*********
# Register name:    REV
# Register address: ffffff400
# Register data:     00000005
#                                 ## ERROR ##
# Data expected: 00000000
# -------------------------------------------------------
# **********Reading  register*********
# Register name:    CAPTURE
# Register address: ffffff100
```

Figure 9 Reading initial data and Assertion (info)



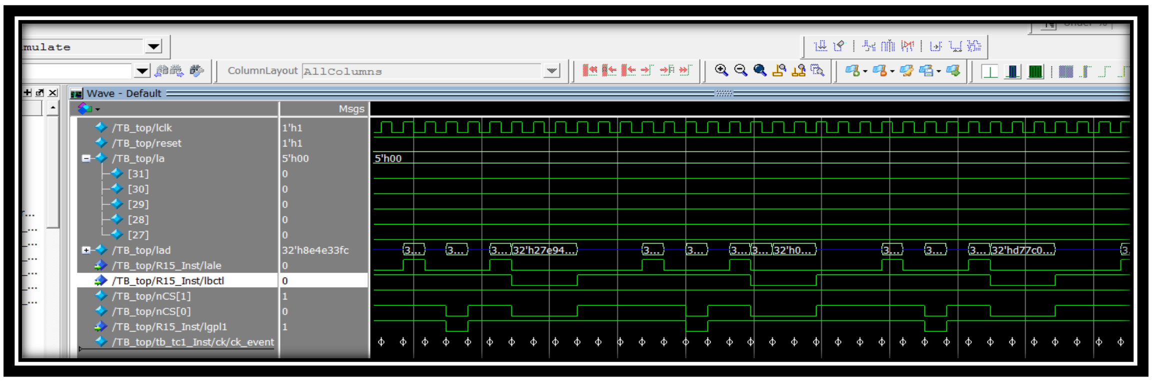
```
#                                   ##OK!##
# -------------------------------------------------------
# **********Writing  register*********
# Register address: ffffff218
# Random data to write: e6728ca0
#                                   ##OK!##
# -------------------------------------------------------
# **********Writing  register*********
# Register address: ffffff208
# Random data to write: 9a58e645
#                                   ##OK!##
# -------------------------------------------------------
# **********Writing  register*********
# Register address: ffffff120
# Random data to write: 00b00020
#                                   ##OK!##
# -------------------------------------------------------
# **********Writing  register*********
# Register address: ffffff120
# Random data to write: 04b10020
#                                   ##OK!##
# -------------------------------------------------------
# Break key hit
```

Figure 10 Constraint Random stimuli generation write random data in random address



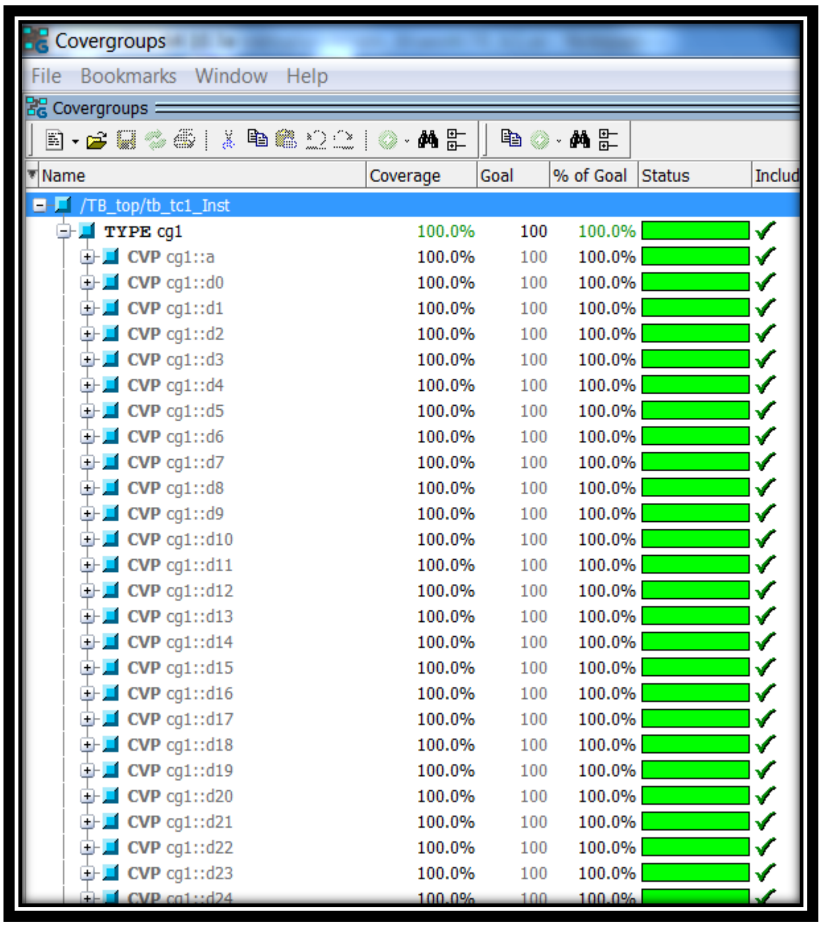
```
Wave - Default                              Msgs
/TB_top/lclk                                1'h1
/TB_top/reset                               1'h1
/TB_top/la                                  5'h00
  [31]                                      0
  [30]                                      0
  [29]                                      0
  [28]                                      0
  [27]                                      0
/TB_top/lad                                 32'hffffff400
/TB_top/R15_Inst/lale                       1
/TB_top/R15_Inst/lbctl                      1
/TB_top/nCS[1]                              1
/TB_top/nCS[0]                              1
/TB_top/R15_Inst/lgpl1                      1
/TB_top/tb_tc1_Inst/ck/ck_event
```

Figure 11 Local bus interface signals for initial read



```
Wave - Default                              Msgs
/TB_top/lclk                                1'h1
/TB_top/reset                               1'h1
/TB_top/la                                  5'h00
  [31]                                      0
  [30]                                      0
  [29]                                      0
  [28]                                      0
  [27]                                      0
/TB_top/lad                                 32'h8e4e33fc
/TB_top/R15_Inst/lale                       0
/TB_top/R15_Inst/lbctl                      0
/TB_top/nCS[1]                              1
/TB_top/nCS[0]                              0
/TB_top/R15_Inst/lgpl1                      1
/TB_top/tb_tc1_Inst/ck/ck_event
```

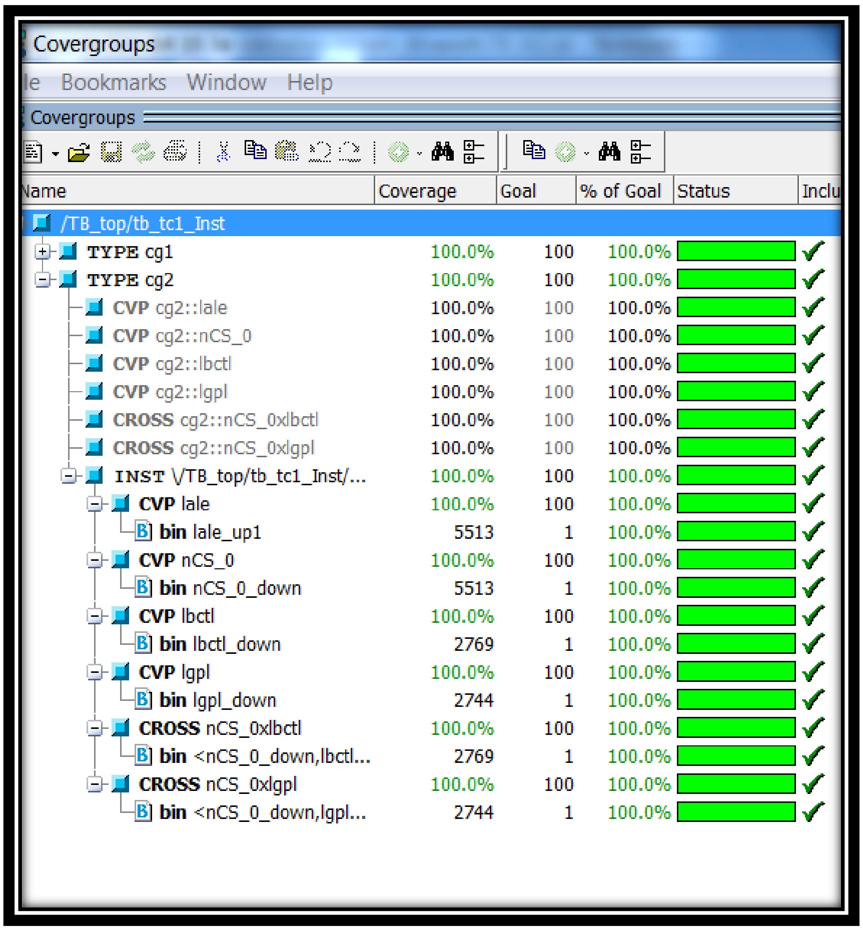
Figure 12 Local bus interface signals for random write and read



Covergroups

Name	Coverage	Goal	% of Goal	Status	Includ
/TB_top/tb_tc1_Inst					
TYPE cg1	100.0%	100	100.0%	🟩	✓
CVP cg1::a	100.0%	100	100.0%	🟩	✓
CVP cg1::d0	100.0%	100	100.0%	🟩	✓
CVP cg1::d1	100.0%	100	100.0%	🟩	✓
CVP cg1::d2	100.0%	100	100.0%	🟩	✓
CVP cg1::d3	100.0%	100	100.0%	🟩	✓
CVP cg1::d4	100.0%	100	100.0%	🟩	✓
CVP cg1::d5	100.0%	100	100.0%	🟩	✓
CVP cg1::d6	100.0%	100	100.0%	🟩	✓
CVP cg1::d7	100.0%	100	100.0%	🟩	✓
CVP cg1::d8	100.0%	100	100.0%	🟩	✓
CVP cg1::d9	100.0%	100	100.0%	🟩	✓
CVP cg1::d10	100.0%	100	100.0%	🟩	✓
CVP cg1::d11	100.0%	100	100.0%	🟩	✓
CVP cg1::d12	100.0%	100	100.0%	🟩	✓
CVP cg1::d13	100.0%	100	100.0%	🟩	✓
CVP cg1::d14	100.0%	100	100.0%	🟩	✓
CVP cg1::d15	100.0%	100	100.0%	🟩	✓
CVP cg1::d16	100.0%	100	100.0%	🟩	✓
CVP cg1::d17	100.0%	100	100.0%	🟩	✓
CVP cg1::d18	100.0%	100	100.0%	🟩	✓
CVP cg1::d19	100.0%	100	100.0%	🟩	✓
CVP cg1::d20	100.0%	100	100.0%	🟩	✓
CVP cg1::d21	100.0%	100	100.0%	🟩	✓
CVP cg1::d22	100.0%	100	100.0%	🟩	✓
CVP cg1::d23	100.0%	100	100.0%	🟩	✓
CVP cg1::d24	100.0%	100	100.0%	🟩	✓

Figure 13 Reaching 100% Functional coverage Covergroup-1 for radom address and data



Name	Coverage	Goal	% of Goal	Status	Inclu
/TB_top/tb_tc1_Inst					
TYPE cg1	100.0%	100	100.0%	🟩	✓
TYPE cg2	100.0%	100	100.0%	🟩	✓
CVP cg2::lale	100.0%	100	100.0%	🟩	✓
CVP cg2::nCS_0	100.0%	100	100.0%	🟩	✓
CVP cg2::lbctl	100.0%	100	100.0%	🟩	✓
CVP cg2::lgpl	100.0%	100	100.0%	🟩	✓
CROSS cg2::nCS_0xlbctl	100.0%	100	100.0%	🟩	✓
CROSS cg2::nCS_0xlgpl	100.0%	100	100.0%	🟩	✓
INST \/TB_top/tb_tc1_Inst/...	100.0%	100	100.0%	🟩	✓
CVP lale	100.0%	100	100.0%	🟩	✓
bin lale_up1	5513	1	100.0%	🟩	✓
CVP nCS_0	100.0%	100	100.0%	🟩	✓
bin nCS_0_down	5513	1	100.0%	🟩	✓
CVP lbctl	100.0%	100	100.0%	🟩	✓
bin lbctl_down	2769	1	100.0%	🟩	✓
CVP lgpl	100.0%	100	100.0%	🟩	✓
bin lgpl_down	2744	1	100.0%	🟩	✓
CROSS nCS_0xlbctl	100.0%	100	100.0%	🟩	✓
bin <nCS_0_down,lbctl...	2769	1	100.0%	🟩	✓
CROSS nCS_0xlgpl	100.0%	100	100.0%	🟩	✓
bin <nCS_0_down,lgpl...	2744	1	100.0%	🟩	✓

Figure 14 Reaching 100% functional coverage Covergroup-2 for local bus interface signals

4.3 Test 3 Universal verification methodology (UVM)

UVM uses constrained random and it is a coverage-driven verification. It has a configurable and flexible environment with testbenches. In UVM tests are separated from traditional module testbenches. Every testbench have an environment including some tests.

The UVM package has a class library that includes other classes.

- **uvm_components:** uvm_components are used to construct a class based hierarchical testbench structure.
- **uvm_objects:** uvm_objects are used as data structures for configuration of the testbenchs.
- **uvm_transactions:** uvm_transactions are used in stimulus generation and analysis.

4.3.1 Test 3 structure

UVM Architecture is written in System Verilog to verify the Design. R15 is the design that is called Design Under Test (DUT) in the UVM tests. The UVM has a run_test() method that can be called from initial block in the top level module.

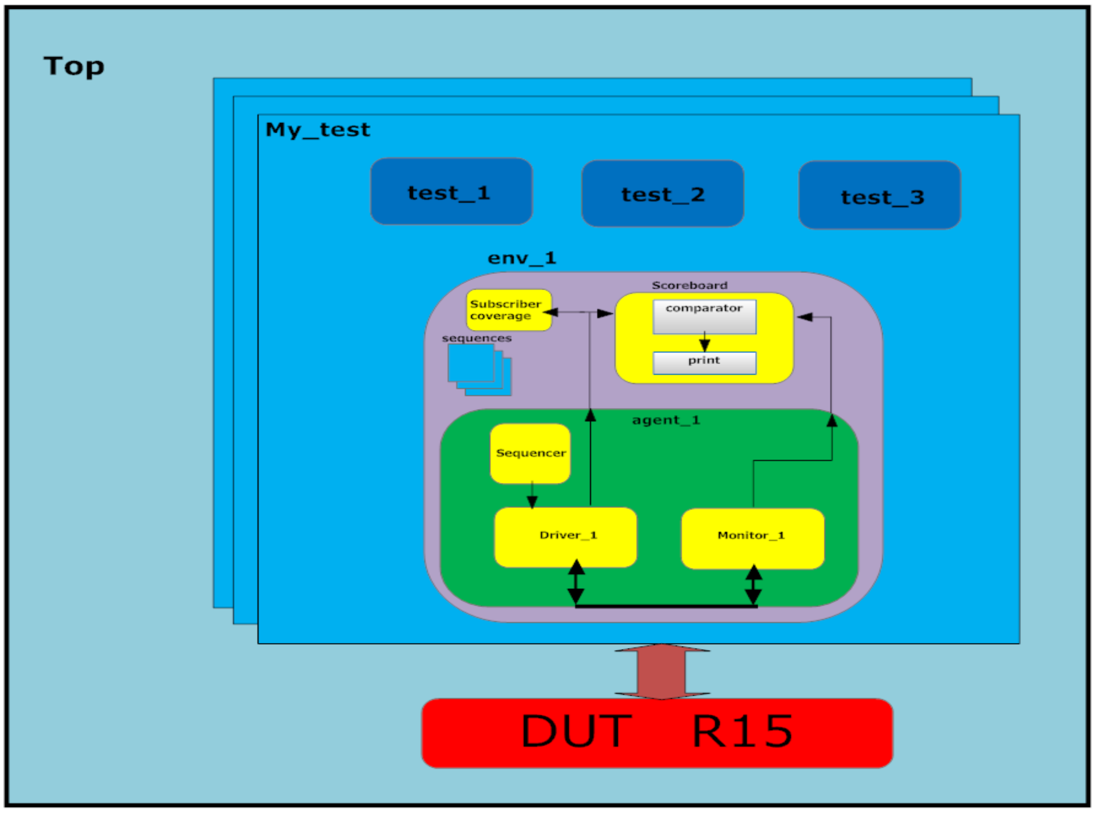


Figure 15 UVM Architecture

4.3.2 Top level module

UVM top level module contains the DUT, testbenches and connections. The run_test() method starts the execution of the UVM phases. The UVM phases controls the order to build the related testbench and generation of the stimulus and writes the report of the result of the simulation.

4.3.3 Interface

An interface connect the signals of DUT in the Top module to the testbench and other components.

4.3.4 DUT

As it is shown in the Figure 15, DUT must be placed in the top level module and connected to the interface.

4.3.5 Clocking

Clock generation produces a 66 MHZ clock in the top module and passes it to the clock signal of the DUV.

4.3.6 Testbench

A testbench is built from components that contains other components. The top level class in the testbench that is called test class configures the testbench, initiates building of the next level down in the hierarchy and initiate stimulus by starting the main sequence.

4.3.7 Test cases

Each test case is implemented by an extension of a test base class.
There are three tests that each of them extend my_test with different number of randomization sequences. For example test_1 with extention of the my_test randomize the data 5000 times and send it to the sequencer in the agent.

```
task run_phase(uvm_phase phase);
        packet_generator seq;
        seq = packet_generator::type_id::create("seq");
        assert( seq.randomize() );
        phase.raise_objection(this);
        seq.start(my_env_h.my_agent_h.my_sequencer_h);
        phase.drop_objection(this);
 endtask // run_phase
```

The test_2 provides a random number of the randomization between 1000 and 4000 times and test_3 uses the turn off mode of the number of randomization and give a random number between 10,000 and 48,000 times for the repetition.

4.3.8 Environment

The environment is the extension of the UVM_env. It is a collection of component that contain sub-components such as scoreboard, subscriber, and sequences. They are orientated around agent.

4.3.9 Agent

Most DUTs have a number of different signal interfaces, each of which can have their own protocol.
In the UVM agent, a collection of components focused around a specific pin-level interface are collected together. Monitor and driver use pin level transaction.
The purpose of the agent is to provide a verification component such as monitor, driver and sequencer which allows users to generate and monitor pin level transactions.
It uses analysis port for connecting driver and monitor to the scoreboard and subscriber in the connection phase. Component are built by factory technique in the building phase.

4.3.10 Sequences

There are two sequences to be used in the UVM tests, named packet_generator and packet_generator_random_number. The packet_generator use the my_transaction which is an extension of UVM_sequence_item to randomize a data value. The packet_generator_random_number do the randomization for a random time. They can be used in any tests to sending to the sequencer.

4.3.11 Subscriber

Subscriber is an extension of the UVM_subscriber using my_transaction for sampling and coverage. Similar to functional coverage in the test 3 it uses covergroups and coverpoints for address and data. There defined bins for address and automatic bins for data. Crossing of address and data is also used for coverage. Sampling is conducted at the end of a function that named scoreboard.

4.3.12 Scoreboard

Scoreboard consists of a comparator and a printer inside the subscriber. The subscriber has a function named scoreboard in the blocks that compares the expected data from my_transaction with data collected by monitor. It writes the result of the comparison as “OK” or “Error”.

4.3.13 Driver

It is an UVM_driver extention using my_transaction. It receives random data and address by seq_item_port.get() method from sequencer. It has an analysis_port to send the data to the scoreboard. There is a run task that the driver communicates with the design through the virtual interface.

4.3.14 Monitor

It is an UVM_monitor extention. Monitor has an analysis_port to send the collected data to the scoreboard. Similar to the driver it has a run task to collect data from design through virtual interface.

4.3.15 Sequencer

Sequencer is a typedef of UVM_sequencer that is built inside agent. Every test will use the sequencer to take the randomization values and sent into the driver. Tests use the raise_objection method before starting the sequence and after finishing the sequence use the drop_objection method.

```
phase.raise_objection(this);
seq.start(my_env_h.my_agent_h.my_sequencer_h);
phase.drop_objection(this);
```

Every time the task function in the driver uses seq_item_port.get(tx) method, the sequencer sends the randomized value from the sequence to the driver. [31]

4.4 Test 4 Complex UVM architecture

This test changes the architecture of the previous test. It uses two agents in every environment. For doing this it is required to define both agents in the environment and change the connections.
Agent_1 is used for sending data to the DUT and Agnet_2 reads the data.

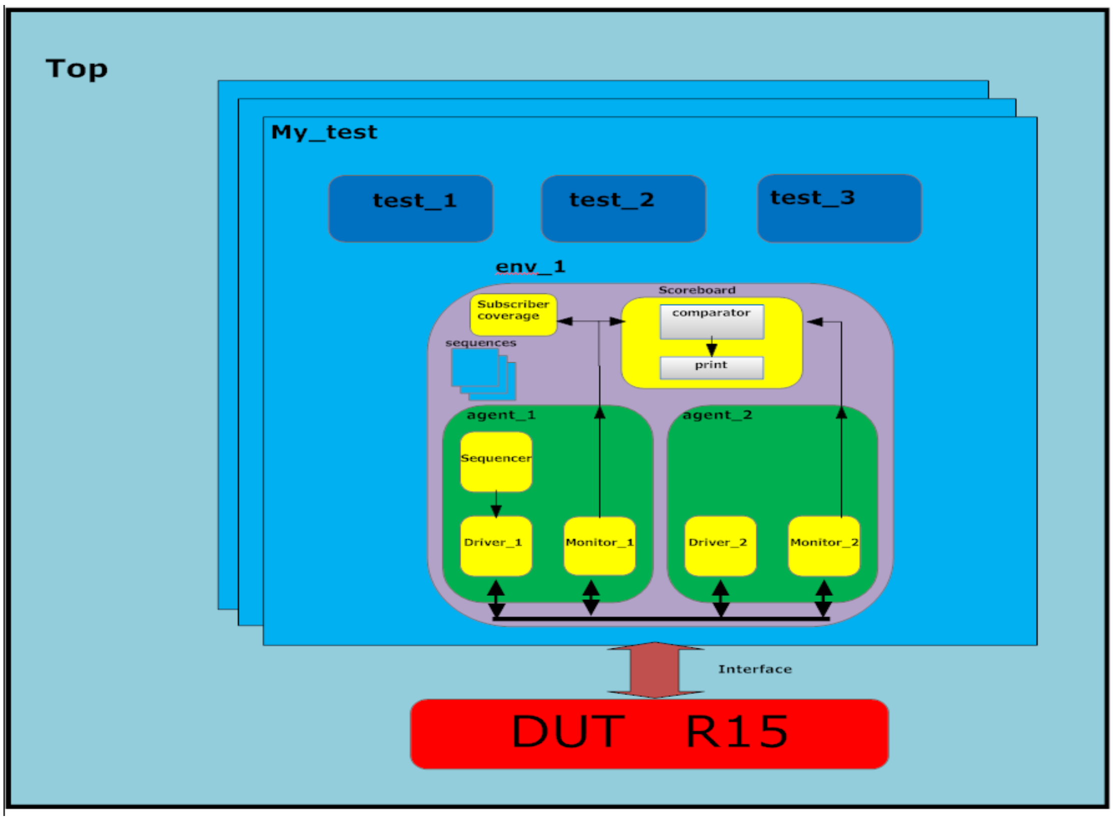


Figure 16 Complex UVM architecture

4.4.1 Agent 1

Agent_1 consits of sequencer, driver_1, monitor_1 and the uvm_analysis_port that send the data from monitor_1 to scoreboard. Monitor_1 reads the data that Driver_1 is sending to the DUT by Virtual interface.

4.4.2 Agent 2

Agent_2 consists of Driver_2 and Monitor_2 and an uvm_analysis_port to send the received data from DUT to the scoreboard. Driver_2 configures the communication ports of the Design by virtual interface and provide the address of the register that must be read. Monitor_2 takes the data value and send it to the scoreboard.

4.4.3 Full architecture

Figure 17 shows the entire architecture of the UVM test. An initial testbench named init_test contained the init_test_1 and init_env. Init_test_1 extends the initial testbench by reading the reset values of the registers and compares it with the expected reset values. It take the reset values and addresses stored in the initial sequence and sends it to the init_driver by sequencer in the init_agent.
Since initial_test does not use the randomization, functional coverage and subscriber are not required.

In the top module there is a run_test("test1") method that user can run every test through it.

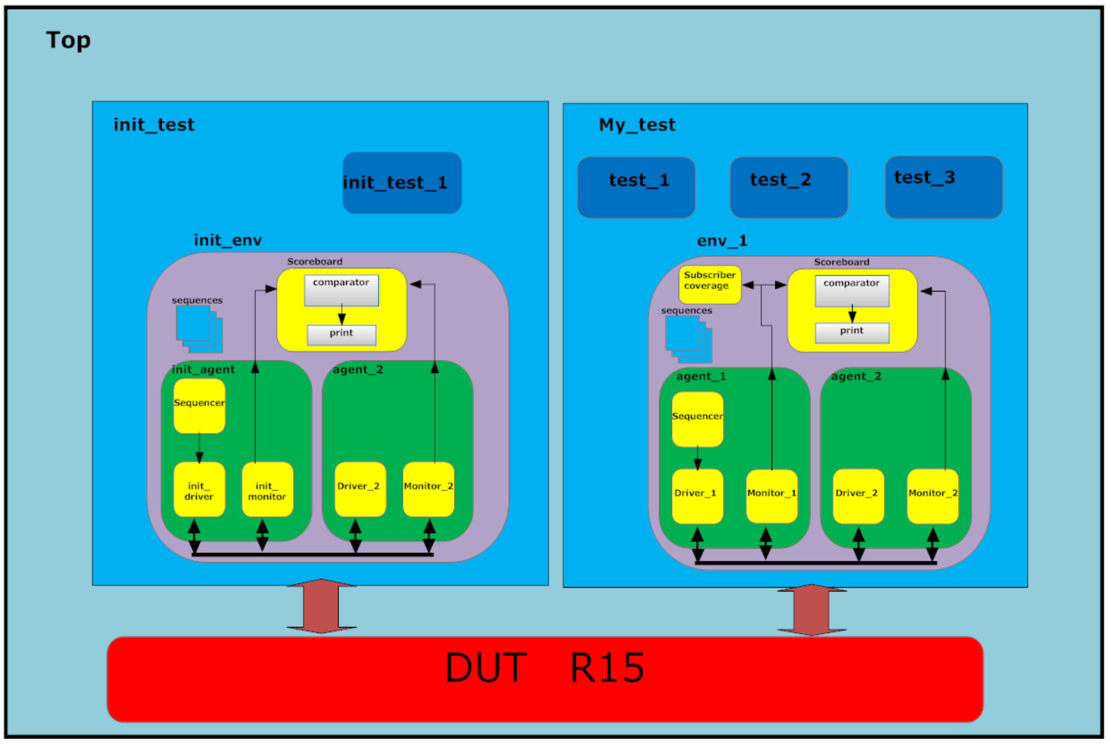
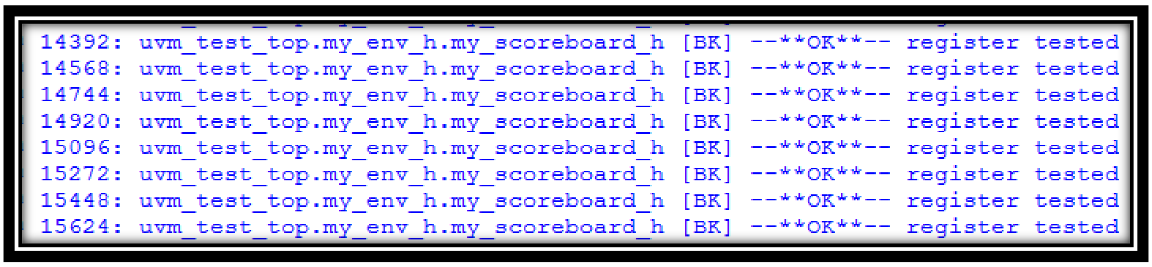


Figure 17 A full overview of architecture of the UVM tests

The test1 does 100 times randomization, test2 use a random number between 5000 and 10,000 times and test3 use the turn off mode and give a random number between 1500 and 3500 times.

4.4.4 Simulation

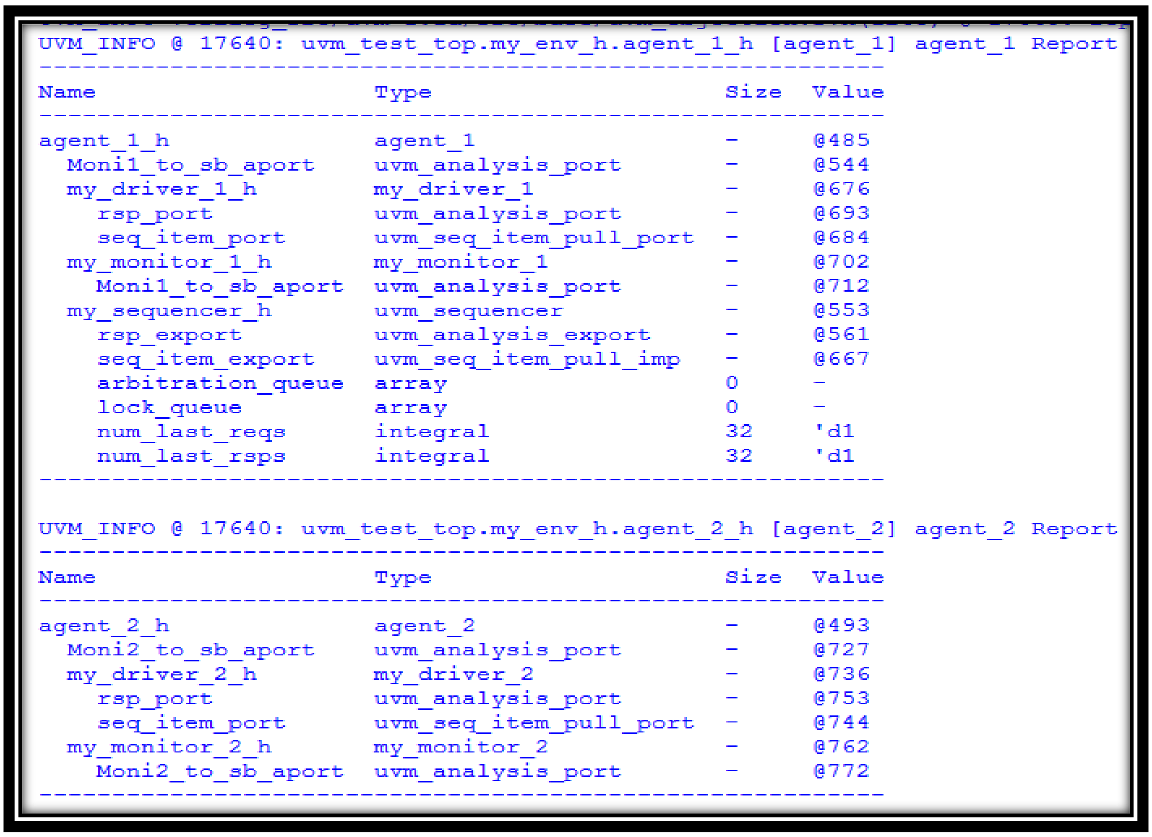
Figure 18 shows the test of registers printed by comparison function in the scoreboard



```
14392: uvm_test_top.my_env_h.my_scoreboard_h [BK] --**OK**-- register tested
14568: uvm_test_top.my_env_h.my_scoreboard_h [BK] --**OK**-- register tested
14744: uvm_test_top.my_env_h.my_scoreboard_h [BK] --**OK**-- register tested
14920: uvm_test_top.my_env_h.my_scoreboard_h [BK] --**OK**-- register tested
15096: uvm_test_top.my_env_h.my_scoreboard_h [BK] --**OK**-- register tested
15272: uvm_test_top.my_env_h.my_scoreboard_h [BK] --**OK**-- register tested
15448: uvm_test_top.my_env_h.my_scoreboard_h [BK] --**OK**-- register tested
15624: uvm_test_top.my_env_h.my_scoreboard_h [BK] --**OK**-- register tested
```

Figure 18 Test of registers by scoreboard function

Figure 19 shows the UVM Agents Report

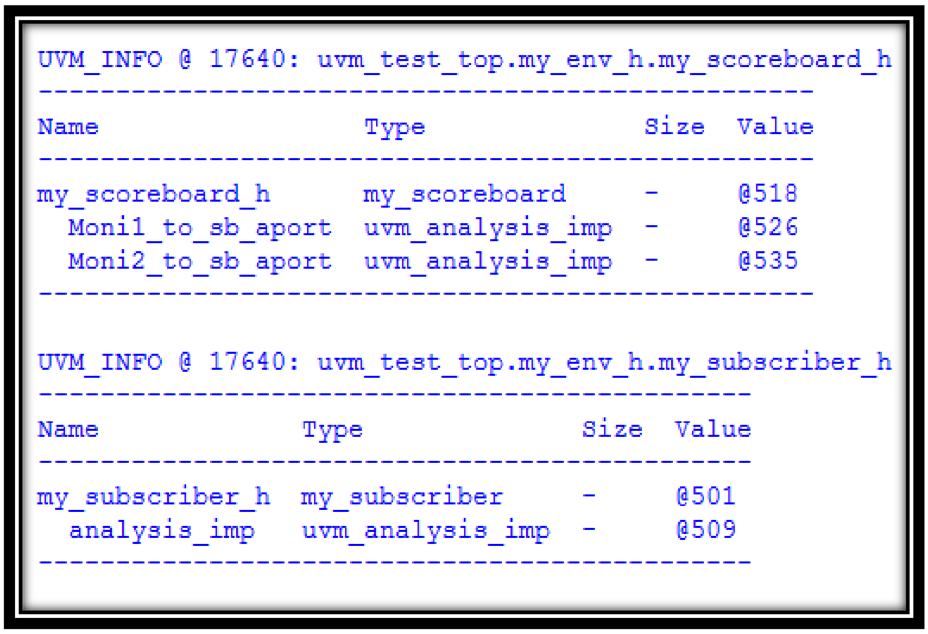


```
UVM_INFO @ 17640: uvm_test_top.my_env_h.agent_1_h [agent_1] agent_1 Report
----------------------------------------------------------
Name                     Type                     Size  Value
----------------------------------------------------------
agent_1_h                agent_1                  -     @485
  Moni1_to_sb_aport      uvm_analysis_port        -     @544
  my_driver_1_h          my_driver_1              -     @676
    rsp_port             uvm_analysis_port        -     @693
    seq_item_port        uvm_seq_item_pull_port   -     @684
  my_monitor_1_h         my_monitor_1             -     @702
    Moni1_to_sb_aport    uvm_analysis_port        -     @712
  my_sequencer_h         uvm_sequencer            -     @553
    rsp_export           uvm_analysis_export      -     @561
    seq_item_export      uvm_seq_item_pull_imp    -     @667
    arbitration_queue    array                    0     -
    lock_queue           array                    0     -
    num_last_reqs        integral                 32    'd1
    num_last_rsps        integral                 32    'd1
----------------------------------------------------------

UVM_INFO @ 17640: uvm_test_top.my_env_h.agent_2_h [agent_2] agent_2 Report
----------------------------------------------------------
Name                     Type                     Size  Value
----------------------------------------------------------
agent_2_h                agent_2                  -     @493
  Moni2_to_sb_aport      uvm_analysis_port        -     @727
  my_driver_2_h          my_driver_2              -     @736
    rsp_port             uvm_analysis_port        -     @753
    seq_item_port        uvm_seq_item_pull_port   -     @744
  my_monitor_2_h         my_monitor_2             -     @762
    Moni2_to_sb_aport    uvm_analysis_port        -     @772
----------------------------------------------------------
```

Figure 19 UVM Agents Report

Figure 20 shows the UVM Scoreboards Report

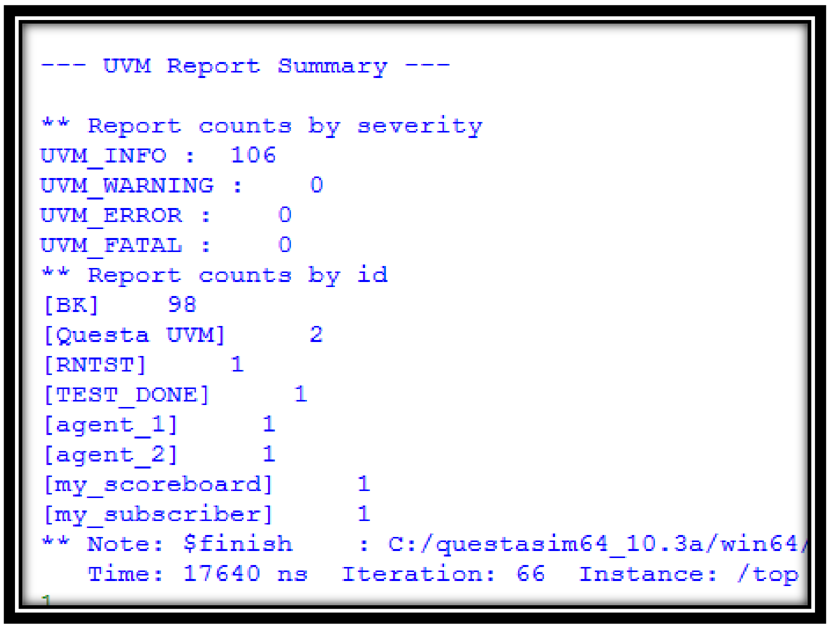


```
UVM_INFO @ 17640: uvm_test_top.my_env_h.my_scoreboard_h
--------------------------------------------------
Name                     Type               Size  Value
--------------------------------------------------
my_scoreboard_h          my_scoreboard      -     @518
  Moni1_to_sb_aport      uvm_analysis_imp   -     @526
  Moni2_to_sb_aport      uvm_analysis_imp   -     @535
--------------------------------------------------

UVM_INFO @ 17640: uvm_test_top.my_env_h.my_subscriber_h
----------------------------------------------
Name                 Type               Size  Value
----------------------------------------------
my_subscriber_h      my_subscriber      -     @501
  analysis_imp       uvm_analysis_imp   -     @509
----------------------------------------------
```

Figure 20 UVM Scoreboards Report

Figure 21 shows the UVM Report Summary



```
--- UVM Report Summary ---

** Report counts by severity
UVM_INFO :  106
UVM_WARNING :    0
UVM_ERROR :    0
UVM_FATAL :    0
** Report counts by id
[BK]    98
[Questa UVM]     2
[RNTST]     1
[TEST_DONE]     1
[agent_1]     1
[agent_2]     1
[my_scoreboard]     1
[my_subscriber]     1
** Note: $finish    : C:/questasim64_10.3a/win64/
   Time: 17640 ns  Iteration: 66  Instance: /top
```

Figure 21 UVM Report Summary

Chapter 5

Conclusion

5.1 Conclusions

In this thesis some methodologies and tools and languages are studied and four different testbenches with chosen methodology by appropriate tools and languages are conducted. Each test has its advantages and disadvantages based on the ABB Robotics verification team and current resources.

5.1.1 Traditional direct test with VHDL (Test_0)

- **Pros**: This test has a Simple architecture. It is low cost and can be conducted with basic requirement of language and tool for simple FPGA designs.
- **Cons**: It is difficult to find bugs in the design due to basic techniques. Verification engineer must target every corner manually by guessing the bugs and it needs a deep understanding of the design. This test is incompatible for complex design.

5.1.2 Open source VHDL verification Methodology (Test_1)

- **Pros**: It provides an advanced methodology that can save cost without need of learning new language for the VHDL teams. OSVVM uses functional coverage and Constraint Random stimulation to reach the full coverage of the design. It works with regular VHDL simulators such as Mentor's ModelSim and Aldec's Active-HDL without additional licenses.
- **Cons**: It needs higher VHDL skills to use the OSVVM methods. Since OSVVM is new feature for VHDL, it has lower flexibility for randomization and random constraint in compare with System Verilog. This weakness must be compensated by programmer. Coverage report also is difficult to be understand by the engineer.

5.1.3 Functional verification with System Verilog (Test_2)

- **Pros**: This test provides an advanced methodology by using System Verilog which is a powerful professional verification language. It has random constraint stimulus generation and functional verification. System Verilog has very powerful assertions that is known as System Verilog Assertion (SVA).
  It has an advanced and graphical functional coverage report. System Verilog is flexible and simple to use and can save the verification time.
- **Cons**: The requirement of new tools such as Mentor Questasim and learning system Verilog language is a cost for the VHDL verification team.

5.1.4 Universal verification Methodology (Test_3)

- **Pros**: Similar to test_3 it has advantages of reusability for components and skills. It is an appropriate methodology for complex designs. UVM is widely used and have support of many verification teams and vendors.
- **Cons**: Tool cost, need higher skills in System Verilog language.

5.1.5 Complex UVM architecture (Test_4)

- **Pros**: Similar to test_3 it has advantages of reusability for components and skills. It is an appropriate Methodology for a complex design. UVM is widely used and have support of many verification teams and vendors.
- **Cons**: Tool cost, need higher skills in System Verilog language.

5.2 Future work

The work described in this thesis has been concerned with the development of different tests and a wide area of technologies and tools. Based on the decision of the design team with support of the provided test environments in this thesis work, future work can be directed to the chosen test method to verify it in more details.

References

[1] “VHDL_designers_guide,” Doulos, [Online]. Available: https://www.doulos.com/knowhow/vhdl_designers_guide/what_is_vhdl/. [Accessed 21 03 2014].

[2] J. Aynsely, “Youtube,” Dolous, [Online]. Available: http://www.youtube.com/watch?v=qaSX0s_Qvm4. [Accessed 21 03 2014].

[3] "Cadence," [Online]. Available: http://www.cadence.com/products/fv/pages/languages.aspx. [Accessed 21 03 2014].

[4] “what_is_verilog,” [Online]. Available: www.doulos.com/knowhow/verilog_designers_guide/what_is_verilog. [Accessed 18 12 2014].

[5] “The e-Hardware Verification Language,” Cadence, [Online]. Available: http://www.cadence.com/products/fv/pages/e_overview.aspx. [Accessed 21 March 2014].

[6] “e_overview,” [Online]. Available: http://www.cadence.com/products/fv/pages/e_overview.aspx.

[7] “open-vera official web page,” [Online]. Available: http://www.open-vera.com/.

[8] “systemverilog_overview,” [Online]. Available: http://www.cadence.com/products/fv/pages/systemverilog_overview.aspx.

[9] “accellera systemc,” [Online]. Available: http://www.accellera.org/downloads/standards/systemc.

[10] “doulos systemc,” [Online]. Available: http://www.doulos.com/knowhow/systemc/.

[11] “doulos psl,” [Online]. Available: https://www.doulos.com/knowhow/psl/. [Accessed 18 12 2014].

[12] “ieee standard,” [Online]. Available: http://standards.ieee.org/findstds/standard/1850-2010.html. [Accessed 18 12 2014].

[13] E. C. A. H. A. N. Janick Bergeron, Verification Methodology manual for system verilog, New York,: Springer Science+Business Media, 2005.

[14] “sunburst-design SVA_Bind.pdf,” [Online]. Available: http://www.sunburst-design.com/papers/CummingsSNUG2009SJ_SVA_Bind.pdf. [Accessed 18 12 2014].

[15] “mentor abv,” [Online]. Available: http://www.mentor.com/products/fv/methodologies/abv/. [Accessed 18 12 2014].

[16] A. Hemani, “IL2450 - System Level Validation course,” [Online]. Available: http://www.ict.kth.se/courses/IL2450. [Accessed 25 7 2014].

[17] A. Hemani, “IL2450 System validation course SystemVerilogFunctionalCoverage,” [Online]. Available: http://www.ict.kth.se/courses/IL2450/files/slides/SystemVerilogFunctionalCoverage.pdf.

[18] COMPREHENSIVE FUNCTIONAL VERIFICATION, B Wile, JC Goss, W Roesner.

[19] “osvvm,” [Online]. Available: http://osvvm.org/about-os-vvm.

[20] “doulos OSVVM,” [Online]. Available: https://www.doulos.com/knowhow/vhdl_designers_guide/OSVVM/.

[21] ALDEC OPEN SOURCE VHDL VERIFICATION METHODOLOGY User’s Guide, ALDEC .

[22] “accellera uvm,” [Online]. Available: http://www.accellera.org/downloads/standards/uvm. [Accessed 12 2014].

[23] “verificationacademy basic-uvm,” [Online]. Available: https://verificationacademy.com/courses/basic-uvm. [Accessed 12 2014].

[24] “altera modelsim,” [Online]. Available: http://www.altera.com/products/software/quartus-ii/modelsim/qts-modelsim-index.html. [Accessed 12 2014].

[25] “mentor modelsim,” [Online]. Available: http://www.mentor.com/products/fv/modelsim/. [Accessed 05 2014].

[26] “mentor questa,” [Online]. Available: http://www.mentor.com/products/fv/questa/. [Accessed 05 2014].

[27] “cadence enterprise_verifier,” [Online]. Available: http://www.cadence.com/products/fv/enterprise_verifier/pages/default.aspx. [Accessed 12 2014].

[28] “synopsys VCS,” [Online]. Available: http://www.synopsys.com/Tools/Verification/FunctionalVerification/Pages/VCS.aspx. [Accessed 12 2014].

[29] “aldec riviera-pro,” [Online]. Available: https://www.aldec.com/en/products/functional_verification/riviera-pro. [Accessed 12 2014].

[30] “cadence enterprise_specman_elite,” [Online]. Available: http://www.cadence.com/products/fv/enterprise_specman_elite/pages/default.aspx. [Accessed 12 2014].

[31] “UVM coock book,” Menthor Graphic Verification Academi .

[32] R. (. Purisai, “How to choose a verification methodology,” 07 09 2004. [Online]. Available:

http://www.eetimes.com/document.asp?doc_id=1217826. [Accessed 24 03 2014].