Subsystem Testbench Development and Verification using System Verilog UVM

Major Project Report

Submitted in partial fulfillment of the requirements
for the degree of

Master of Technology
in
Electronics and Communication Engineering
(Communication Engineering)

By

Harshkumar Desai
(18mecc06)

Department of Electronics and Communication Engineering
Institute of Technology
Nirma University
Ahmedabad-382481

May 2020

Subsystem Testbench Development and Verification using System Verilog UVM

Major Project Report

Submitted in partial fulfillment of the requirements
for the degree of

Master of Technology
in
Electronics and Communication Engineering
(Communication Engineering)

By

Harshkumar Desai
(18MECC06)

Under the Guidance of

External Project Guide:
Melvin Simon
HPG MMSS VAL,
Intel Technology India Pvt. Ltd.
Bangalore

Internal Project Guide:
Dr. Mangal Singh
Associate Professor
Institute of Technology,
Nirma University,
Ahmedabad

Department of Electronics and Communication Engineering
Institute of Technology
Nirma University
Ahmedabad-382481
May 2020

Certificate

This is to certify that the Major Project entitled **"Subsystem Testbench Development and Verification using System Verilog UVM"** submitted by **Harshkumar Desai (18MECC06),** towards the partial fulfillment of the requirements for the degree of Master of Technology in Communication Engineering, Nirma University, Ahmedabad is the record of work carried out by him under our supervision and guidance. In our opinion, the submitted work has reached a level required for being accepted for examination.The results embodied in this major project, to the best of our knowledge, haven't been submitted to any other university or institution for award of any degree or diploma.

Date: Place: Ahmedabad

Dr. Mangal Singh
Internal Guide
Associate Proffessor
Institute of Technology
Nirma University, Ahmedabad

Dr. Yogesh Trivedi
Professor and PG Coordinator
Communication Engineering
Institute of Technology
Nirma University, Ahmedabad

Dr. Dhaval Pujara
Professor and Head of Department
Institute of Technology
Nirma University, Ahmedabad

Dr. R. N. Patel
Director
Institute of Technology
Nirma University, Ahmedabad

Certificate

This is to certify that the Major Project entitled **"Subsystem Testbench Development and Verification using System Verilog UVM"** submitted by **Harshkumar Desai (18MECC06)**, towards the partial fulfillment of the requirements for the degree of Master of Technology in Communication Engineering, Nirma University, Ahmedabad is the record of work carried out by him under our supervision and guidance. In our opinion, the submitted work has reached a level required for being accepted for examination.

Date: Place: Bangalore

Melvin Simon
HPG MMSS VAL,
Intel Technology India Pvt. Ltd.
Bangalore.

Contents

List of Figures

Declaration

This is to certify that

1. The thesis comprises my original work towards the degree of Master of Technology in Communication Engineering at Nirma University and Intel Technology India Pvt. Ltd. and has not been submitted elsewhere for a degree.
2. Due acknowledgment has been made in the text to all other material used.

Harshkumar Desai
18MECC06

Disclaimer

Acknowledgement

I take this opportunity to my express my profound gratitude and regards to my internship project guide Dr. Mangal Singh for his exemplary guidance, monitoring and constant encouragement.

I would also like to thank Mr. Shanthamoorthi Velusamy, external guide of my internship project, for his valuable time and efforts during the implementation of this project.

I would also like to cordially thanks Mr. Melvin Simon and Mr. M Avinash, mentors of my project from Intel Technology India Pvt. Ltd., for guidance, in-vigilance and encouragement regarding the project.

I would also like to thank Dr. Nagendra Gajjar and Dr. Sachin Gajjar, who visited Intel for review of the work and their precious guidance.

I thanks my Parents, faculty members, friends and colleagues for their constant support and encouragement during this project work.

-Harshkumar Desai
18MECC06

Abstract

A large System on Chip (SoC) has millions of interconnects between modules, sub-modules and ports. Verifying that these connections have been made correctly is an important step in the verification process.This is commonly begins with simulation, and then proceed to formal verification. Both methods begins with a specification of all the required interconnects and are then meant to verify that all the inputs, outputs and paths between modules (i.e. Source and Destination nets) are connected as expected. The physical number of connections to check is one issue, as designs contain thousands of wires which potentially all need to be checked for correctness. Debugging presents a secondary though often equally challenging issue. The reason is that although checking the connectivity with dynamic tests using a directed or constrained random approach will certainly find some of the connectivity bugs, any problem will show up only as a functional issue inside the blocks under test, which doesn't necessarily help in pinpointing the problem connection. The work has been carried out in this project is showing how connectivity checks can be done for such complex designs like modules connected by AXI protocol, modules connected to sub-modules and how identification of connectivity failures between modules and sub-modules. Disk Space is also a issue. if user starts running regression in disk which has not enough space then regression starts hanging or suspended. so a shall script has been developed to manage disk space by notifying user to cleanup their unwanted data from hard disk. In this script first it creates data base of every disks, the database contains user's folder location and user's name. then a unique list is created and from that unique user list script searches for folders user has created from the database. using list of user's folder lists it counts their usage and a notification mail is sent to the users.

Chapter 1

Introduction

1.1 Motivation

A System on Chip (SoC) have thousands of connections between blocks, sub-blocks and nets. Verification of these connections are connected correctly or not is an important stage in the verification process. It is commonly initiates with simulation, and then proceed to formal verification. Both methods initiates with a specification of all the required connections and are then meant to verify that all the inputs, outputs and paths between blocks (i.e. Source and Destination nets) are connected as expected.

1.2 Problem Statement

A large soc has thousands of connections.And we need to ensure connectivity between modules, sub-modules and modules to NOC at subsystem level using Formal connectivity verification, SystemVerilog assertions and Simulation connectivity verification in Functional verification process.
Assertions exhaustively verifies that every possible value on the Source net is equal to the value on the Destination net. This is a necessary but not sufficient condition for being formally connected. The assertion would also hold true if both the Source and Destination were tied to the same constant value. So, we need to ensure that connection is directional and there is structural path present between source and destination as well as source and destination should have same value when enable expression is true.

1.3 Prerequisites

- Knowledge of AXI I/O signals for modules connected With AXI Bus Protocol.
- Knowledge of UVM for making Test Cases.
- Knowledge of writing System Verilog Assertions.
- Knowledge of Shell Scripting.
- Ability to use Verdi tool.
- Ability to use JasperGold tool

1.4 Objective

Assuring that source is connected to a destination; we need to verify connectivity based on below aspects under these three conditions are met :

- They always have the same value when the Enable expression is true
- There is a structural path from Source to Destination.
- The connection is directional from Source to Destination

If we don't verify all three aspects of each connection pathway, the verification will not be completed.

1.5 Outline of Thesis

1.5.1 Chapter 1 : Introduction

In this chapter motivation, objective, problem statement for the project, which are the prerequisites has been covered.

1.5.2 Chapter 2 : Literature Survey

In this chapter, standard VLSI design life cycle has been covered which indicates a chip design process from specification declaration to fabrication and post fabrication validation. Then section 2.2 describes architecture of standard SystemVerilog Testbench its components and functions of each block in systemVerilog testbench. Section 2.3 describes architecture of a UVM test bench and describes role of each component of testbench. Section 2.4 describes AMBA AXI protocol with details of read and write operations performed in AXI protocol. Functions of each read and write signals described in this section. Section 2.5 describes systemverilog assertions. It covers assertion's components like sequence, property and implications in systemverilog assertions.

1.5.3 Chapter 3 : Connectivity Verification by UVM Testbench

In connectivity Verification by UVM Testbench chapter, Introduction section describes connectivity consideration how connectivity checks performed. section types of connectivity checks describes different types of connectivity checks like Direct point-to-point checking, Direct point-to-point checking through hierarchy,Conditional point-to-point checking,Point-to-point with delay and Point-to-point with no delay, section methods of connectivity verification shows which methods can be used for connectivity checks. two methods Simulation connectivity checks and Formal Connectivity Verification are described in this chapter.section connectivity checks done in this project describes System on chip (SoC) Architecture and Subsystem Architecture and connectivity checks done at subsystem level using assertions and using Simulation.

1.5.4 chapter 5 : Formal Connectivity Verification

In Formal Connectivity Verification chapter, there is description of how connectivity checks are being implemented and how the jaspergold tool verifies the connectivity.

1.5.5 Chapter 6 : Disk Space Management

In Intel, there is Disk Space issue if user ran a regression and if there is not enough space in disk regression gets stops for solution of this problem a script has been developed to send alert message to user regarding their disk space usage in this chapter a details of this script like description, flow chart and sample of alert message has been shown.

1.6 Summary

In Introduction chapter, Motivation about the project, Problem Statement for the project, which are the prerequisites for the project and objective of project and Outline if the thesis has been covered.

Chapter 2

Literature Survey

2.1 Standard VLSI Design Life Cycle

Very-large-scale integration (VLSI) is the process of creating an integrated circuit (IC) by combining thousands of transistors into a single chip. VLSI began in the 1970s when complex semiconductor and communication technologies were being developed. The microprocessor is a VLSI device. [6]

Before the introduction of VLSI technology, most ICs had a limited set of functions they could perform. An electronic circuit might consist of a CPU, ROM, RAM and other glue logic. VLSI lets IC designers add all of these into one chip. [6]

The electronics industry has achieved a phenomenal growth over the last few decades, mainly due to the rapid advances in large scale integration technologies and system design applications. With the advent of very large scale integration (VLSI) designs, the number of applications of integrated circuits (ICs) in high-performance computing, controls, telecommunications, image and video processing, and consumer electronics has been rising at a very fast pace. [6]

The current cutting-edge technologies such as high resolution and low bit-rate video and cellular communications provide the end-users a marvelous amount of applications, processing power and portability. This trend is expected to grow rapidly, with very important implications on VLSI design and systems design. [6]

The block diagram represents standard VLSI Design Life Cycle in which various stages involved in Specification to fabrication. [6]

1. **Specification:** This is the first step in the design process where we define the important parameters of the system that has to be designed into a specification. [6]

2. **High level Design:** In this stage, various details of the design architecture

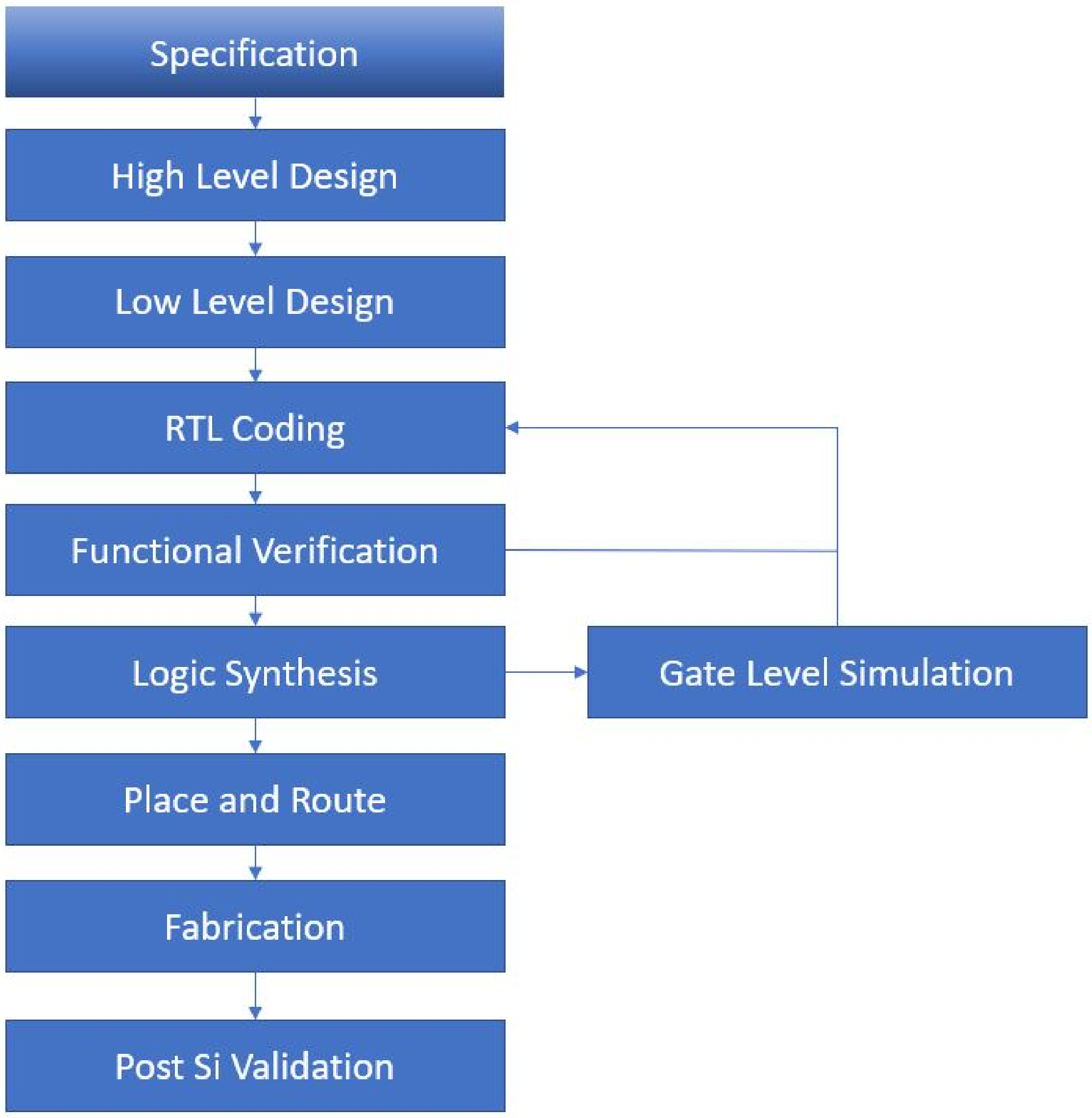


Figure 2.1: VLSI Design Life Cycle

are defined. In this stage, details about the different functional blocks and the interface communication protocols between them etc. are defined. [6]

3. **Low level Design:**This phase is also known as microarchitecture phase. In this phase lower level design details about each functional block implementation are designed. This can include details like modules, state machines, counters, MUXes, decoders, internal registers etc. [6]

4. **RTL coding:**In RTL coding phase, the micro design is modelled in a Hardware Description Language like Verilog/VHDL, using synthesizable constructs of the language. Synthesizable constructs are used so that the RTL model can be input to a synthesis tool to map the design to actual

gate level implementation later. [6]

5. **Functional Verification:**Functional Verification is the process of verifying the functional characteristics of the design by generating different input stimulus and checking for correct behavior of the design implementation. [6]

6. **Logic Synthesis:**Synthesis is the process in which a synthesis tool like design compiler takes in the RTL, target technology, and constraints as inputs and maps the RTL to target technology primitives. Functional equivalence checks are also done after synthesis to check for equivalence between the input RTL model and the output gate level model. [6]

7. **Placement and Routing:** Gate-level netlist from the synthesis tool is taken and imported into place and route tool in the Verilog netlist format. All the gates and flip-flops are placed, Clock tree synthesis and reset is routed. After this each block is routed, output of the PR tool is a GDS file, which is used by a foundry for fabricating the ASIC. [6]

8. **Gate level Simulation:**The Placement and Routing tool generates an SDF (Standard Delay File) that contains timing information of the gates. This is back annotated along with gate level netlist and some functional patterns are run to verify the design functionality. A static timing analysis tool like Prime time can also be used for performing static timing analysis checks. [6]

9. **Fabrication:**Once the gate level simulations verify the functional correctness of the gate level design after the Placement and Routing phase, then the design is ready for manufacturing. The final GDS file (a binary database file format which is the default industry standard for data exchange of integrated circuit or IC layout artwork) is normally send to a foundry which fabricates the silicon. Once fabricated, proper packaging is done and the chip is made ready for testing. [6]

10. **Post silicon Validation:**Once the chip is back from fabrication, it needs to be put in a real test environment and tested before it can be used widely in the market. This phase involves testing in lab using real hardware boards and software/firmware that programs the chip. Since the speed of simulation with RTL is very slow compared to testing in lab with real silicon, there is always a possibility to find a bug in silicon validation and hence this is very important before qualifying the design for a market. [6]

2.2 SystemVerilog TestBench

2.2.1 About TestBench

TestBench or Verification environment is a group of classes or components. where each component is performing a specific operation. i.e, generating stimulus, driving, monitoring, etc. and those classes will be named based on the operation. [7] Systemverilog Testbench is Shown Below:

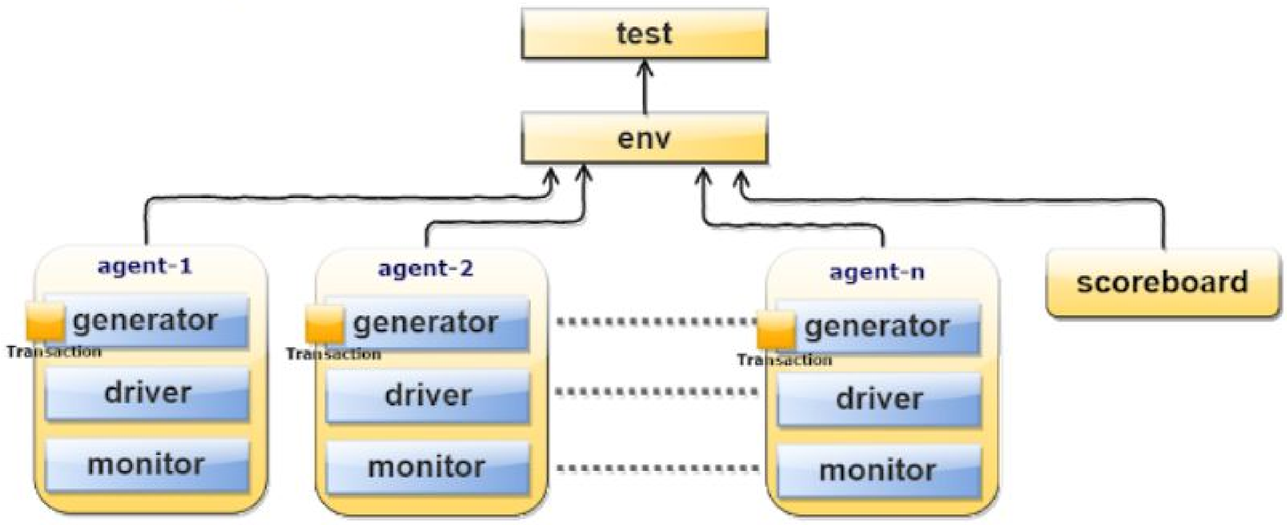


Figure 2.2: SystemVerilog TestBench [1]

2.2.2 Testbench Component

1. **Trnasaction:** Defines the pin level activity generated by agent (to drive to DUT through the driver) or the activity has to be observed by agent (Placeholder for the activity monitored by the monitor on DUT signals). [1]

2. **Generator:** Generates the stimulus (create and randomize the transaction class) and send it to Driver. [1]

3. **Driver:** Receives the stimulus (transaction) from the generator and drives the packet level data inside a transaction into pin level (to DUT). [1]

4. **Monitor:** Observes pin level activity on interface signals and converts into packet level which is sent to the components such as scoreboard. [1]

5. **Agent:** An agent is a container class, which groups the class's (generator, driver, and monitor) specific to an interface or protocol. [1]

6. **Scoreboard:** Receives data items from monitors and compares them with expected values. Expected values can be either golden reference values or generated from the reference model. [1]

7. **Environment:** The environment is a container class for grouping higher level components like agent's and scoreboard. [1]

8. **Test:** The test is responsible for,

    - Configuring the testbench.
    - Initiate the testbench components construction process.
    - Initiate the stimulus driving.

9. **Testbench-top:** This is the topmost file, which connects the DUT and TestBench. It consists of DUT, Test and interface instances, the interface connects the DUT and TestBench. [1]

2.3 Universal Verification Methodology (UVM)

The Universal Verification Methodology (UVM) consists of class libraries needed for the development of well constructed, reusable SystemVerilog based Verification environment. In simple words, UVM consists of a set of base classes with methods defined in it, the SystemVerilog verification environment can be developed by extending these base classes. [3]

2.3.1 UVM testbench

Role of each testbench element is explained below,

- **UVM test:** The test is the topmost class. the test is responsible for, [3]
    - configuring the testbench.
    - Initiate the testbench components construction process by building the next level down in the hierarchy ex: env.
    - Initiate the stimulus by starting the sequence.

- **UVM Environment:** Env or environment: The environment is a container component for grouping higher level components like agent's and scoreboard. [3]

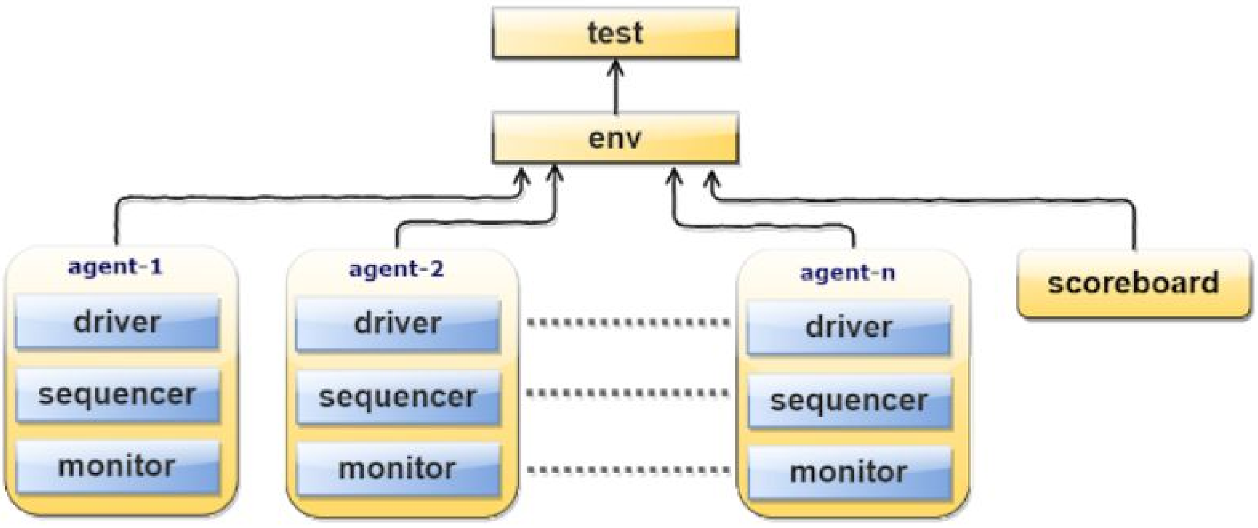


Figure 2.3: UVM testbench [2]

- **UVM Agent:**UVM agent groups the uvm-components specific to an interface or protocol. [3] example: groups the components associated with BFM(Bus Functional Model).

  The components of an agent are,

  - **UVM Sequence item:** The sequence-item defines the pin level activity generated by agent (to drive to DUT through the driver) or the activity has to be observed by agent (Placeholder for the activity monitored by the monitor on DUT signals). [3]
  - **UVM Driver:** Responsible for driving the packet level data inside sequence-item into pin level (to DUT). [3]
  - **UVM Sequence:** Defines the sequence in which the data items need to be generated and sent/received to/from the driver. [3]
  - **UVM Sequencer:** Responsible for routing the data packet's(sequence-item) generated in sequence to the driver or vice verse. [3]
  - **UVM Monitor:** Observes pin level activity on interface signals and converts into packet level which is sent to components such as scoreboards. [3]

- **UVM Scoreboard:**Receives data item's from monitor's and compares with expected values.expected values can be either golden reference values or generated from the reference model. [3]

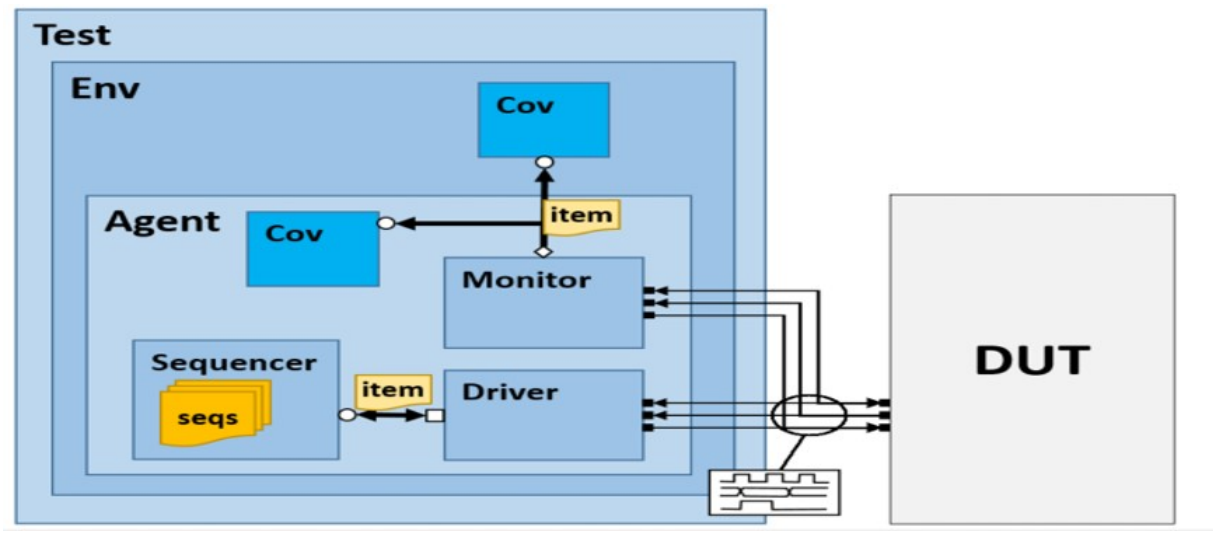


Figure 2.4: UVM TestBench Block Diagram [3]

2.4 AMBA AXI Protocol

This chapter describes the architecture of the AXI protocol and the basic transactions that the protocol defines. It contains the following sections:

- About the AXI Procol
- Architecture
- Basic transaction
- Additional features

2.4.1 About AXI Protocol

The AMBA AXI protocol is targeted at high-performance, high-frequency system designs and includes a number of features that make it suitable for a high-speed submicron interconnect. [4]
The objectives of the latest generation AMBA interface are to:

- be suitable for high-bandwidth and low-latency designs
- enable high-frequency operation without using complex bridges
- meet the interface requirements of a wide range of components
- be suitable for memory controllers with high initial access latency
- provide flexibility in the implementation of interconnect architectures

- be backward-compatible with existing AHB and APB interfaces.

The key features of the AXI protocol are:

- separate address/control and data phases
- support for unaligned data transfers using byte strobes
- burst-based transactions with only start address issued
- separate read and write data channels to enable low-cost Direct Memory Access (DMA)
- ability to issue multiple outstanding addresses
- out-of-order transaction completion
- easy addition of register stages to provide timing closure.

As well as the data transfer protocol, the AXI protocol includes optional extensions that cover signaling for low-power operation. [4]

2.4.2 Architecture

The AXI protocol is burst-based. Every transaction has address and control information on the address channel that describes the nature of the data to be transferred. The data is transferred between master and slave using a write data channel to the slave or a read data channel to the master. In write transactions, in which all the data flows from the master to the slave, the AXI protocol has an additional write response channel to allow the slave to signal to the master the completion of the write transaction. [4]
The AXI protocol enables:

- address information to be issued ahead of the actual data transfer. [5]
- support for multiple outstanding transactions
- support for out-of-order completion of transactions.

Figure 2.5 shows how a read transaction uses the read address and read data channels.
Figure 2.6 shows how a write transaction uses the write address, write data, and write response channels.

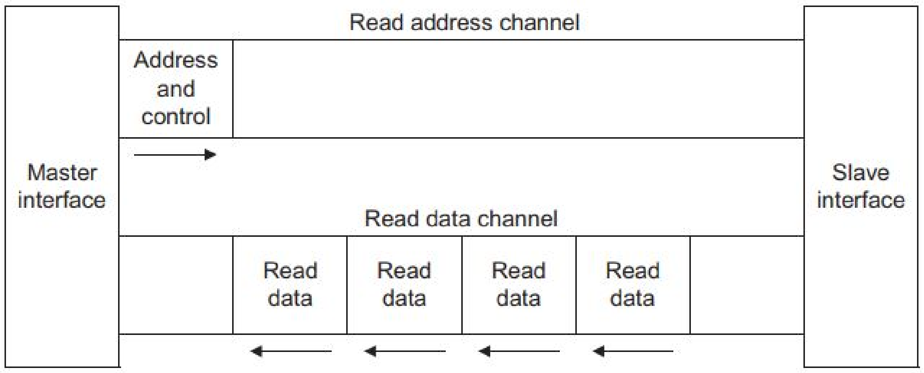


Figure 2.5: Channel architecture of reads [4]

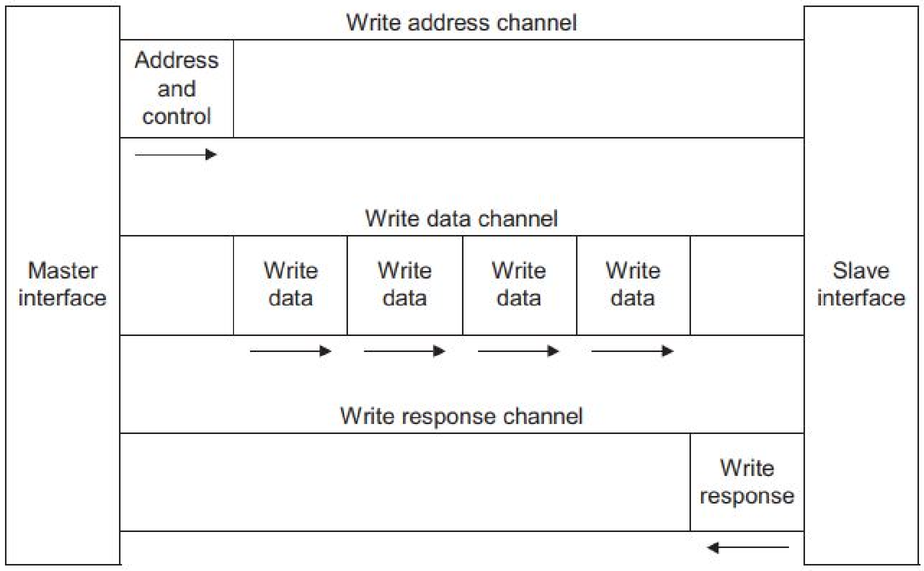


Figure 2.6: Channel architecture of writes [4]

2.4.3 Channel definition

Each of the five independent channels consists of a set of information signals and uses a two-way VALID and READY handshake mechanism. [8] The information source uses the VALID signal to show when valid data or control information is available on the channel. The destination uses the READY signal to show when it can accept the data. Both the read data channel and the write data channel also include a LAST signal to indicate when the transfer of the final data item within a transaction takes place. [9]

Read and write address channels Read and write transactions each have their own address channel. The appropriate address channel carries all of the required address and control information for a transaction. [4]
The AXI protocol supports the following mechanisms:

- variable-length bursts, from 1 to 16 data transfers per burst.
- bursts with a transfer size of 8-1024 bits.
- wrapping, incrementing, and non-incrementing bursts.
- atomic operations, using exclusive or locked accesses.
- system-level caching and buffering control.
- secure and privileged access.

Read data channel The read data channel conveys both the read data and any read response information from the slave back to the master. [4] The read data channel includes:

- the data bus, that can be 8, 16, 32, 64, 128, 256, 512, or 1024 bits wide.
- a read response indicating the completion status of the read transaction.

Write data channel The write data channel conveys the write data from the master to the slave and includes: [4]

- the data bus, that can be 8, 16, 32, 64, 128, 256, 512, or 1024 bits wide
- one byte lane strobe for every eight data bits, indicating which bytes of the data bus are valid.

Write data channel information is always treated as buffered, so that the master can perform write transactions without slave acknowledgement of previous write transactions. [4]

Write response channel The write response channel provides a way for the slave to respond to write transactions. All write transactions use completion signaling. [4]
The completion signal occurs once for each burst, not for each individual data transfer within the burst. [4]

2.4.4 Interface and interconnect

A typical system consists of a number of master and slave devices connected together through some form of interconnect, as shown in Figure The AXI protocol

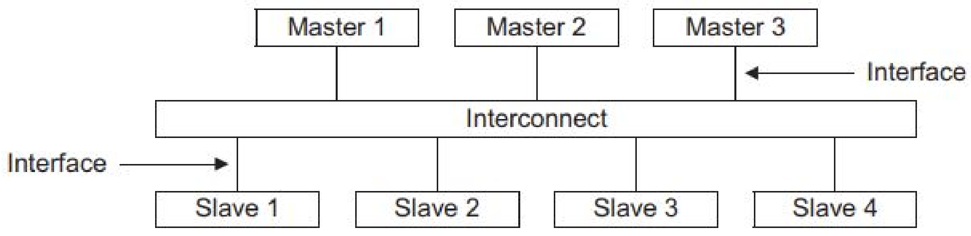


Figure 2.7: Interface and interconnect [5]

provides a single interface definition for describing interfaces:

- between a master and the interconnect
- between a slave and the interconnect
- between a master and a slave.

The interface definition enables a variety of different interconnect implementations. The interconnect between devices is equivalent to another device with symmetrical master and slave ports to which real master and slave devices can be connected. [5] Most systems use one of three interconnect approaches:

- shared address and data buses
- shared address buses and multiple data buses
- multilayer, with multiple address and data buses.

In most systems, the address channel bandwidth requirement is significantly less than the data channel bandwidth requirement. Such systems can achieve a good balance between system performance and interconnect complexity by using a shared address bus with multiple data buses to enable parallel data transfers. [5]

2.4.5 Basic Transaction

This section gives examples of basic AXI protocol transactions. Each example shows the VALID and READY handshake mechanism. Transfer of either address information or data occurs when both the VALID and READY signals are HIGH.

Read burst example Figure 2.8 shows a read burst of four transfers. In this example, the master drives the address, and the slave accepts it one cycle later. Note that master also drives a set of control signals showing the length and type of the burst, but these signals are omitted from the figure for clarity.
After the address appears on the address bus, the data transfer occurs on the read data channel. The slave keeps the VALID signal LOW until the read data is available. For the final data transfer of the burst, the slave asserts the RLAST signal to show that the last data item is being transferred.
Read address channel signals:

- **ARID[3:0]:** Read address ID. This signal is the identification tag for the read address group of signals.
- **ARADDR[31:0]:** Read address. The read address bus gives the initial address of a read burst transaction. Only the start address of the burst is provided and the control signals that are issued alongside the address detail how the address is calculated for the remaining transfers in the burst.
- **ARLEN[3:0:** Burst length. The burst length gives the exact number of transfers in a burst. This information determines the number of data transfers associated with the address.
- **ARSIZE[2:0]:** Burst size. This signal indicates the size of each transfer in the burst.
- **ARBURST[1:0]:** Burst type. The burst type, coupled with the size information, details how the address for each transfer within the burst is calculated.

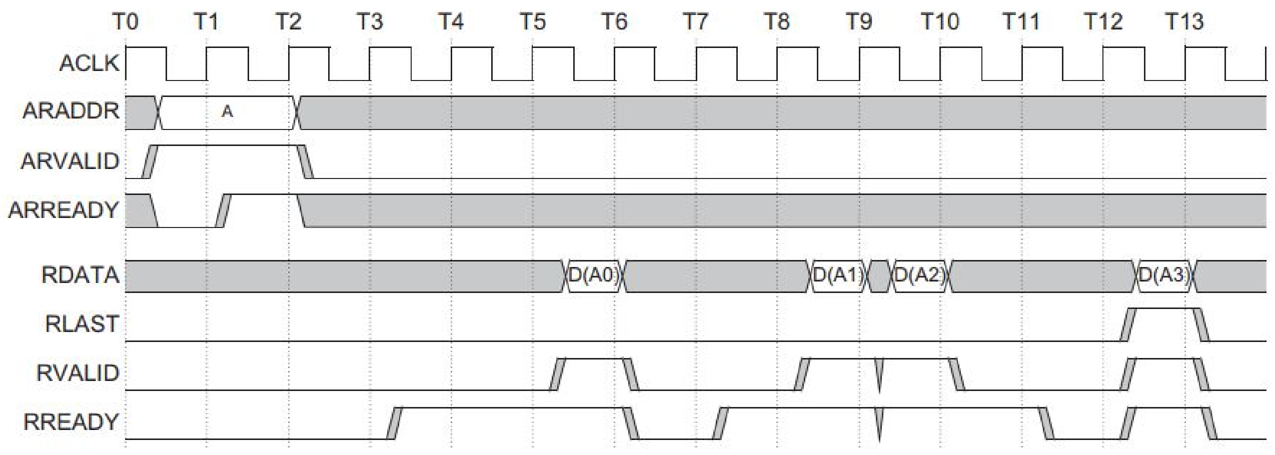


Figure 2.8: AXI Read burst [4]

- **ARLOCK[1:0]:** Lock type. This signal provides additional information about the atomic characteristics of the transfer.
- **ARCACHE[3:0]**Cache type. This signal provides additional information about the cacheable characteristics of the transfer.
- **ARPROT[2:0]:** Protection type. This signal provides protection unit information for the transaction.
- **ARVALID:** Read address valid. This signal indicates, when HIGH, that the read address and control information is valid and will remain stable until the address acknowledge signal, ARREADY, is high.
- **ARREADY:** Read address ready. This signal indicates that the slave is ready to accept an address and associated control signals: 1 = slave ready 0 = slave not ready.

Read data channel signals:

- **RID[3:0]:** Read ID tag. This signal is the ID tag of the read data group of signals. The RID value is generated by the slave and must match the ARID value of the read transaction to which it is responding.
- **RDATA[31:0]:** Read data. The read data bus can be 8, 16, 32, 64, 128, 256, 512, or 1024 bits wide.
- **RRESP[1:0]:** Read response. This signal indicates the status of the read transfer. The allowable responses are OKAY, EXOKAY, SLVERR, and DECERR.

- **RLAST:** Read last. This signal indicates the last transfer in a read burst.
- **RVALID:** Read valid. This signal indicates that the required read data is available and the read transfer can complete: 1 = read data available 0 = read data not available.
- **RREADY:** Read ready. This signal indicates that the master can accept the read data and response information: 1= master ready 0 = master not ready.

Write burst example Figure 1-6 shows a write transaction. The process starts when the master sends an address and control information on the write address channel. The master then sends each item of write data over the write data channel. When the master sends the last data item, the WLAST signal goes HIGH. When the slave has accepted all the data items, it drives a write response back to the master to indicate that the write transaction is complete. [4]
Write address channel signals:

- **AWID[3:0]:** Write address ID. This signal is the identification tag for the write address group of signals.
- **AWADDR[31:0]:** Write address. The write address bus gives the address of the first transfer in a write burst transaction. The associated control signals are used to determine the addresses of the remaining transfers in the burst.
- **AWLEN[3:0]:** Burst length. The burst length gives the exact number of transfers in a burst. This information determines the number of data transfers associated with the address.
- **AWSIZE[2:0]:** Burst size. This signal indicates the size of each transfer in the burst. Byte lane strobes indicate exactly which byte lanes to update.
- **AWBURST[1:0]:** Burst type. The burst type, coupled with the size information, details how the address for each transfer within the burst is calculated.
- **AWLOCK[1:0]:** Lock type. This signal provides additional information about the atomic characteristics of the transfer.
- **AWCACHE[3:0]:** Cache type. This signal indicates the bufferable, cacheable, write-through, write-back, and allocate attributes of the transaction.

- **AWPROT[2:0]:** Protection type. This signal indicates the normal, privileged, or secure protection level of the transaction and whether the transaction is a data access or an instruction access

- **AWVALID:** Write address valid. This signal indicates that valid write address and control information are available: 1 = address and control information available 0 = address and control information not available. The address and control information remain stable until the address acknowledge signal, AWREADY, goes HIGH.

- **AWREADY:** Write address ready. This signal indicates that the slave is ready to accept an address and associated control signals: 1 = slave ready 0 = slave not ready.

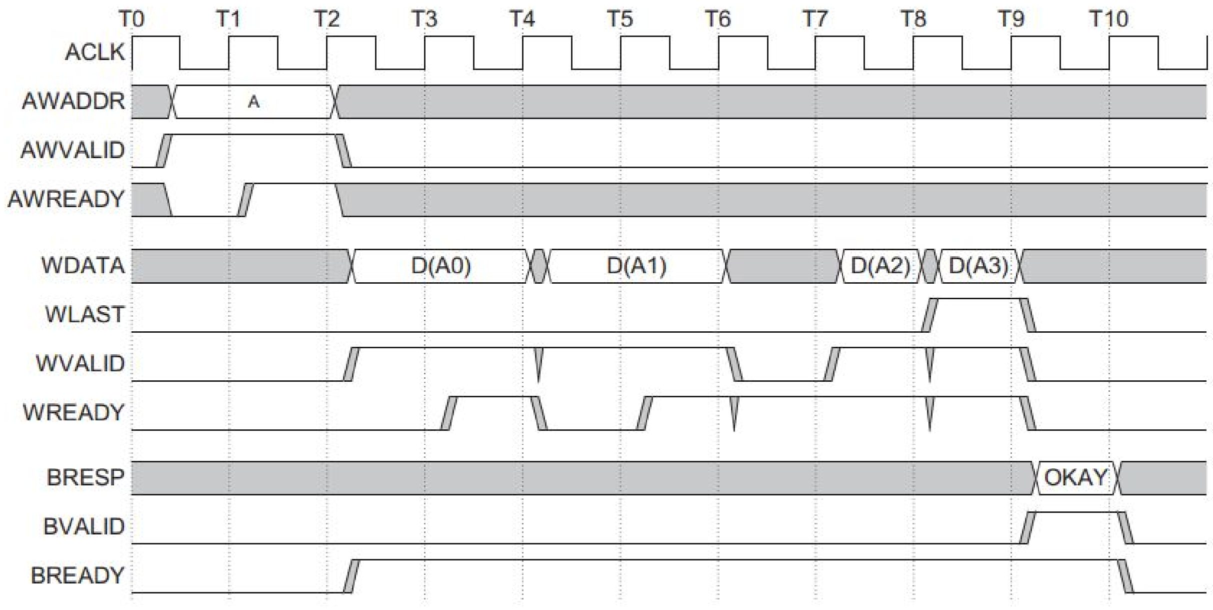


Figure 2.9: Write burst [4]

Write data channel signals:

- **WID[3:0]:** Write ID tag. This signal is the ID tag of the write data transfer. The WID value must match the AWID value of the write transaction.

- **WDATA[31:0]:** Write data. The write data bus can be 8, 16, 32, 64, 128, 256, 512, or 1024 bits wide.

- **WSTRB[3:0]:** Write strobes. This signal indicates which byte lanes to update in memory. There is one write strobe for each eight bits of the write data bus.

- **WLAST:** Write last. This signal indicates the last transfer in a write burst.
- **WVALID:** Write valid. This signal indicates that valid write data and strobes are available: 1 = write data and strobes available 0 = write data and strobes not available.
- **WREADY:** Write ready. This signal indicates that the slave can accept the write data: 1 = slave ready 0 = slave not ready.

Write response channel signals:

- **BID[3:0]:** Response ID. The identification tag of the write response. The BID value must match the AWID value of the write transaction to which the slave is responding.
- **BRESP[1:0]:** Write response. This signal indicates the status of the write transaction. The allowable responses are OKAY, EXOKAY, SLVERR, and DECERR.
- **BVALID:** Write response valid. This signal indicates that a valid write response is available: 1 = write response available 0 = write response not available.
- **BREADY:** Response ready. This signal indicates that the master can accept the response information. 1 = master ready 0 = master not ready

Overlapping read burst example:
Figure 2.10 shows how a master can drive another burst address after the slave accepts the first address. This enables a slave to begin processing data for the second burst in parallel with the completion of the first burst. [4]

2.5 SyatemVerilog Assertions

2.5.1 Introduction

Basically assertions are used to validate the behavior of a design. An assertion is a check embedded in design or bound to a design unit during the simulation. Warnings or errors are generated on the failure of a specific condition or sequence of events.
We need assertions to:

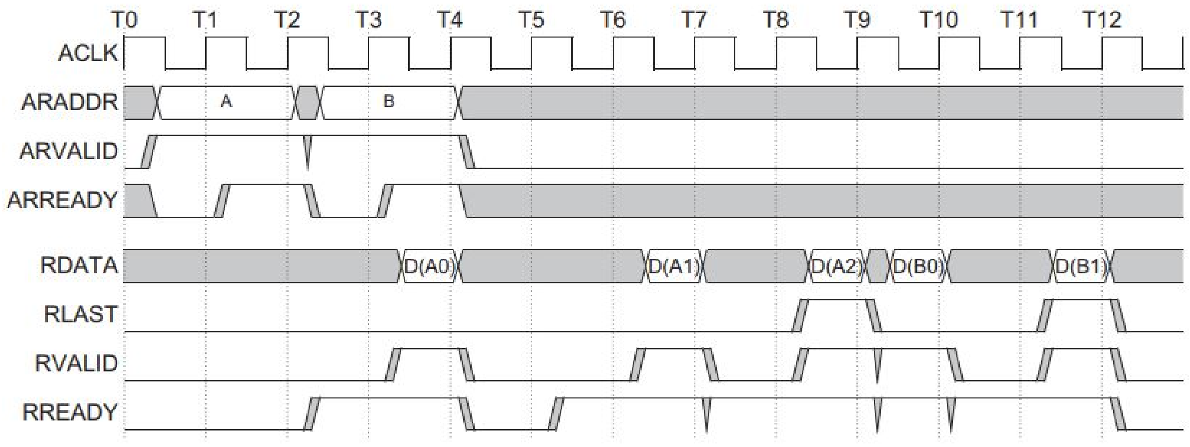


Figure 2.10: Overlapping read bursts [4]

- verify occurrence of a specific condition or sequence of events
- Provide functional coverage.

There are two types of assertions:

- Immediate Assertions
- Concurrent Assertions

Immediate assertions check: Immediate assertions check for a condition at the current simulation time.
An immediate assertion is the same as an if..else statement with assertion control. Immediate assertions have to be placed in a procedural block definition.

Concurrent Assertions: Concurrent assertions check the sequence of events spread over multiple clock cycles.

- The concurrent assertion is evaluated only at the occurrence of a clock tick.
- The test expression is evaluated at clock edges based on the sampled values of the variables involved.
- It can be placed in a procedural block, a module, an interface or a program definition.

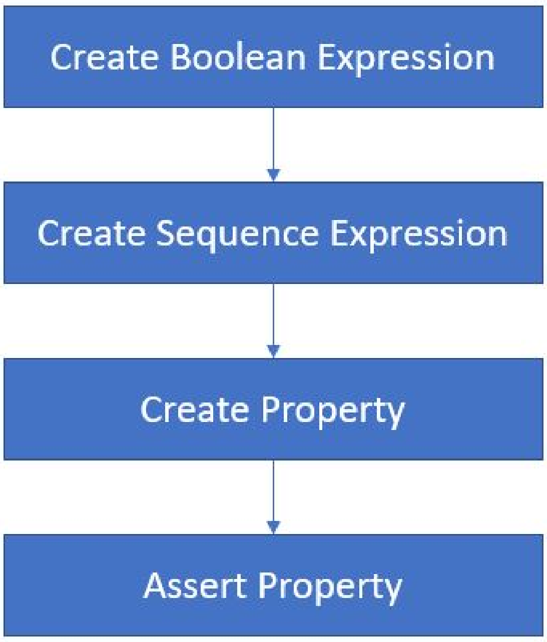


Figure 2.11: SVA Building flow

2.5.2 Building blocks of SystemVerilog Assertions

Below diagram shows the steps involved in the creation of an SVA checker,

- **Boolean expressions:** The functionality is represented by the combination of multiple logical events. These events could be simple Boolean expressions.
- **Sequence:** Boolean expression events that evaluate over a period of time involving single/multiple clock cycles. SVA provides a keyword to represent these events called "sequence."
- **Property:** A number of sequences can be combined logically or sequentially to create more complex sequences. SVA provides a keyword to represent these complex sequential behaviors called "property".
- **Assert:** The property is the one that is verified during a simulation. It has to be asserted to take effect during a simulation. SVA provides a keyword called "assert" to check the property.

Syntax for SystemVerilog Assertions

- **Sequence:**

sequence sequence-name;
Boolean expression or condition
endsequence

- **Property:**

property property-name;
test-expression or sequence-expression
endproperty

- **Assert:**

assertion-name: assert-property(property-name);

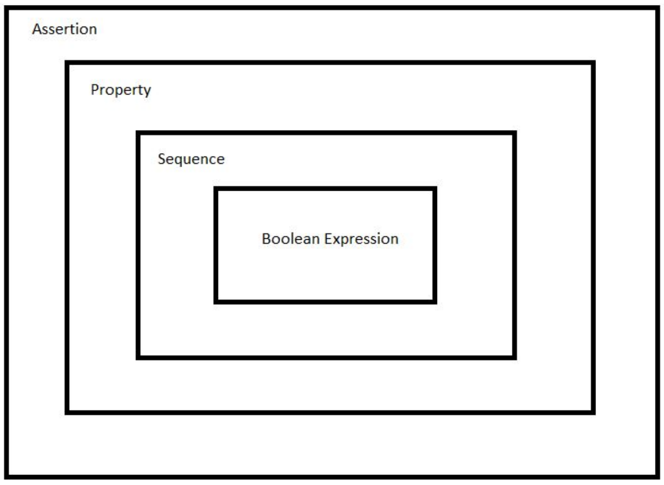


Figure 2.12: Assertion Architecture

So, it has been concluded that boolean expression is subset of sequence, sequence is subset of property and property is subset of assertion.

2.6 Summary

In this chapter, section 2.1 explains Standard VLSI Design Life Cycle has been covered which indicates a chip design process from specification declaration to fabrication and post fabrication validation. Then section 2.2 it describes Architecture of standard SystemVerilog Testbench its components and functions of each block in systemVerilog testbench. Section 2.3 describes Architecture of a UVM test bench. And describes role of each component of testbench.

Section 2.4 describes about AMBA AXI Protocol In which how a read and write operations performed in AXI protocol. functions of each read and write siganls described in this section.section 2.5 describes systemverilog assertions. It covers assertion's components like sequence, property and implications in systemverilog assertions.

Chapter 3

Connectivity Verification in Subsystem

3.1 Introduction

We can ensure connectivity between ports of module if below three conditions are satisfied.

1. Their should be same value when the Enable expression is true.
2. There should be structural path from Source to Destination.
3. The connection should be directional from Source to Destination.

So we need to ensure all of three aspects of connectivity verification.
Connectivity verification involves checking device wiring. we need to ensure that the design elements been assembled correctly or not. It is checking that the interconnects between modules of logic in a design are correct or not, for example, that output of module A1 is correctly connected to input of module A2. Some times it is also a difficult verification task. The physical number of interconnects to check is one issue, as designs contain large numbers of wires (thousands of) which potentially all need to be verified for correctness.
Debugging is also a challenging issue. The reason is that although checking the connectivity with dynamic tests using a directed or constrained random approach will certainly find some of the connectivity bugs, any problem will show up only as a functional issue inside the blocks under test, which doesn't necessarily help in pinpointing the problem connection. Use of assertions may alleviate the debug problem by catching design errors at their source. However, the volume of checking required can still be staggering. [10]
In response to such challenges, formal verification offers us a solution that is quick, exhaustive and allows for efficient debug. It's true that traditionally, chip-level formal verification is impractical. The approach usually targets the block level to keep the size of the state space to an appropriate level. But given

that connectivity checking is focused solely on the wiring — which is generally a simple part of the device, compared to the complexity found at the block-level — the state space can with some assumptions be reduced to a manageable size. The nature of this simplification depends on the type of checking that is required. [10]
After first outlining several types of connectivity checks, this paper then provides details, including code, of a new semi-automated verification flow used by several Mentor Graphics customers to simplify connectivity checking. The flow is based on a script-based environment, about which sufficient information is provided to begin implementing the new verification approach [10]

3.2 Types of point-to-point connectivity checks

Below are several types of checking connectivity:

- Direct point-to-point checking
- Direct point-to-point checking through hierarchy
- Conditional point-to-point checking
- Point-to-point with delay
- Point-to-point with no delay

Direct point-to-point checking
The simplest form of connectivity checking is point-to-point verification. it checks that whether port A is connected to port B or not at the same level of hierarchy.
For example, if a design has five sub-modules which all are located at the top level, verification entails simply verifying the connections between these five blocks and the top level.
In this case, rather than model the whole device, we could simply blackbox the five sub-blocks. There is no need to read the HDL for these blocks as the checking is not dependent on the block contents.
Direct point-to-point checking through hierarchy
This is nearly same as in nature to the simple checking approach, but contains checking that a port on a module in one level of hierarchy is physically and correctly connected to a module in a other level of hierarchy, or that a single port at the source connects to multiple end points.

For example, consider a write enable for a memory. This could originate as a single top level input pin but could be connected to many memory instances across many different hierarchical locations.
As hierarchy is now involved, the upper module instances cannot be uniformly black boxed. Black boxing should be performed at the highest hierarchical level, but only for those blocks which do not sit on the path of our write enable, or of any logic that may affect the write enable connectivity. Though more challenging and requiring some design knowledge, this more selective black boxing approach still allows for significant simplification of the state space.
Conditional point-to-point checking
A connection between two points might be conditional on some other condition in the system or the state of another signal. like, when verifying pin multiplexing, the selected I/O path will be dependent on the value of the control signal. Compounding matters, the destination of the signal may be an inverse value, which also may need to be considered when implementing the checking.
Point-to-point with delay
There may be cases where a point-to-point connection is expected, but the propagation may take a number of cycles rather than being immediate. This accordingly requires point-to-point checks to be refined.
Point-to-point with no delay
This is similar to the simple point to point checking described earlier with one significant difference: The user requires that the check explicitly verify that not only is A connected to B but also that no sequential logic appears on the path.

3.3 Connectivity verification methods

There are two methods to ensure connectivity.

1. Simulation connectivity verification

2. Formal connectivity verification

1. **Simulation connectivity verification**
   When checking connectivity by simulation, we force a value on to the Source signal and check it at the Destination signal. If we force enough values we can say that the Source is connected to the Destination, but we have not proven that there is a connection, only that we observed the same values on the Destination as on the Source. If there are muxes in the path,

we aso have to specify the values that make the Source value propagate to the Destination.

A connection in an SoC has a direction from input port to output port and in simulation, the direction of the connection is implicitly verified by the test methodology. If a value is driven on to the input (Source) it can be observed on the output (Destination), but if we drive a value on the output port it will not be reflected on the input port and the verification fails.

Connectivity verification in simulation does not verify whether a structural path exists, so we are only verifying the first and third requirement of the formal definition of 'connected'.

2. **Formal connectivity verification** By formal analysis for connectivity verification,we can have a much stronger proof that the Source and Destination are properly connected. Instead of forcing discrete values on to the Source and checking them at the Destination when the Enable expression is valid, we can use assertions to prove that source and destination are properly connected. in SystemVerilog assertions are used to check source and destination signal are same or not. for that following assertion can be used:

$$assertproperty@(posedgeclk)(enable|->(source==destination)) \tag{3.1}$$

This exhaustively verifies that every possible value on the Source port is equal to the value on the Destination port.

3.4 Connectivity Checking

3.4.1 Introduction

System-on-chip (SoC) is a circuit that contains components like processor, bus, and other elements on a single monolithic substrate. Various circuits like RAM, ROM, signal processing units, I/O interfaces, mixed signal circuits and logic circuits, are each together together forms on a single chip. As technology advances, the integration of the various units included in a SoC design becomes increasingly complicated. In the industry, a system-on-chip is formed by combining together multiple stand-alone designs of large-scale integrated circuits to provide full functionality for an application. In this project goal is to ensure connectivity at subsystem level. Subsystem is a component in System on chip, which is is responsible for some specific task. So a system on chip is formed by multiple subsystems. Above diagram shows a example of system on chip.

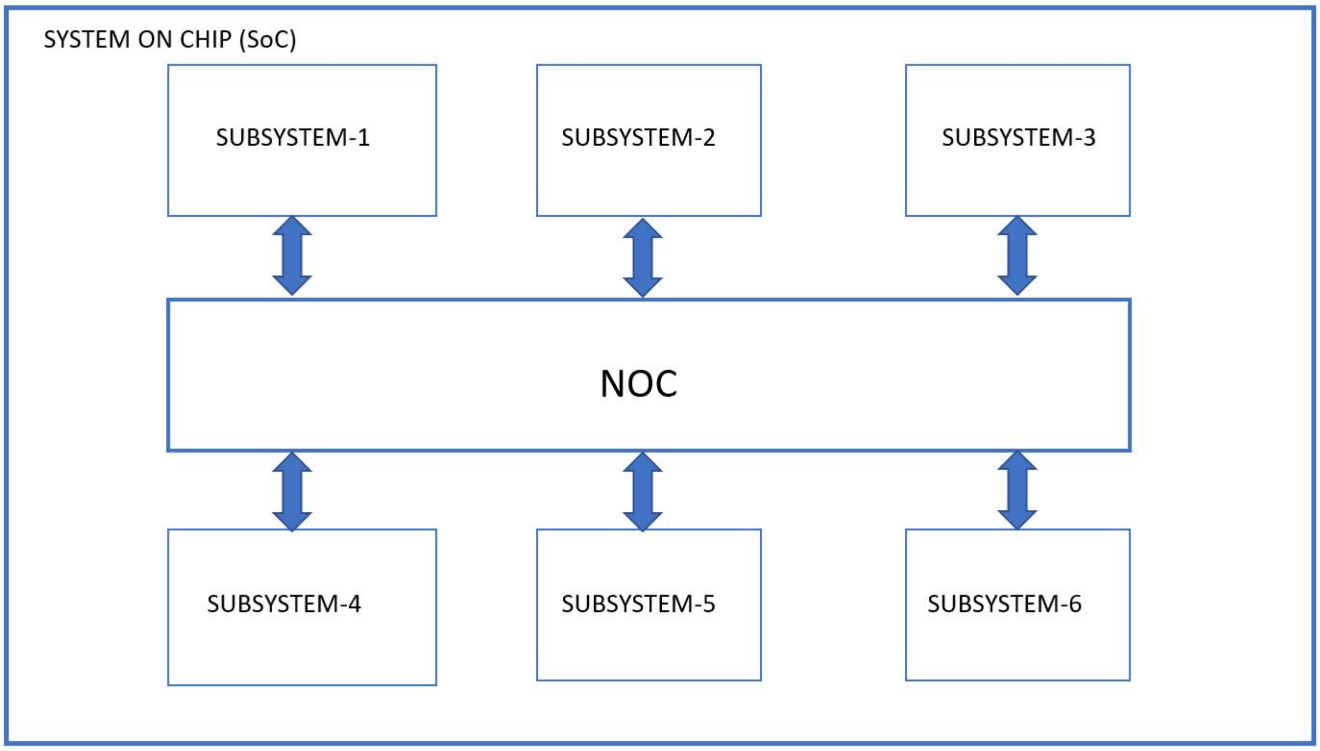


Figure 3.1: Architecture of System on Chip

3.4.2 Connectivity checks at subsection level

subsystem contains multiple modules inside it so we need to ensure that when all siganls are connected properly or not. and we also need to ensure that signal is properly through multiplexers and de-multiplexers and thruogh hierarchies. Figure below shows architecture of a subsystem for which we need to ensure connectivity for module connected by different bus protocols like AXI, APB, etc.

Connectivity checking is done by two methods

1. **Connectivity checking by Simulation**
   In simulation we check connectivity by forcing some value at source side and comparing at destination if source and destination are not same then it should display error. Figure below shows enable source and destination siganls. which are showing same value. In figure above Figure fuse-hd1prf-RM[3:0] is a 4 bit bus connected to multiple signals. fuse-hd1prf-RM[3:0]

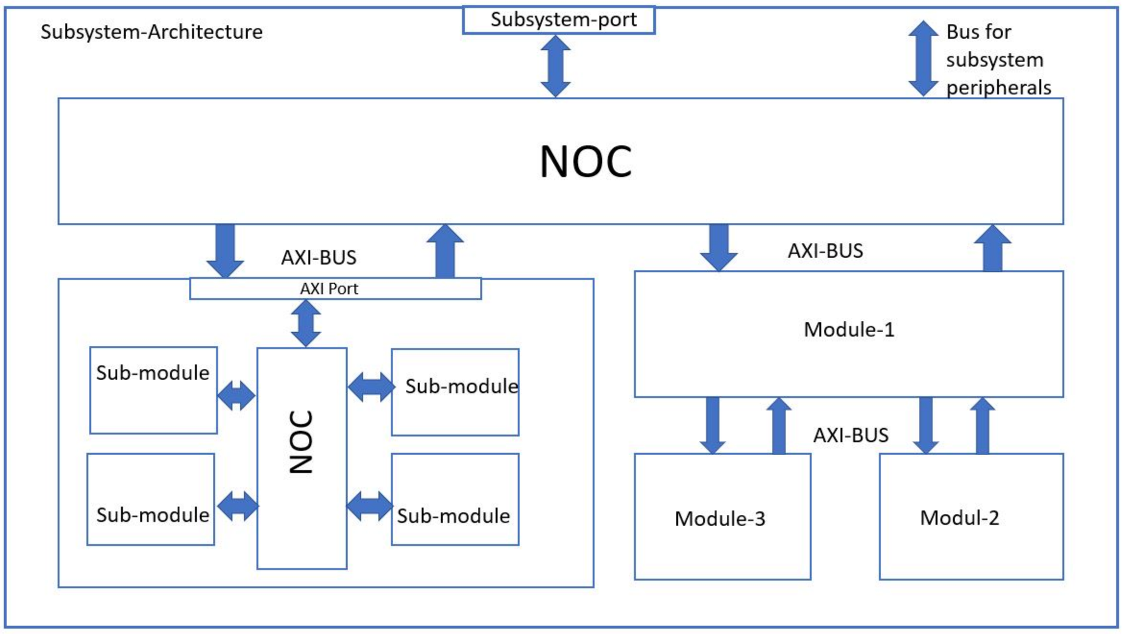


Figure 3.2: Architecture of Subsystem

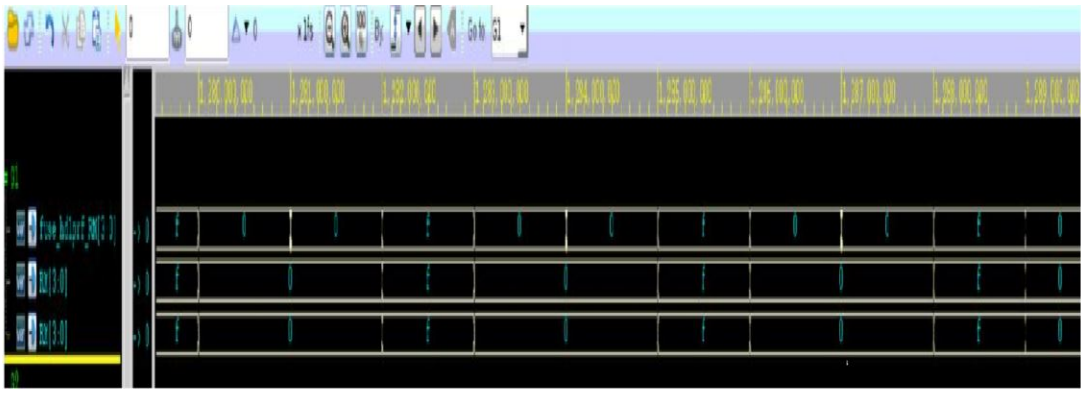


Figure 3.3: Connectivity checking by simulation

is a suorce signal and RM[3:0] is a destination signal. when source is becoming f at the same time we destination signals RM[3:0] becoming f and when source is becoming 0 at the same time we destination signals RM[3:0] becoming 0. so we can say that source and destination signals shows same. this will make us confident that functionality of these signals is correct. but problem is with this approach is we can not check connectivity all the time and say source and destination are connected. for that we need assertions for checking functionality at every positive edge of clock.

2. **Connectivity checking by Assertion**
   In assertions we ensure functionality of signals like when enable signals becomes low to high we check that input signal and output signals are same

or not.
below is a sample code given for connectivity checking assertion.

```
module assertions-example
input clk,
input restet
input enable,
input source,
output destination,

property connectivity-check;
@(posedge clk) disable iff (reset)
```

$$enable|- > source == destination; \tag{3.2}$$

```
endproperty

connectivity-: assert property (connectivity-check)
‘uvminfo(”COMPUTE ASSERTIONS”, ”CONNECTIVITY CHECK PASS”,
UVM-LOW)
else
-error(-time,”connectivity-check-fail”);
endmodule
```

3.5 Summary

In Connectivity Verification in Subsystem chapter, Introduction section describes Connectivity consideration how connectivity checks performed. section types of connectivity checks describes different types of connectivity checks like Direct point-to-point checking, Direct point-to-point checking through hierarchy,Conditional point-to-point checking,Point-to-point with delay and Point-to-point with no delay, section methods of connectivity verification shows which methods can be used for connectivity checks. two methods Simulation Connectivity checks and Formal Connectivity Verification are described in this chapter.section connectivity checks done in this project describes System on chip

(SoC) Architecture and Subsystem Architecture and connectivity checks done at subsystem level using assertions and using Simulation.

Chapter 4

Formal Connectivity Verification

4.1 Introduction

Connectivity verification includes verifying device wiring. it is checking that the interconnects between modules of logic in a System on Chip are correct or not. for example, that output port-pin1 on module A1 is correctly connected to input port-pin1' on module A2. This is sometimes a difficult verification job. The actual number of connections to verify is one issue, as designs contain n numbers of wires which should be verified for correctness. Debugging is also a secondary though and challenging issue. The reason is that although verifying the connectivity with dynamic tests using a directed or constrained random approach will certainly find few connectivity problems, any problem will point out only as a functional issue inside the module under test, which will not necessarily help in pointing the problem connection. Use of assertions may alleviate the debug problem by catching design issues at their origin. However, the volume of verifying required can still be staggering. Because of such challenges, formal verification provides us a solution that is instant, exhaustive and allows for faster debug. It's true that earlier, chip-level formal verification is impractical. formal verification point of view usually targets the module level to keep the size of the state space to an suitable level. But given that connectivity verifying is focuses only on the connections — which are normally a simple part of the device, compared to the difficulty found at the module-level — the state space can with some acceptation be reduced to a feasible size. The nature of this minimization depends on the type of verification that is required. [10]

4.2 Types of point-to-point connectivity checks

4.2.1 Direct point-to-point checking

The easiest form of connectivity verification is point-to-point verification is if port A connected port B or not at the same level of hierarchy.
For example, if a design has eight modules or ips which all sit at the top level, verification entails simply verify the interconnects between these eight modules and the top level.
In this case, in place of model the entire device, we could simply ignore the eight sub-modules. There is no need to read the HDL for these module as the verification is not dependent on the module contents.

4.2.2 Direct point-to-point checking through hierarchy

This is similar in nature to the simple verification approach, but involves verification that a port on a module in one level of hierarchy is physically and correctly connected to a module in a separate level of hierarchy, or that a single port at the source connects to multiple end points.[10]

As hierarchy is now involved, the upper module instances cannot be uniformly blackboxed. Blackboxing should be performed at the highest hierarchical level, but only for those blocks which do not sit on the path of our write enable, or of any logic that may affect the write enable connectivity. Though more challenging and requiring some design knowledge, this more selective blackboxing approach still allows for significant simplification of the state space.[10]

4.3 Other types of checking

There may be multiple types of verification that may required in verification in the inter-module connectivity of a System on Chip. So far we've only taken point to point verification, where A = B under all conditions at the same or different levels of hierarchy. Many connections will be of this nature but other types of verification may also be required.

4.3.1 Conditional point-to-point checking

Inter-connection between two ports can be conditional or may have some behavior in the system or the state of another signal. For example, when checking pin multiplexing, the selected I/O path could be dependent on the value of the control signal. in some cases, the destination of the signal may have an inverse value, which also may need to be considered when implementing the verification.

4.3.2 Point-to-point with delay

There can be cases where a end-to-end connection may expected, but the propagation may take a number of cycles rather than being immediate. This accordingly requires end-to-end checks to be refined.

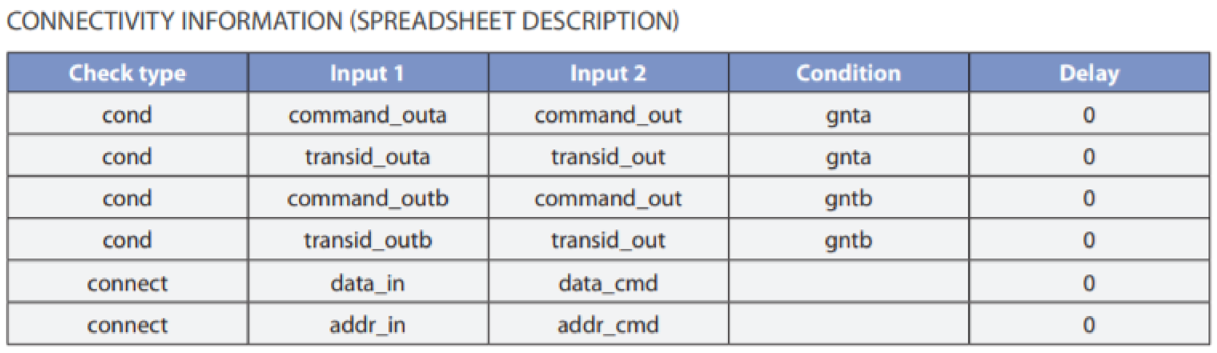
4.3.3 Point-to-point with no delay

This is similar to the simple point to point checking described earlier with onesignificant difference: The user requires that the check explicitly verify that notonly is A connected to B but also that no sequential logic appears on the path.

4.4 Constructing the checks

we need to create large number of checks that need to be created to fully check the device connectivity and a user need to create the required assertions.
A common approach is through the use of a spreadsheet constructed to a predefined format, one that details the required points which needs to be connected, any path delays involved, inversion, conditions and so on. A tool or script can then parse the spreadsheet and convert it to an assertion language, for example SystemVerilog Assertions (SVA) or the Property Specification Language (PSL). Figure shows an example of a spreadsheet description with some connectivity information.

Let's walk through the spreadsheet. We have specified two check types: "cond," which is a conditional connection; and "connect," which is a direct non-conditional connection. This will allow us to create different assertion types during the creation of the checkers. There are "Input1" and "Input2" fields, which detail the from and to points in the design for the connectivity check. The "Condition" column is used to detail any signal which needs to be set to allow the point-to-point connection to be true. For the "connect" checks, there is no condition; the check will be direct, non-conditional. Finally, all delay fields are zero, which specifies no delay on all connections. Once a spreadsheet format is fixed and populated, an appropriate tool or script can parse this spreadsheet


CONNECTIVITY INFORMATION (SPREADSHEET DESCRIPTION)

Check type	Input 1	Input 2	Condition	Delay
cond	command_outa	command_out	gnta	0
cond	transid_outa	transid_out	gnta	0
cond	command_outb	command_out	gntb	0
cond	transid_outb	transid_out	gntb	0
connect	data_in	data_cmd		0
connect	addr_in	addr_cmd		0

Figure 4.1: CSV file format

and create assertions which will be fed as targets to the formal tool. One such approach is to use generic property templates and then add the connectivity information for each property instance in a separate checker description. This can then be bound (using SystemVerilog's bind construct) to the top level of the design.[10]

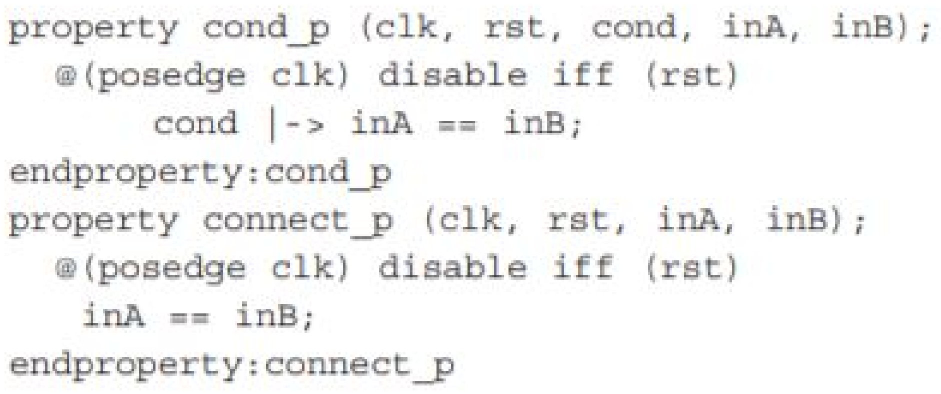
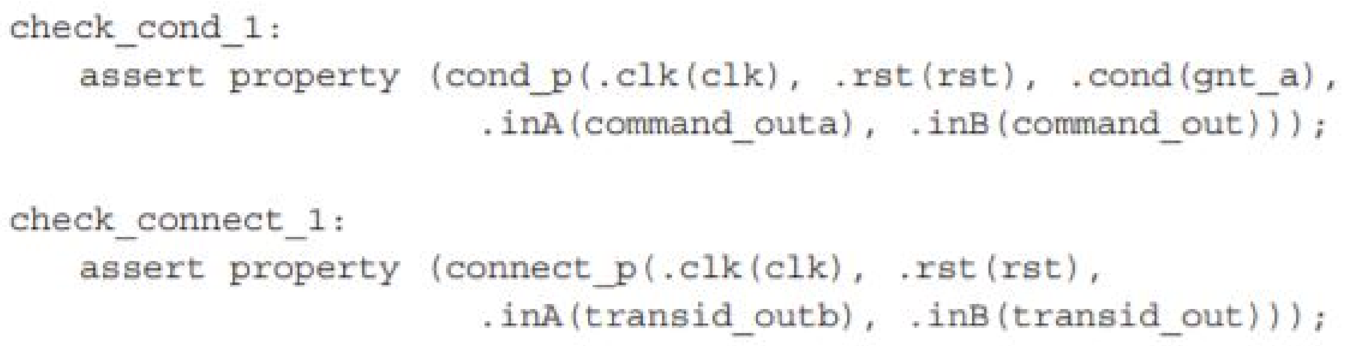

```
property cond_p (clk, rst, cond, inA, inB);
  @(posedge clk) disable iff (rst)
       cond |-> inA == inB;
endproperty:cond_p
property connect_p (clk, rst, inA, inB);
  @(posedge clk) disable iff (rst)
    inA == inB;
endproperty:connect_p
```

Figure 4.2: Two generic property templates

The property cond - p allows for conditional checking while connect - p allows for direct, non-conditional checking. This template file can contain many unique types of connectivity checks and once defined need not be edited by the user. The file becomes project-independent and reusable as it contains no design information. The source that is automatically created from the spreadsheet is the checker detail, which contains instances of the different checks from the template file, with the appropriate signal names added. An example is shown in Figure 4.3.[10]



```
check_cond_1:
   assert property (cond_p(.clk(clk), .rst(rst), .cond(gnt_a),
                         .inA(command_outa), .inB(command_out)));

check_connect_1:
   assert property (connect_p(.clk(clk), .rst(rst),
                         .inA(transid_outb), .inB(transid_out)));
```

Figure 4.3: Conditional assertions

4.5 Summary

This chapter includes details regarding how modules can be connected, types of connections like Direct point-to-point checking, Direct point-to-point checking through hierarchy, Conditional point-to-point checking, Point-to-point with delay, Point-to-point with no delay. In this chapter it is also discussed how checks are constructed and how the tool and scripts build the checks through assertions.

Chapter 5

Disk Space Management

5.1 Introduction

In Intel we have a Disk Space memory Issue number of disks are used by huge no of users. As a verification team we need to run regressions etc. that consumes high amount of memory for that user should have require space in disk otherwise their regression starts hanging or get cancelled after waiting for so much time.for clean up the space user need to remember their run area whrere they have consumption and they can cleanup unwanted data and run their data in that cleaned path .

5.2 Disk Space Usage Alert

For that a mail alert has been sent to every users who are consuming disk in team. For that a shell script has been developed by me which will make a data base in which data path of users are saved from that database data of every single users being separated with there data. then a sorted unique list of users is being created and then form that users their usage of disk in all location being counted and then a one file is created that contains data consumption and path of that file then total consumption being added everyday in a log file. Then a HTML table being constructed form unique user's consumption file. then from log file a graph of their disk consumption being created. graph is created as a jpeg image then that graph image is encoded with base64 encoding end then html table and graph sent to the user via a html mail.

5.3 Flow Chart

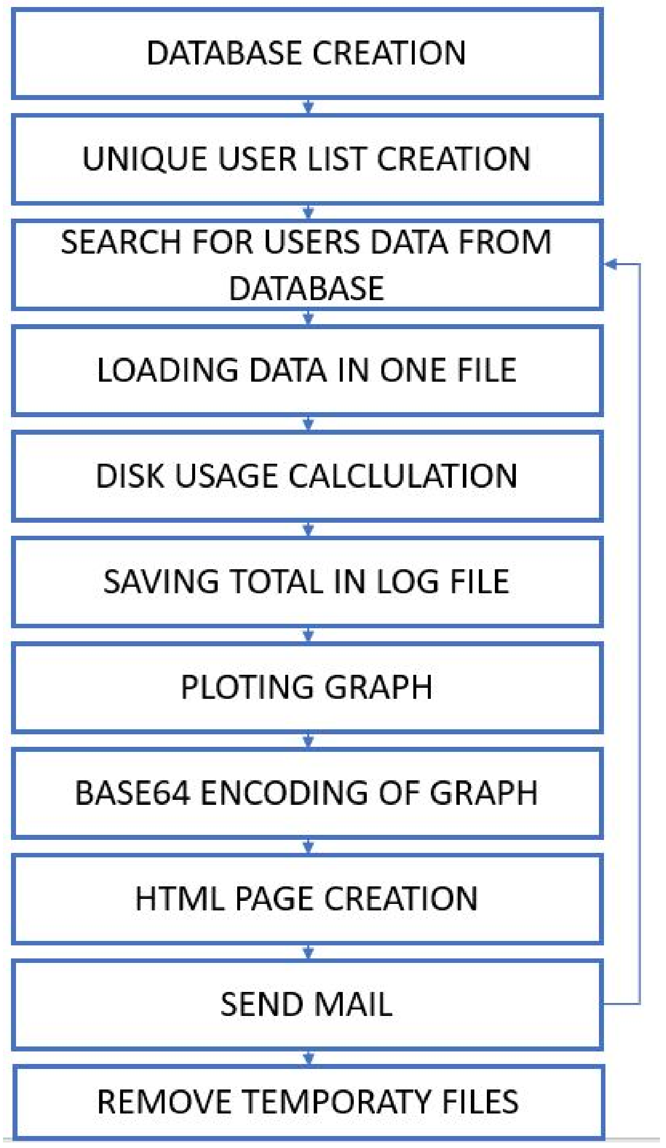


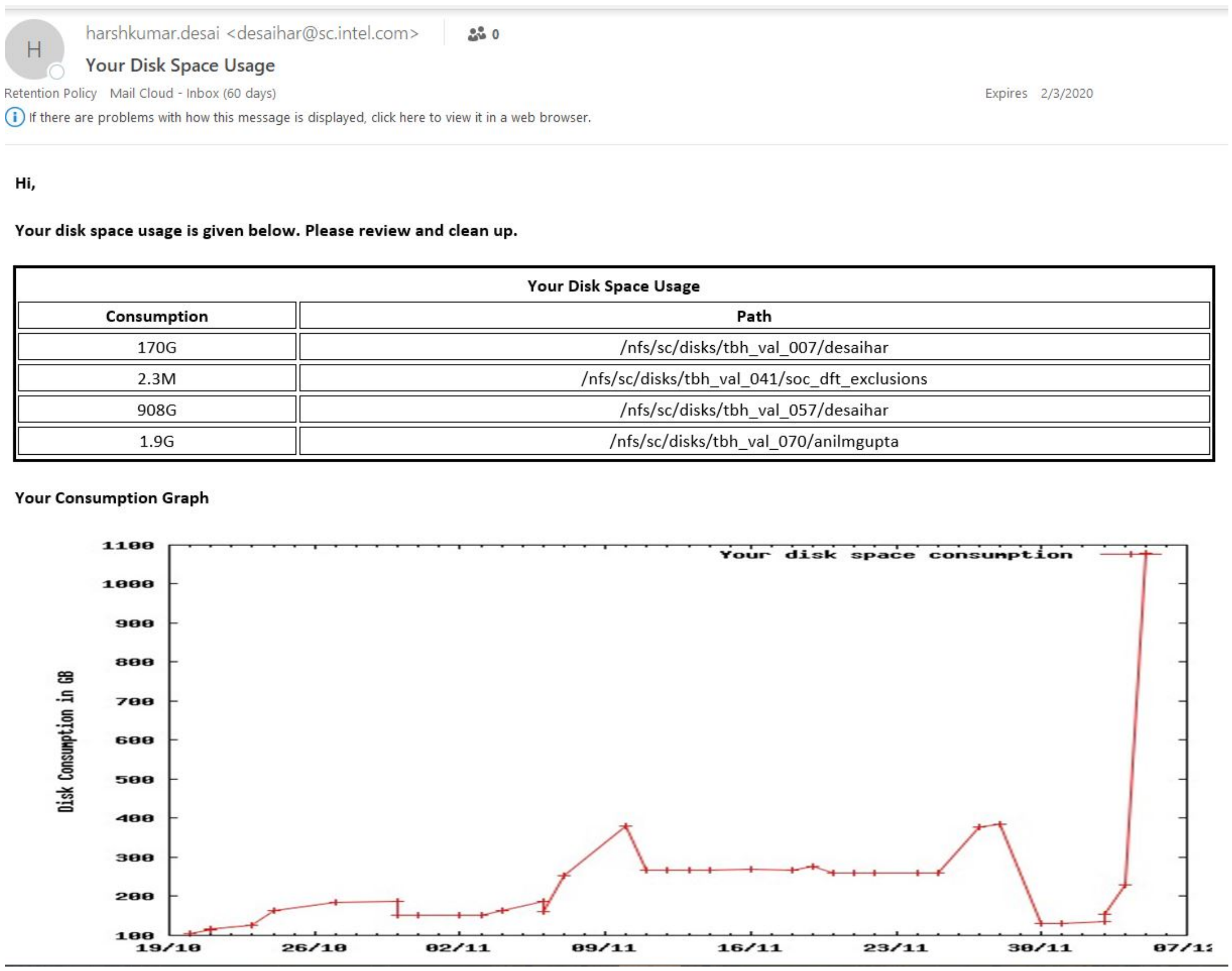
Figure 5.1: Flow Chart Of Disk Space Management

- Database creation: Shell script requires a input file which contains list of disks which needs to be in alert. Form that list file it will go inside each disks and search for folders in a disks and their owners and saves it in database file.
- Unique User List Creation: then from database, it copies username column and sort it and unique it based on user name.
- Search for user's data from database: now from each user it searches for user's data and load it into user's unique file.

- Disk Usage Calculation: From user's data file it will calculate usage of each folder and load it into user's usage file.
- Saving total into lof file: From user's usage it'll calculate total usage of all disks and load it into a log file.
- Plotting Graph: log file contains information of user's total data consumptopm and date . so using gnuplot it will genetate a graph of disk usage vs. date. and returns it in a form of jpeg.
- Base64 encoging of graph: image file can not be send directly it need to be base 64 encoded. so it encodes image as base64 and saves base64 to a saparate text file.
- HTML page creation: For creating table of disk usage and attaching image we need to create a html page. so this html page will be created automaticlly by this script.
- Sendmail: this html mail is sent by the script to user as a alert. and then flow goes to next user.
- Remove temporary files: script generates so many temporary files durina a run. after a complete run at the end it will delete all temprory files.

After completion of all these steps script will send mail to users regarding their data path and usage that helps user's to remember that at which location their data has been stored, how much data has been consumed and graph helps user to realise their data consumption keeps increasing. Output mail looks like below sample mail.
Ideally graph should go in up down pattern. If a user's is not cleaning up his/her data his/her graph will show it and we can ask them to clean up old data.

5.4 Disk Alert Mail



harshkumar.desai <desaihar@sc.intel.com>

Your Disk Space Usage

Retention Policy Mail Cloud - Inbox (60 days) Expires 2/3/2020

If there are problems with how this message is displayed, click here to view it in a web browser.

Hi,

Your disk space usage is given below. Please review and clean up.

Your Disk Space Usage

Consumption	Path
170G	/nfs/sc/disks/tbh_val_007/desaihar
2.3M	/nfs/sc/disks/tbh_val_041/soc_dft_exclusions
908G	/nfs/sc/disks/tbh_val_057/desaihar
1.9G	/nfs/sc/disks/tbh_val_070/anilmgupta

Your Consumption Graph

Figure 5.2: Mail Alert For Disk Space

5.5 Summary

In this Chapter, Trouble shooting of a disk spacemanagemaent issue is discussed. how we have notified users for their disk space cleanup and where their data is stored and how much data they have stored in intel disks. how this informations are collacted and how this information is sent to users is discussed in this chapter.

Chapter 6

Conclusion

In this project, we can see that by comparing source and destination signals we can have idea that source and destination modules are connected but we can't say that there there is structural path from Source to Destination and connection is directional. To ensue connectivity between source and destination we need assertions that checks functionality of source and destination at every positive cycle of clock signal.Assertions verifies that every possible value on the Source port is equal to the value on the Destination port. and we can say that connection is proper between source and destination.

Bibliography

[1] S. Vijayaraghavan and M. Ramanathan, *"A practical guide for SystemVerilog assertions"*. Springer Science & Business Media, 2005.

[2] H. Height, *A practical guide to adopting the universal verification methodology (UVM).* Lulu. com, 2010.

[3] M. siemens business, *"Universal Verification Methodology UVM Cookbook "*. Intel Enterprise, 2018.

[4] A. AMBA, "Protocol version 2.0 specification," *ARM Ltd*, pp. 1–1, 2010.

[5] S. Pradeep and C. Laxmi, "Design and verification environment for amba axi protocol for soc integration," in *NCRIET-2014*, vol. 3, pp. 338–343, 2014.

[6] S.-M. Kang and Y. Leblebici, *CMOS digital integrated circuits.* Tata McGraw-Hill Education, 2003.

[7] D. I. Rich, "The evolution of systemverilog," *IEEE Design & Test of Computers*, no. 4, pp. 82–84, 2003.

[8] H. D. Foster, "Trends in functional verification: A 2014 industry study," in *Proceedings of the 52nd Annual Design Automation Conference*, p. 48, ACM, 2015.

[9] P. Gurha and R. Khandelwal, "Systemverilog assertion based verification of amba-ahb," in *2016 International Conference on Micro-Electronics and Telecommunication Engineering (ICMETE)*, pp. 641–645, IEEE, 2016.

[10] M. Handover and A. Ayari, "Using formal verification to check soc connectivity correctness," *Digital Verification Technology, Mentor Graphics*, 2011.