Automation of Scalable Verification Environment

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

Kishan Raithatha

(12MECC23)

Electronics and Communication Engineering Branch

Electrical Engineering Department

Institute of Technology, Nirma University

Ahmedabad-382 481

May 2014

Automation of Scalable Verification Environment

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

Kishan Raithatha

(12MECC23)

Under the Guidance of

Dr. D.K.Kothari

Electronics and Communication Engineering Branch

Electrical Engineering Department

Institute of Technology, Nirma University

Ahmedabad-382 481

May 2014

Declaration

This is to certify that

I) The thesis comprises my original work towards the degree of Master of Technology in Communication Engineering at Nirma University and has not been submitted elsewhere for Degree.

II) Due Acknowledgment has been made in the text to all other material used.

Kishan Raithatha

(12MECC23)

CERTIFICATE

This is to certify that the Major Project entitled **Automation of Scalable Verification Environment** submitted by **Kishan Raithatha(12MECC23)**, towards the partial fulfillment of the requirements for the degree of Master of Technology in Communication Engineering of Nirma University, Ahmedabad is the record of work carried out by him under my supervision and guidance. In my 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 my knowledge, have not been submitted to any other university or institution for award of any degree or diploma.

Date: Place: Ahmedabad

Internal Guide

Dr. D.K.Kothari
(Professor, EC)

Program Coordinator

Dr. D.K.Kothari
(Professor, EC)

Dr. P N Tekwani
Head of Electrical Engineering Department

Dr. K Kotecha
Director, Institute of Technology
Nirma University

ACKNOWLEDGEMENTS

I am indeed thankful to my Internel guide **Dr.D.K.Kothari**, Institute of Technology, Nirma University, Ahmedabad for his valuable guidance and encouragement and extensive support provided during the thesis preparation. I am also highly thankful to **Mr. Kishor Kulkarni** , Sr. Tech Lead ,Intel Technology India Pvt. Ltd. , for assigning an interesting problem statement and giving valuable guidance throughout the project. I am also grateful to **Mr. Chandra Prakash**, Design Verification Engineer, Intel TechnologyIndia Pvt. Ltd, who has always been there to resolve my queries and also teaching me many new concepts which I wasnt aware about. Both of them have been very helpful during the entire project period and have always been a great source of encouragement.

I would also like to thank **Dr. K Kotecha**, Director, Institute of Technology, Nirma University, Ahmedabad, **Dr. P N Tekwani** Head of Electrical Engineering Department for provding an oppurtunity for pursuing internship at Intel Technology India Pvt. Ltd.

I would like to thank my Manager, **Mr. Manoj Velayudha**, Intel Technology India Pvt. Limited, Bangalore and **Mr. Krishnan Subramoniam** for their invaluable support.

I am also thankful to all my faculty members for preparing my foundation based on which I am able to perform many tasks today.

Last but not the least I would like to thank my family members for the emotional support they provided on the basis of which I was able to carry out the project work and also to GOD who has provided me the strength for overcoming the obstacles faced during the project work.

Kishan Raithatha(12MECC23)

Abstract

Verification plays a major role in any SOC development. But however with increasing complexity of the SOCs , the process of verification is turning out to be the major bottleneck in any device development. More than 50% of the overall time spent for the project is consumed in verification and the scenario is getting worse day by day . There is a need to speed up the verification process.

The project assigned targeted the scalable verification environment of IA based hardware accelerators. The assigned SOC had various scalable entities, the verification environment of which was first studied and was made scalable. The remaining automating tasks had been performed using scripts resulting into faster testbench development for any scaled version of the assigned SOC.

The user only needs to give inputs corresponding to the no. of various scalable components in the SOC from the GUI and the memory map associated with the slave peripherals. The backend scripts would then scale up the existing verification environment hence saving the overall time spent for verification.The project implemented has been tested for different scalable versions of the RTL and is working fine.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The complexity of SOC devices are increasing day by day, along with them the time spent in verification is also increasing exponentially. According to recent statistics the mean time spent in verification of the total project time is more than 50%. As shown in the figure 1.1 the mean time spent in verification in the year 2007 was around 46 % which got increased to 51% in 2010 and to 53% in 2012 . Owing to this there is a very high need to speed up the verification process. The following figure taken from [6] depicts the same.

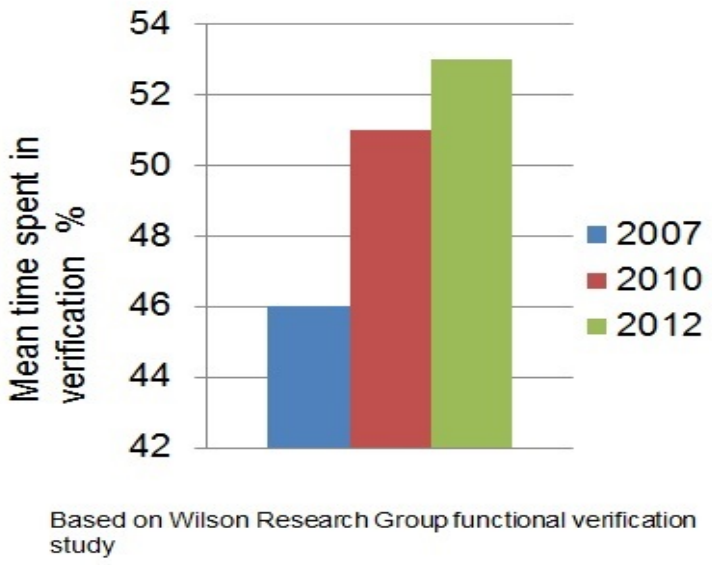


Figure 1.1: Mean Time spent in Verification

1.2 The Verification Process

The entire verification process can be sub-divided as shown in figure 1.2 which is taken from [7].

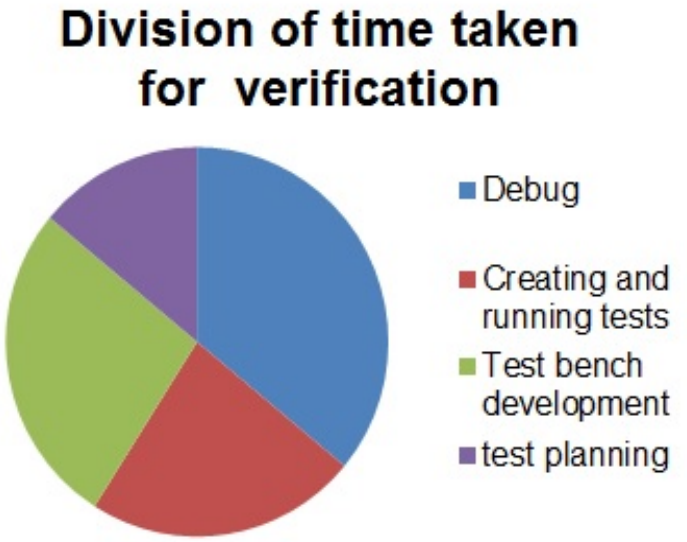


Figure 1.2: Verification process division

As depicted in the figure 1.2 the major time spent in verification is in functional debugs, the next portion of time is taken by the test bench development the next is consumed by creating and running test cases and the remaining is consumed from test planning.

A similar analogy can also be applied to developing a verification environment of a scaled up verision. If some of these tasks can be automated the device can be verified faster. This is the main goal of project to automate as many tasks as possible and thereby speeding up the verification process.

1.3 Targeted SOC

The scalable verification environment associated with intel processor based hardware accelerators has been targeted. An outline of the architecture is provided in the figure 1.3.

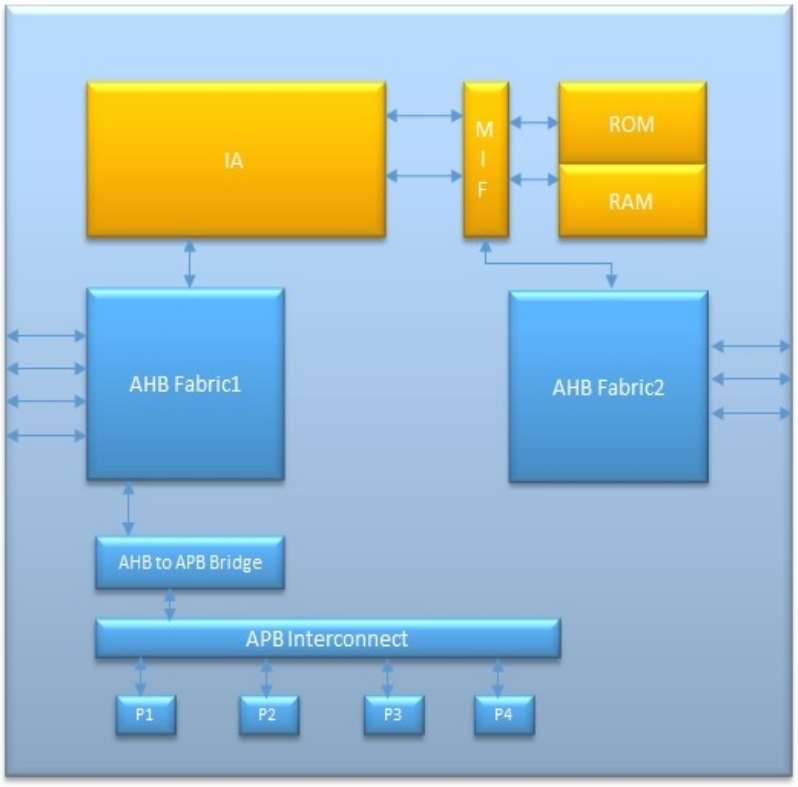


Figure 1.3: The SOC used for implementation

The SOC targeted has an Intel processor along with two AHB fabrics, an APB fabric, various peripherals and system memory. There are many scalable components such as external ports associated with AHB fabric 1 and 2, peripherals associated with the APB fabric and so on. The verification Environment associated with the SOC was first studied and made scalable. The process of generating verification environment for different scalable versions is fully automated using scripts made in PERL.

1.4 Organization of the Thesis

The thesis is organized as follows:

Chapter 2 describes the Universal Verification Methodology(UVM) which is the used

methodology for the assigned project for coding the verification IPs. The need and vaious features of UVM are discussed.

The verification environment once coded is compiled and simulated using the tool VCS(Verilog Compiler and Simulator) and for carrying out the debugs one of the tools which is a part of VCS is DVE(Discovery Visual Environment). For generating various RTL configurations based on the AMBA subsystem another tool CoreAssembler can be used. **Chapter 3** has a description regarding the same. These tools have been extensively used throughout the project work. Also the chapter describes the various languages used for designing the utility such as system verilog which is used for coding the verification environment and Perl which is used for developing scripts for automating the tasks.

The SoC used has a couple of busses belonging to AMBA sub-system such as AHB Lite and APB. An extensive knowledge of these busses such as their signal description, the type of transfers supported need to be known for carrying out the debugs when errors occur. **Chapter 4** gives an overview of all the busses that are a part of AMBA sub-system, and more significance is given to AHB Lite which is a part of the assigned SoC.

Chapter 5 gives the actual implementation details of the work carried out, the approach followed and the results obtained. It gives an overview of the way the existing code is first made scalable and what role the Perl scripts are making and other details of the GUI used.

The scalablity in addition to number of AHB Fabric masters and slaves and APB peripherals can also be targeted pertaining to data width of AHB fabric 2 and RAM size.**Chapter 6** describes the way the verification environment is made scalable w.r.t data width of AHB fabric 2 - an option to select between the data width of 32 bit/ 64 bit and also how support is provided for a scalable RAM size.

Once the verification environment is scaled the test regression suite cant be run straight away. Few of the files specifying the order of compilation and elaboration need to be modified. **Chapter 7** gives a description regarding the additional script that automated the compilation and elaboration process resulting into an additional 2 hours saving in time spent for verification.

Chapter 8 has the final conclusion regarding the outcome of the project and the future scope for implementation.

1.5 Summary

The chapter provided an overview of the project. The need for the utility which would automate the verification environment for a scalable design is discussed. An overview of the architecture of the SoC is also provided. The last section describes about the organization of the thesis, a short description of various chapters in the thesis is provided.

Chapter 2

UVM : The Universal Verification Methodology

2.1 Intoduction

UVM - Universal Verification Methodology is a methodology for functional verification using SystemVerilog, complete with a supporting library of SystemVerilog code. UVM was created by Accellera based on the OVM (Open Verification Methodology) version 2.1.1. The roots of these methodologies lie in the application of the languages IEEE 1800 SystemVerilog, IEEE 1666 SystemC, and IEEE 1647 e.

UVM is a methodology for the functional verification of digital hardware, primarily using simulation. The hardware or system to be verified would typically be described using Verilog, SystemVerilog, VHDL or SystemC at any appropriate abstraction level. This could be behavioral, register transfer level, or gate level. UVM is explicitly simulation-oriented, but UVM can also be used alongside assertion-based verification, hardware acceleration or emulation.

When we run RTL simulations in Verilog or VHDL, we can think of UVM as replacing whatever framework and coding style we use for our test benches. But UVM test benches are more than traditional HDL test benches, which might wiggle a few pins on the design-under-test (DUT) and rely on the designer to inspect a waveform diagram to verify correct operation. UVM test benches are complete verification environments composed of reusable verification components, and used as part of an overarching methodology of constrained random, coverage-driven, verification.

2.2 UVM- Need and features

Simulation might be caricatured as the process of poking test vectors into a model of the design-under-test and observing how that model behaves. A traditional Verilog or VHDL test bench might contains processes to read raw vectors or commands from a file, use those to change the values of the wires connected to the DUT over time, and perhaps collect output from the DUT and dump it to another file. This is fine as far as it goes, but this process does not scale up well to support the reliable verification of very complex systems.

A good verification methodology starts with a statement of the function the DUT is intended to perform. From this is derived a verification plan, broken down feature-by-feature, and agreed in advance by all those with a specific interest in creating a working product. This verification plan is the basis for the whole verification process. Verification is only complete when every item on the plan has been tested to an acceptable level, where the meaning of "acceptable" and the priorities assigned to testing the various features have also been agreed in advance and are continually reviewed during the project.

Verification of complex systems should not be reliant on manual inspection of detailed waveforms and vector sets. Functional checking must be automated if the process is to scale well, as must the collection of verification metrics such as the coverage of features in the verification plan and the number of bugs found by each test. Along with the verification plan, automated checking and functional coverage collection and analysis are cornerstones of any good verification methodology, and are explicitly addressed by SystemVerilog and UVM. Checkers and a functional coverage model, linked back to the verification plan, take engineering time to create but result in much improved quality of verification.

All simulation-based verification suffers from the issue that you can never run enough test vectors to exhaustively test the whole design, or even any significant part of a complex design. One way to address this issue is using constrained random stimulus. The use of random stimulus brings two very significant benefits. Firstly, random stimulus is great for uncovering unexpected bugs, because given enough time and resources it can

allow the entire state space of the design to be explored free from the selective biases of a human test writer. Secondly, random stimulus allows compute resources to be maximally utilised by running parallel compute farms and overnight runs. Of course, pure random stimulus would be nonsensical, so adding constraints to make random stimulus legal is an important part of the verification process, and is explicitly supported by SystemVerilog and UVM.

A. **Checkers, Coverage and Constraints**

Constrained random verification relies on Checkers, Coverage and Constraints. Each of these "three C's" plays a key role in the verification process and is supported by explicit features of the SystemVerilog language. Firstly, checkers ensure functional correctness. Nothing is gained by throwing more and more random stimulus into a design to take functional coverage to ever higher levels unless the design-under-test is being checked automatically for functional correctness. Checkers can be implemented using SystemVerilog assertions or using regular procedural code. Assertions can be embedded within the design-under-test, placed on the external interfaces, or can be part of the verification environment. UVM provides mechanisms and guidelines for building checkers into the verification environment and for logging reports.

Secondly, coverage provides a measure of the functional completeness of the testing, and tells you when you've met the goals set out in the verification plan, and thus when you have finished simulating. SystemVerilog offers two separate mechanisms for functional coverage collection; property-based coverage (cover directives) and sample-based coverage (covergroups). Both can be used in a UVM verification environment. The specification and execution of the coverage model is intimately tied to the verification plan, and many simulation tools are able to annotate coverage information onto the verification plan document, facilitating tight management control.

Thirdly, constraints provide the means to reach coverage goals by shaping the random stimulus to push the design-under-test into interesting corner cases. Without shaping, random stimulus alone may be insufficient to exercise many of the deeper states of the design-under-test. Constrained random stimulus is still random, but

the statistical distribution of the vectors is shaped to ensure that interesting cases are reached. SystemVerilog has dedicated language features for expressing constraints, and UVM goes further by providing mechanisms that allow constraints to be written as part of a test rather then embedded within dedicated verification components. This and other features of UVM facilitate the creating of reusable verification components.

B. **Tests and Coverage**

The features enumerated in the verification plan should be captured as a set of coverage statements that together form an executable coverage model. With many simulation tools, the verification plan will include references to the corresponding coverage statements, and as simulation runs, coverage data is back-annotated from the simulator onto the verification plan feature-by-features. This provides direct feedback on the effectiveness of any given test. Holes in the coverage goals can be plugged by writing further tests. The verification plan itself is not part of UVM proper, but is a vital element in the verification process. UVM provides guidance on how to collect coverage data in a reusable manner.

With directed testing, tests are written with the purpose of pushing the design into specific states and exercising specific cases. With constrained random testing, the role of the tests shifts slightly. Although a constrained random test may be written with specific coverage goals in mind, it is not assumed before-the-fact that any particular test will actually test one feature rather than another. The constrained random test is run, and the coverage model is used to empirically measure which features the test did in fact exercise. Tests can be graded after-the-fact using the coverage data, and the most effective tests, that is those that achieve the highest coverage in the fewest number of cycles, can be used to form the basis of a regression test set.

C. **Engineering Effort**

With constrained random verification, the focus of the engineering effort changes from writing directed tests to building automated checkers and an executable cov-

erage model. Random stimulus then enables compute resources to be fully utilized in the pursuit of hitting coverage goals. The total number of man-hours dedicated to verification will not necessarily decrease, but verification quality will be dramatically improved, and the verification process will become far more transparent and predictable, both to the verification team itself and to outside observers. Automated coverage collection gives accurate feedback on the progress of the verification effort, and the emphasis on verification planning ensures that resources are focussed on achieving agreed goals.

D. **Verification Reuse**

UVM facilitates the construction of verification environments and tests, both by providing reusable machinery in the form of a library of SystemVerilog classes, and also by providing a set of guidelines for best practice when using SystemVerilog for verification. Verification productivity can be enhanced by reusing verification components, and this is an important objective of UVM. Verification reuse is enabled by having a modular verification environment where each component has clearly defined responsibilities, by allowing flexibility in the way in which components are configured and used, by having a mechanism to allow imported components to be customized to the application at hand, and by having well-defined coding guidelines to ensure consistency.

The architecture of UVM has been designed to encourage modular and layered verification environments, where verification components at all layers can be reused in different environments. Low-level driver and monitor components can be reused across multiple designs-under-test. The whole verification environment can be reused by multiple tests and configured top-down by those tests. Finally, test scenarios can be reused from application to application. This degree of reuse is enabled by having UVM verification components able to be configured in a very flexible way without modification to their source code. This flexibility is built into the UVM class library.

2.3 UVM Test Bench components

An UVM testbench is composed of reusable verification environments called verification components. A verification component is an encapsulated, ready-to-use, configurable verification environment for an interface protocol, a design submodule, or a full system. Each verification component follows a consistent architecture and consists of a complete set of elements for stimulating, checking, and collecting coverage information for a specific protocol or design. The verification component is applied to the device under test (DUT) to verify your implementation of the protocol or design architecture.
The following subsections describe the components of a verification component.

- **Data Item (Transaction)** Data items represent the input to the device under test (DUT). Examples include networking packets, bus transactions, and instructions. The fields and attributes of a data item are derived from the data items specification. For example, the Ethernet protocol specification defines valid values and attributes for an Ethernet data packet. In a typical test, many data items are generated and sent to the DUT. By intelligently randomizing data item fields using SystemVerilog constraints, you can create a large number of meaningful tests and maximize coverage.

- **Driver (BFM)** A driver is an active entity that emulates logic that drives the DUT. A typical driver repeatedly receives a data item and drives it to the DUT by sampling and driving the DUT signals. (If you have created a verification environment in the past, you probably have implemented driver functionality.) For example, a driver controls the read/write signal, address bus, and data bus for a number of clocks cycles to perform a write transfer.

- **Sequencer** A sequencer is an advanced stimulus generator that controls the items that are provided to the driver for execution. By default, a sequencer behaves similarly to a simple stimulus generator and returns a random data item upon request from the driver. This default behavior allows you to add constraints to the data item class in order to control the distribution of randomized values. Unlike generators that randomize arrays of transactions or one transaction at a time, a sequencer captures important randomization requirements out-ofthe- box. A partial

list of the sequencers built-in capabilities includes:

  - Ability to react to the current state of the DUT for every data item generated.
  - Captures the order between data items in user-defined sequences, which forms a more structured and meaningful stimulus pattern.
  - Enables time modeling in reusable scenarios.
  - Supports declarative and procedural constraints for the same scenario.
  - Allows system-level synchronization and control of multiple interfaces

- **Monitor** A monitor is a passive entity that samples DUT signals but does not drive them. Monitors collect coverage information and perform checking. Even though reusable drivers and sequencers drive bus traffic, they are not used for coverage and checking. Monitors are used instead. A monitor:

  - Collects transactions (data items). A monitor extracts signal information from a bus and translates the information into a transaction that can be made available to other components and to the test writer.
  - Extracts events. The monitor detects the availability of information (such as a transaction), structures the data, and emits an event to notify other components of the availability of the transaction. A monitor also captures status information so it is available to other components and to the test writer.
  - Performs checking and coverage. Checking typically consists of protocol and data checkers to verify that the DUT output meets the protocol specification. Coverage also is collected in the monitor.
  - Optionally prints trace information.

  A bus monitor handles all the signals and transactions on a bus, while an agent monitor handles only signals and transactions relevant to a specific agent. Typically, drivers and monitors are built as separate entities (even though they may use the same signals) so they can work independently of each other. However, you can reuse code that is common between a driver and a monitor to save time. Do not have monitors depend on drivers for information so that an agent can operate passively when only the monitor is present.

- **Agent**

  Sequencers, drivers, and monitors can be reused independently, but this requires the environment integrator to learn the names, roles, configuration, and hookup of each of these entities. To reduce the amount of work and knowledge required by the test writer, UVM recommends that environment developers create a more abstract container called an agent. Agents can emulate and verify DUT devices. They encapsulate a driver, sequencer, and monitor. Verification components can contain more than one agent. Some agents (for example, master or transmit agents) initiate transactions to the DUT, while other agents (slave or receive agents) react to transaction requests. Agents should be configurable so that they can be either active or passive. Active agents emulate devices and drive transactions according to test directives. Passive agents only monitor DUT activity.

- **Environment**

  The environment (env) is the top-level component of the verification component. It contains one or more agents, as well as other components such as a bus monitor. The env contains configuration properties that enable you to customize the topology and behavior and make it reusable. For example, active agents can be changed into passive agents when the verification environment is reused in system verification.

2.4 UVM Class Library

The UVM Class Library provides all the building blocks you need to quickly develop well-constructed, reusable, verification components and test environments (as shown in the below figure). The library consists of base classes, utilities, and macros. Components may be encapsulated and instantiated hierarchically and are controlled through an extendable set of phases to initialize, run and complete each test. These phases are defined in the base class library but can be extended to meet specific project needs. A pictorial description of UVM Class Heirarchy taken from [1] is depicted in the below figure.

The advantages of using the UVM Class Library include:

- A robust set of built-in features

  The UVM Class Library provides many features that are required for verification, including complete implementation of printing, copying, test phases, factory meth-

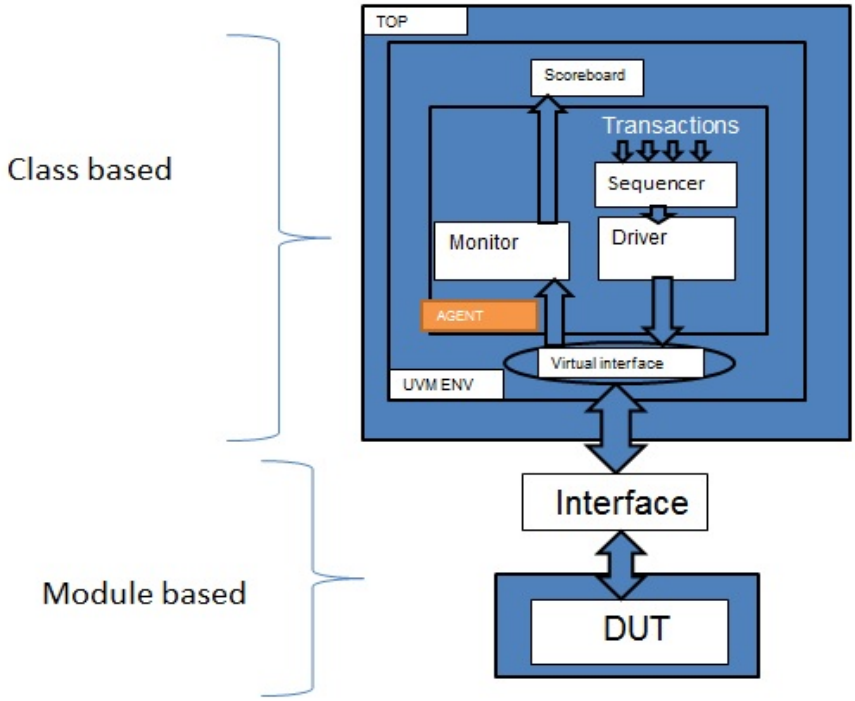


Figure 2.1: UVM Verification Environment

ods, and more.

- Correctly-implemented UVM concepts

  Each component in the block diagram in Figure 2.2 is derived from a corresponding UVM Class Library component. Figure 2.3 shows the same diagram using the derived UVM Class Library base classes. Using these base-class elements increases the readability of your code since each components role is predetermined by its parent class. A pictorial description of a typical UVM Environment which is taken from [1] is shown in the below figure.

2.5 Other UVM Facilities

The UVM Class Library also provides various utilities to simplify the development and use of verification environments. These utilities support debugging by providing a user-controllable messaging utility. They support development by providing a standard communication infrastructure between verification components (TLM) and flexible verification environment construction (UVM factory).

The UVM Class Library provides global messaging facilities that can be used for failure

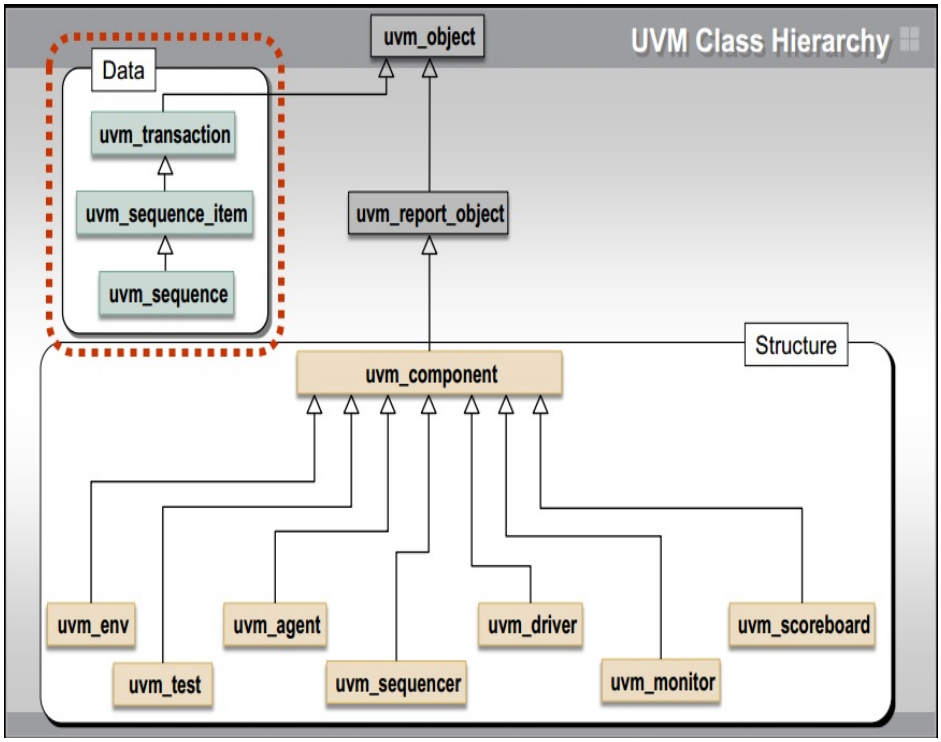


Figure 2.2: UVM Class Heirarchy

reporting and general reporting purposes. Both messages and reporting are important aspects of ease of use.

- **UVM Factory**

  The factory method is a classic software design pattern that is used to create generic code, deferring to run time the exact specification of the object that will be created. In functional verification, introducing class variations is frequently needed. For example, in many tests you might want to derive from the generic data item definition and add more constraints or fields to it; or you might want to use the new derived class in the entire environment or only in a single interface; or perhaps you must modify the way data is sent to the DUT by deriving a new driver. The factory allows you to substitute the verification component without having to provide a derived version of the parent component as well.

  The UVM Class Library provides a built-in central factory that allows:

  – Controlling object allocation in the entire environment or for specific objects.

  – Modifying stimulus data items as well as infrastructure components (for example, a driver).

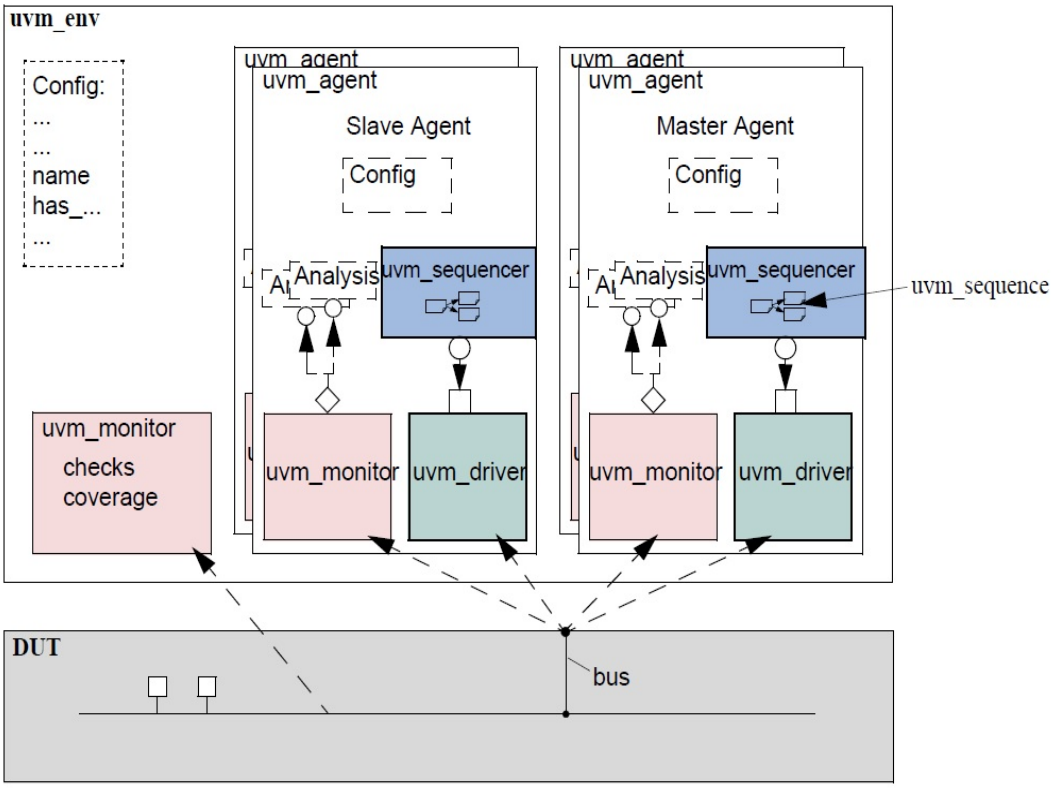


Figure 2.3: Typical UVM Environment using UVM Library Classes

Using the UVM built-in factory reduces the effort of creating an advanced factory or implementing factory methods in class definitions. It facilitates reuse and adjustment of predefined verification IP in the endusers environment. One of the biggest advantages of the factory is that it is transparent to the test writer and reduces the object-oriented expertise required from both developers and users.

- **Transaction-Level Modeling (TLM)**

  UVM components communicate via standard TLM interfaces, which improves reuse. Using a SystemVerilog implementation of TLM in UVM, a component may communicate via its interface to any other component that implements that interface. Each TLM interface consists of one or more methods used to transport data. TLM specifies the required behavior (semantic) of each method, but does not define their implementation. Classes inheriting a TLM interface must provide an implementation that meets the specified semantic. Thus, one component may be connected at the transaction level to others that are implemented at multiple levels of abstrac-

tion. The common semantics of TLM communication permit components to be swapped in and out without affecting the rest of the environment.

2.6 Summary

This chapter described about the Universal Verification Methodology (UVM) which is used in the project for coding the verification IPs. The chapter has a complete description of the methodology ranging from its need, to the various features of it.

Chapter 3

The AMBA sub-system

A set of AMBA on chip busses are a part of the SoC architecture. The main ones are AHB Lite and APB and multi layer AHB-Lite. The same are studied . The current chapter includes a brief overview of the AMBA busses.

3.1 Introduction

The Advanced Microcontroller Bus Architecture (AMBA) specification defines an onchip communications standard for designing high-performance embedded microcontrollers. Three distinct buses are defined within the AMBA specification:

- the Advanced High-performance Bus (AHB)
- the Advanced System Bus (ASB)
- the Advanced Peripheral Bus (APB).
- Advanced High-performance Bus (AHB) The AMBA AHB is for high-performance, high clock frequency system modules.The AHB acts as the high-performance system backbone bus. AHB supports the efficient connection of processors, on-chip memories and off-chip external memory interfaces with low-power peripheral macrocell functions. AHB is also specified to ensure ease of use in an efficient design flow using synthesis and automated test techniques.
- Advanced System Bus (ASB) The AMBA ASB is for high-performance system modules.AMBA ASB is an alternative system bus suitable for use where the high-performance features of AHB are not required. ASB also supports the efficient

connection of processors, on-chip memories and off-chip external memory interfaces with low-power peripheral macrocell functions.

- Advanced Peripheral Bus (APB) The AMBA APB is for low-power peripherals.AMBA APB is optimized for minimal power consumption and reduced interface complexity to support peripheral functions. APB can be used in conjunction with either version of the system bus.

3.2 The AMBA AHB

AHB is a new generation of AMBA bus which is intended to address the requirements of high-performance synthesizable designs. It is a high-performance system bus that supports multiple bus masters and provides high-bandwidth operation. AMBA AHB implements the features required for high-performance, high clock frequency systems including:

- burst transfers
- split transactions
- single-cycle bus master handover
- single-clock edge operation
- non-tristate implementation
- wider data bus configurations (64/128 bits).

Bridging between this higher level of bus and the current ASB/APB can be done efficiently to ensure that any existing designs can be easily integrated. An AMBA AHB design may contain one or more bus masters, typically a system would contain at least the processor and test interface. However, it would also be common for a Direct Memory Access (DMA) or Digital Signal Processor (DSP) to be included as bus masters. The external memory interface, APB bridge and any internal memory are the most common AHB slaves. Any other peripheral in the system could also be included as an AHB slave. However, low-bandwidth peripherals typically reside on the APB. A typical AMBA AHB system design contains the following components: **AHB master** A bus master is able to initiate read and write operations by providing an address and control information. Only one bus

master is allowed to actively use the bus at any one time. **AHB slave** A bus slave responds to a read or write operation within a given address-space range. The bus slave signals back to the active master the success, failure or waiting of the data transfer. **AHB arbiter** The bus arbiter ensures that only one bus master at a time is allowed to initiate data transfers. Even though the arbitration protocol is fixed, any arbitration algorithm, such as highest priority or fair access can be implemented depending on the application requirements.An AHB would include only one arbiter, although this would be trivial in single bus master systems. **AHB decoder** The AHB decoder is used to decode the address of each transfer and provide a select signal for the slave that is involved in the transfer.A single centralized decoder is required in all AHB implementations.

3.3 The AMBA ASB

Introducing the AMBA ASB ASB is the first generation of AMBA system bus. ASB sits above the current APB and implements the features required for high-performance systems including:

- burst transfers
- pipelined transfer operation
- multiple bus master.

A typical AMBA ASB system may contain one or more bus masters. For example, at least the processor and test interface. However, it would also be common for a Direct Memory Access (DMA) or Digital Signal Processor (DSP) to be included as bus masters. The external memory interface, APB bridge and any internal memory are the most common ASB slaves. Any other peripheral in the system could also be included as an ASB slave. However, low-bandwidth peripherals typically reside on the APB. An AMBA ASB system design typically contains the following components: **ASB master** A bus master is able to initiate read and write operations by providing an address and control information. Only one bus master is allowed to actively use the bus at any one time. **ASB slave** A bus slave responds to a read or write operation within a given address-space range. The bus slave signals back to the active master the success, failure or waiting of the data transfer. **ASB decoder** The bus decoder performs the decoding of the transfer addresses and selects slaves appropriately. The bus decoder also ensures that the bus

remains operational when no bus transfers are required. A single centralized decoder is required in all ASB implementations. **ASB arbiter** The bus arbiter ensures that only one bus master at a time is allowed to initiate data transfers. Even though the arbitration protocol is fixed, any arbitration algorithm, such as highest priority or fair access can be implemented depending on the application requirements. An ASB would include only one arbiter, although this would be trivial in single bus master systems.

3.4 The AMBA APB

The APB is part of the AMBA hierarchy of buses and is optimized for minimal power consumption and reduced interface complexity. The AMBA APB appears as a local secondary bus that is encapsulated as a single AHB or ASB slave device. APB provides a low-power extension to the system bus which builds on AHB or ASB signals directly. The APB bridge appears as a slave module which handles the bus handshake and control signal retiming on behalf of the local peripheral bus. By defining the APB interface from the starting point of the system bus, the benefits of the system diagnostics and test methodology can be exploited. The AMBA APB should be used to interface to any peripherals which are low bandwidth and do not require the high performance of a pipelined bus interface. The latest revision of the APB is specified so that all signal transitions are only related to the rising edge of the clock. This improvement ensures the APB peripherals can be integrated easily into any design flow, with the following advantages:

- high-frequency operation easier to achieve
- performance is independent of the mark-space ratio of the clock
- static timing analysis is simplified by the use of a single clock edge
- no special considerations are required for automatic test insertion
- many Application Specific Integrated Circuit (ASIC) libraries have a better selection of rising edge registers
- easy integration with cycle-based simulators.

These changes to the APB also make it simpler to interface it to the new AHB. An AMBA APB implementation typically contains a single APB bridge which is required to

convert AHB or ASB transfers into a suitable format for the slave devices on the APB. The bridge provides latching of all address, data and control signals, as well as providing a second level of decoding to generate slave select signals for the APB peripherals. All other modules on the APB are APB slaves. The APB slaves have the following interface specification:

- address and control valid throughout the access (unpipelined) zero-power interface during non-peripheral bus activity (peripheral bus is static when not in use)
- timing can be provided by decode with strobe timing (unclocked interface)
- write data valid for the whole access (allowing glitch-free transparent latch implementations).

3.5 Choice of the right bus

A full AHB or ASB interface is used for:

- bus masters item on-chip memory blocks
- external memory interfaces
- high-bandwidth peripherals with FIFO interfaces
- DMA slave peripherals.

A simple APB interface is recommended for:

- simple register-mapped slave devices
- very low power interfaces where clocks cannot be globally routed
- grouping narrow-bus peripherals to avoid loading the system bus.

3.6 AHB Lite

The AHB Lite bus was a part of the targeted SoC hence an in details description has been included of the same. AMBA AHB-Lite addresses the requirements of high-performance synthesizable designs. It is a bus interface that supports a single bus master and provides high-bandwidth operation.A pictorial description of same which is taken from [8] is shown AHB-Lite implements the features required for high-performance, high clock frequency systems including:

- burst transfers
- single-clock edge operation
- non-tristate implementation
- wide data bus configurations, 64, 128, 256, 512, and 1024 bits.

The most common AHB-Lite slaves are internal memory devices, external memory interfaces, and high bandwidth peripherals. Although low-bandwidth peripherals can be included as AHB-Lite slaves, for system performance reasons they typically reside on the AMBA Advanced Peripheral Bus (APB). Bridging between this higher level of bus and APB is done using a AHB-Lite slave, known as an APB bridge. Figure shows a single master AHB-Lite system design with one AHB-Lite master and three AHB-Lite slaves. The bus interconnect logic consists of one address decoder and a slave-to-master multiplexor. The decoder monitors the address from the master so that the appropriate slave is selected and the multiplexor routes the corresponding slave output data back to the master.

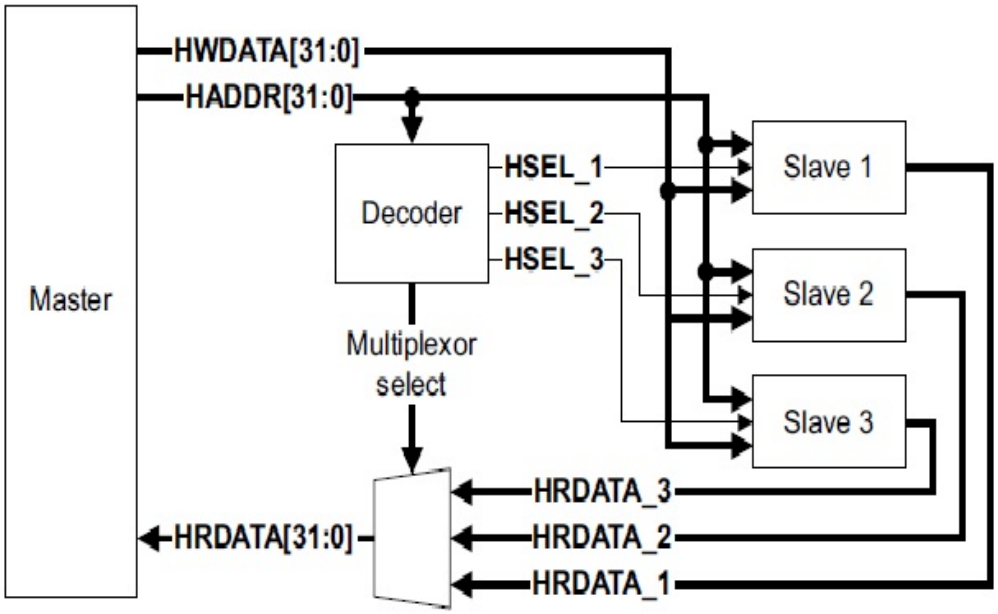


Figure 3.1: AHB Lite Block Diagram

3.6.1 Signal descriptions

The various signals along with their description have been listed in the following tables which are taken from [8].

Name	Source	Description
HCLK	Clock source	The bus clock times all bus transfers. All signal timings are related to the rising edge of HCLK
HRESETn	Reset controller	The bus reset signal is active LOW and resets the system and the bus. This is the only active low AHB-Lite signal.

Table 3.1: Global signals

Name	Source	Description
HADDR[31:0]	Slave and decoder	The 32-bit system address bus.
HBURST[2:0]	Slave	The burst type indicates if the transfer is a single transfer or forms part of a burst. Fixed length bursts of 4, 8, and 16 beats are supported. The burst can be incrementing or wrapping. Incrementing bursts of undefined length are also supported.
HMASTLOCK	Slave	When HIGH, this signal indicates that the current transfer is part of a locked sequence. It has the same timing as the address and control signals.
HPROT[3:0]	Slave	The protection control signals provide additional information about a bus access and are primarily intended for use by any module that wants to implement some level of protection. The signals indicate if the transfer is an opcode fetch or data access, and if the transfer is a privileged mode access or user mode access. For masters with a memory management unit these signals also indicate whether the current access is cacheable or bufferable.
HSIZE[2:0]	Slave	Indicates the size of the transfer, that is typically byte, halfword, or word. The protocol allows for larger transfer sizes up to a maximum of 1024 bits.
HTRANS[1:0]	Slave	Indicates the transfer type of the current transfer.
HWDATA[31:0]	Slave	The write data bus transfers data from the master to the slaves during write operations. A minimum data bus width of 32 bits is recommended. However,this can be extended to enable higher bandwidth operation.
HWRITE	Slave	Indicates the transfer direction. When HIGH this signal indicates a write transfer and when low indicates a read. It must remain constant throughout a burst transfer.

Table 3.2: Master Signals

3.6.2 Transfers

An AHB-Lite transfer consists of two phases: Address Lasts for a single HCLK cycle unless its extended by the previous bus transfer. Data That might require several HCLK cycles. Use the HREADY signal to control the number of clock cycles required to complete the transfer. HWRITE controls the direction of data transfer to or from the master. Therefore, when:

- HWRITE is HIGH, it indicates a write transfer and the master broadcasts data on

Name	Source	Description
HRDATA[31:0]	Multiplexor	During read operations,the read data bus transfers data from the selected slave to the multiplexor. The multiplexor then transfers the data to the master. A minimum data bus width of 32 bits is recommended. However this can be extended to enable higher bandwidth operation.
HREADYOUT	Multiplexor	When HIGH, the HREADYOUT signal indicates that a transfer has finished on the bus . The signal can be driven LOW to extend a transfer.
HRESP	Multiplexor	The transfer response, after passing through the multiplexor, provides the master with additional information on the status of a transfer. When LOW, the HRESP signal indicates that the transfer status is OKAY. When HIGH , the HRESP signal indicates that the transfer status is ERROR.

Table 3.3: Slave Signals

Name	Source	Description
HSELx	Slave	Each AHB-Lite slave has its own slave select signal HSELx and this signal indicates that the current transfer is intended for the selected slave. When the slave is initially selected it must also monitor the status of HREADY to ensure that the previous bus transfer has completed before it responds to the current transfer.The HSELx signal is a combinatorial decode of the address bus.

Table 3.4: Decoder Signals

Name	Source	Description
HRDATA[31:0]	Master	Read data bus, selected by the decoder
HREADY	Master and slave	When HIGH, the HREADY signal indicates to the master and all slaves, that the previous transfer is complete.
HRESP	Master	Transfer response, selected by the decoder

Table 3.5: Multiplexor Signals

the write data bus, HWDATA[31:0]

- HWRITE is LOW, a read transfer is performed and the slave must generate the data on the read data bus, HRDATA[31:0].

The simplest transfer is one with no wait states, so the transfer consists of one address cycle and one data cycle. Following figures show a simple read transfer and a simple write transfer. In a simple transfer with no wait states:

- The master drives the address and control signals onto the bus after the rising edge

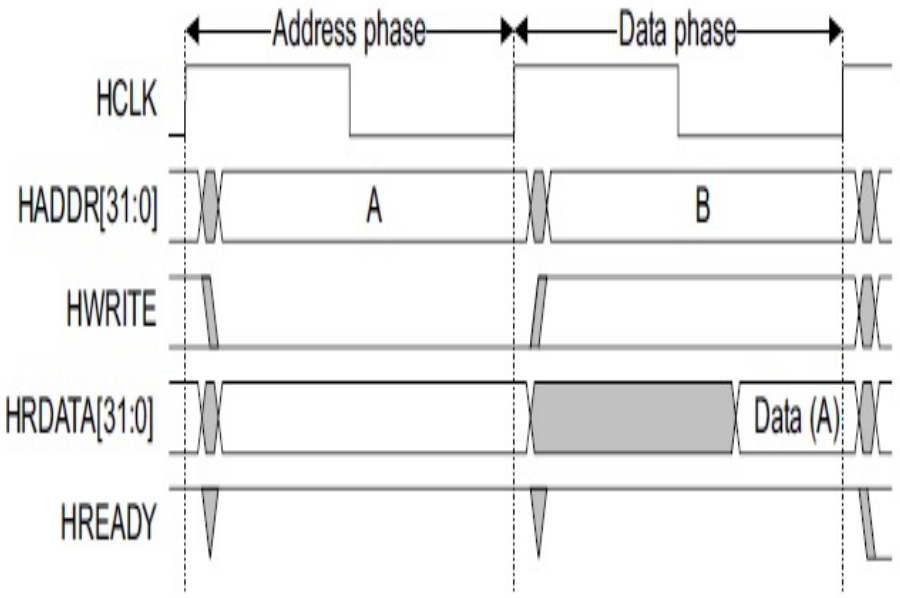


Figure 3.2: Read Transfer

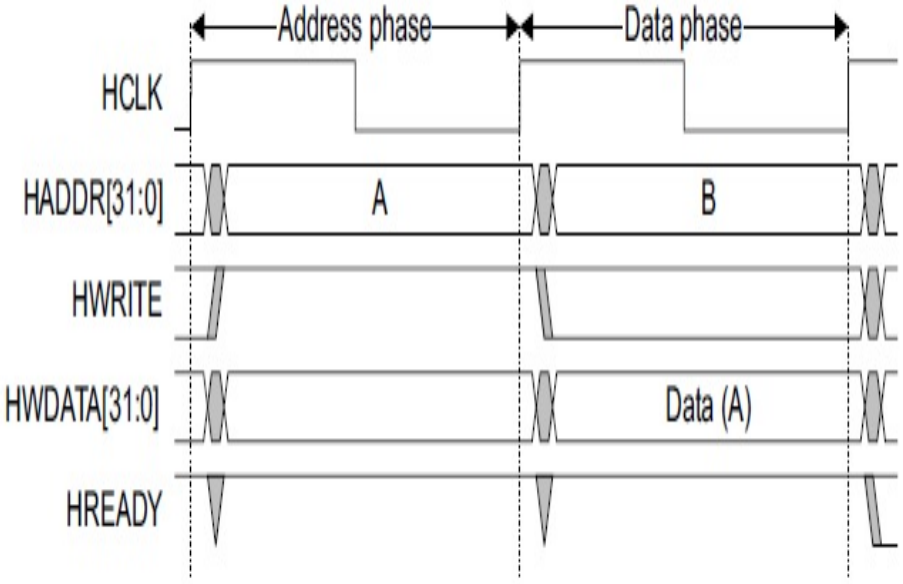


Figure 3.3: Write transfer

of HCLK.

- The slave then samples the address and control information on the next rising edge of HCLK.

- After the slave has sampled the address and control it can start to drive the appropriate HREADY response. This response is sampled by the master on the third rising edge of HCLK.

This simple example demonstrates how the address and data phases of the transfer occur during different clock cycles. The address phase of any transfer occurs during the data phase of the previous transfer. This overlapping of address and data is fundamental to the pipelined nature of the bus and enables high performance operation while still providing adequate time for a slave to provide the response to a transfer. A slave can insert wait states into any transfer to enable additional time for completion. A pictorial description of read transfer with wait states which is taken from [8] is shown in the following figure 3.4 and a write transfer with wait states also taken from [8] is reproduced in figure 3.5.

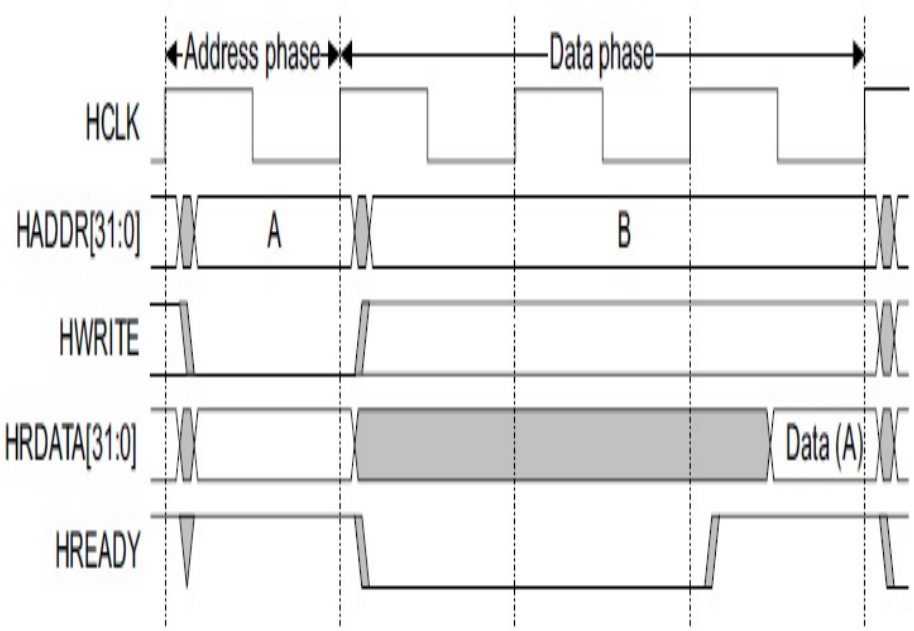


Figure 3.4: Read Transfer with Wait states

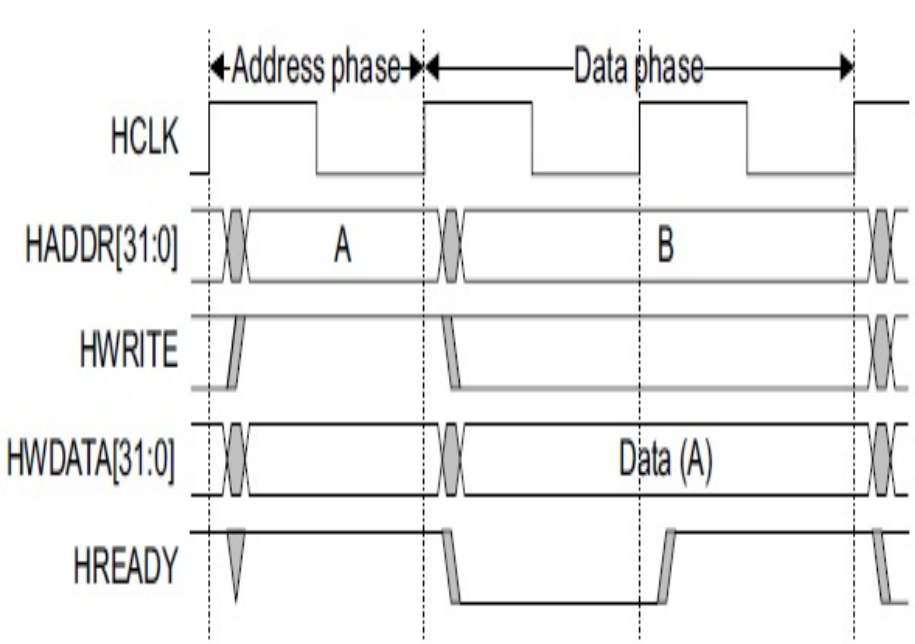


Figure 3.5: Write transfer with Wait states

3.7 MultiLayer AHB Lite

Multi-layer AHB is an interconnection scheme based on the AHB protocol that allows for parallel access paths between multiple masters and slaves in a system. This is achieved by using a more complex interconnection matrix and gives the benefit of increased overall bus bandwidth as well as a more flexible choice of system architecture. A key advantage of Multi-layer AHB is that standard AHB master and slave modules may be used without the need for modification. In situations where the system bottleneck is the result of limited bandwidth across the system bus, Multi-layer AHB solves the issue by multiplying the available bandwidth in proportion to the number of bus layers. Additional benefits arise from the reduction in bus transaction latency as a result of the increased bus capacity. Using Multi-layer AHB, a wide variety of bus structures can be created. With a single layer the structure is identical to the conventional AHB bus structure. Full Multi-layer AHB consists of a bus layer for each of the bus masters, with each layer connected to every slave through the slave multiplexor. Typical systems are more likely to fit between these structures with slaves connected to a sub-set of the layers, or multiple bus masters on a single layer. A simple block diagram of multi layer AHB fabric which is taken from [9] is shown in figure 3.6. The Key Benifits of the Multi Layer AHB are as follows:

- Complex multi-master systems can be constructed which have a flexible architecture. This removes the need to fix design decisions about the allocation of system resources to particular masters at the hardware design stage.
- Each AHB layer can be very simple if it only has one master, so no arbitration or master-to-slave muxing is required. These layers can use the AHB-Lite protocol, described below.
- Arbitration effectively becomes point arbitration at each peripheral, and is only necessary when more than one master wants to access the same slave simultaneously.
- The only hardware that needs to be added to the standard AHB transport infrastructure is the multiplexor block to connect the multiple masters to the peripherals.
- Previously designed masters and slaves can be reused without modification because the multi-layer architecture is compatible with the existing AHB protocol.

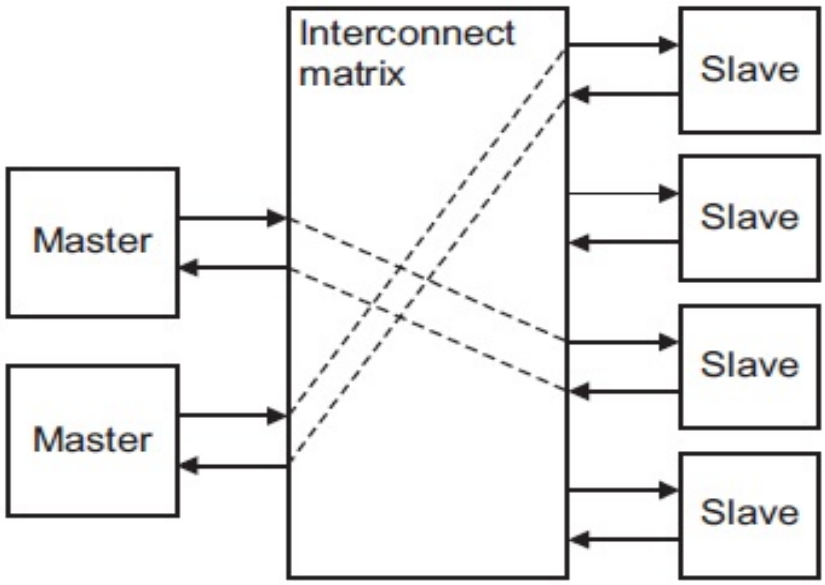


Figure 3.6: Multi Layer AHB Block Diagram

Implementation In the simplest implementation of a multi-layer system, each master has its own AHB layer and is connected to the slave devices by an interconnect matrix. The implementation details can be shown as in figure 3.7. The figure has been taken from [9]. Within the interconnect matrix:

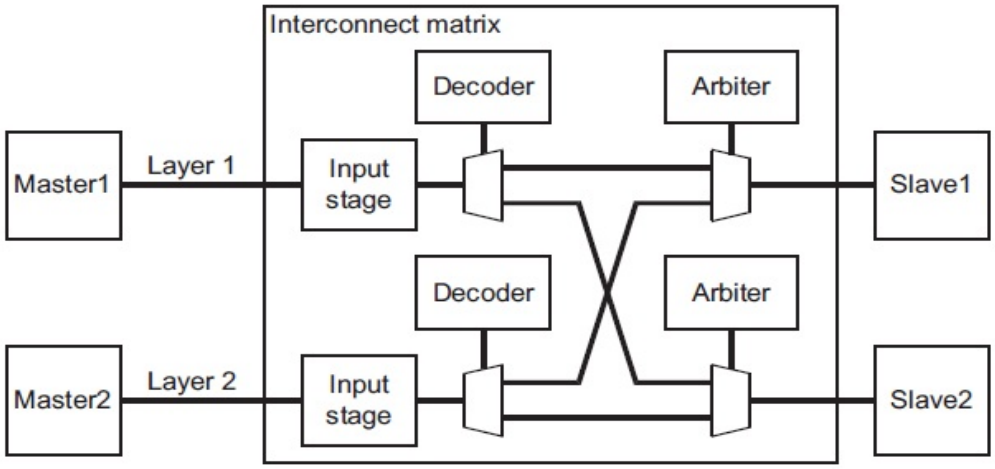


Figure 3.7: Multi Layer AHB Implementation Details

- Every layer has a Decode stage that determines which slave is required for a transfer.
- A mux routes the transfer from the appropriate layer to the required slave.

If two layers require access to the same slave at the same time, the arbitration within the interconnect matrix must determine which layer has highest priority. The layer that is not given access is waited using HREADY until it is given access to the required slave. When a layer is waited an Input Stage is used to store a copy of the pipelined address and control information until the access to the shared slave is given. Each slave port has its own arbitration and a number of different schemes can be used. For example:

- input layers can be serviced in round-robin manner, changing every transfer or every burst
- the arbitration can use a fixed priority scheme where certain high priority layers are always given access in preference to lower priority layers. The number of input/output ports on the interconnect matrix is completely flexible and can be adapted to suit the system requirements.

3.8 Summary

The chapter has a a brief description of the busses which come under the AMBA subsystem. A brief overview is provided about the AHB, ASB and APB busses. A more

detailed description is provided about the AHB Lite and Mullti Layer AHB Lite which are a part of the targeted SoC.

Chapter 4

Tools and Languages used

4.1 Tools

A set of EDA tools are used for the project. For carrying out the compilation and simulation the tool used is VCS (Verilog Compiler and simulator) from Synopsys and for generating the various configurations of RTL the tool used is coreAssembler which is another tool available from the same vendor Synopsys. The following sections give a brief overview of the tools.

4.1.1 VCS- Verilog Compiler and Simulator

Verilog simulators are software packages that emulate the Verilog hardware description language. Verilog simulation software has come a long way since its early origin as a single proprietary product offered by one company. Today, Verilog simulators are available from many vendors, at all price points. The suites bundle the simulator engine with a complete development environment: text editor, waveform viewer, and RTL-level browser.
Synopsys Verilog Compiler Simulator is a tool from Synopsys specifically designed to simulate and debug designs. VCS also uses VirSim, which is a graphical user interface to VCS used for debugging and viewing the waveforms.

There are three main steps in debugging the design, which are as follows

- Compiling the VerilogVHDL source code.
- Running the Simulation.
- Viewing and debugging the generated waveforms.

You can interactively do the above steps using the VCS tool. VCS first compiles the verilog source code into object files, which are nothing but C source files. VCS can compile the source code into the object files without generating assembly language files.VCS then invokes a C compiler to create an executable file. We use this executable file to simulate the design. You can use the command line to execute the binary file which creates the waveform file, or you can use VirSim.
Apart from this VCS offers industry-leading performance and capacity, complemented by a complete collection of advanced methodology-aware testbench and constraint debug features, bug-finding, coverage, planning and assertion technologies. VCS multicore technology delivers a 2x verification speed-up and cuts down verification time by running the design, testbench, assertions, coverage and debug in parallel on machines with multiple cores . VCS Partition Compile flow allows users to achieve up to 10 times faster compile turnaround time by only recompiling code that has changed. VCS also supplies a comprehensive suite of diagnostic tools, including simulation memory and time profiling, interactive constraint debugging, smart logging, and more to help users quickly analyze issues. VCS with native low power simulation and UPF support, delivers innovative voltage-aware verification techniques to find bugs in modern low power designs with integrated debug and high performance. With its built-in debug and visualization environment; support for all popular design and verification languages, including Verilog, VHDL, SystemVerilog, OpenVera, and SystemC; and the VMM, OVM, and UVM methodologies, VCS helps users develop high-quality designs.

- **High-Performance, Full-Featured, Native Testbench and Industry-Leading Systemverilog Support**

  VCS Native Testbench (NTB) technology provides built-in natively-compiled support for full-featured SystemVerilog and OpenVera testbenches, including object-oriented, constrained-random stimulus and functional coverage capabilities. VCS industry-leading, high-performance constraint solver technology is powered by multiple solver engines that simultaneously analyze all user specified constraints to rapidly generate high-quality random stimulus that verifies corner case behavior. The constraint solver engines will find a solution to user constraints, if one exists, minimizing constraint conflicts and maximizing verification productivity.

VCS further expands its capabilities with Echo constraint expression convergence technology. Echo automatically generates stimuli to efficiently cover the testbench constraint space, significantly reducing the manual effort needed to verify large numbers of functional scenarios. Echo is a perfect fit for all teams using SystemVerilog testbenches with random constraints.

VCS also provides a rich set of engines for reducing compile turnaround time and runtime, including pre-compiled IP support targeted at IP integration, Partition Compile to isolate portions of the testbench that are not changing during development cycles, dynamic reconfiguration to compile for a target and select which model is used at runtime, and save and restore functionality to save common states and apply them to subsequent runs reducing simulation time. Combined, these tools offer the most comprehensive set of solutions to maximize simulation efficiency and reduce turnaround time.

- **Comprehensive Coverage**

  VCS provides high-performance, built-in coverage technology to measure verification completeness. Comprehensive coverage includes code coverage, functional coverage and assertion coverage as well as user-defined metrics. Unified coverage aggregates all aspects of coverage in a common database, thereby allowing powerful queries and useful unified report generation. The unified coverage database offers 2x to 5x improvement in merge times and up to 2x reduction in disk space usage, which is critical for large regression environments

- **Complete Assertion Technologies**

  The native assertion technology in VCS enables an efficient methodology for deploying design-for-verification (DVF) techniques. The built-in support of SystemVerilog and OpenVera assertions allows designers to easily adopt DFV and find more bugs quickly. A rich assertion-checker library and a unique library of Assertion IP make it even easier to deploy assertions across teams and improve verification quality. The assertions serve both simulation and formal property verification environments.

- **Advanced Debugging and Visualization Environment DVE**

VCS includes the Discovery Visualization Environment (DVE), an advanced, full-featured debug and visualization environment . DVE has been specifically architected to work with all of the advanced bugfinding technology in VCS and shares a common look and feel with other Synopsys graphical-based analysis tools. DVE enables easy access to design and verification data along with an intuitive drag-and-drop or menu-andicon driven environment.

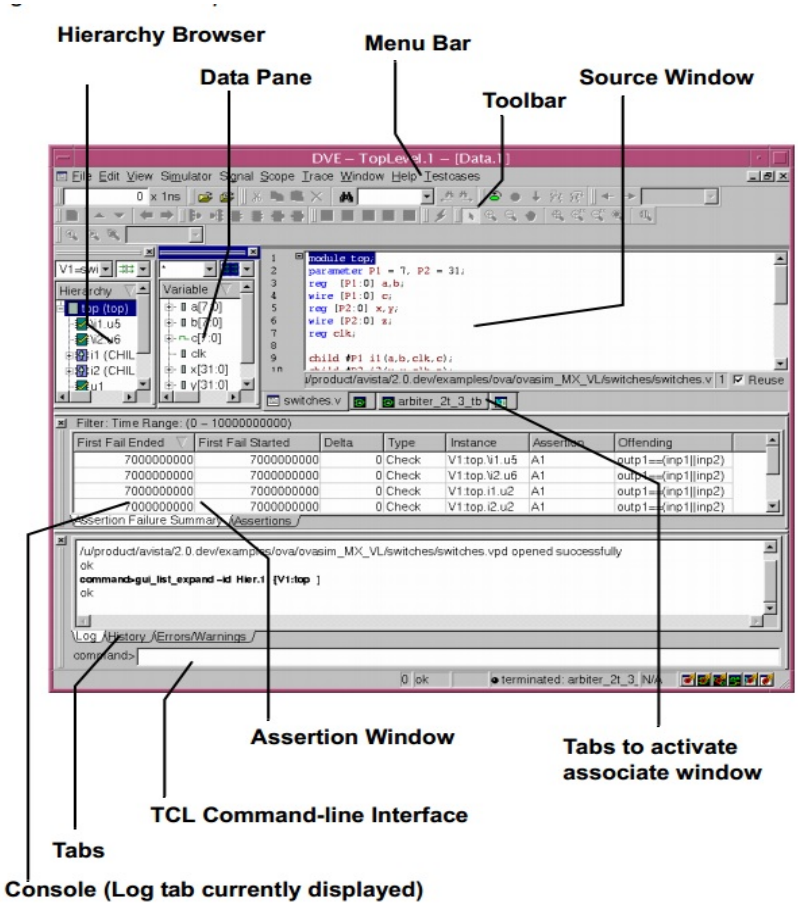


Figure 4.1: DVE top level window

Transaction-level debug is seamlessly integrated into DVE, allowing users to analyze and debug transactions in both list view and waveform view. Its debug capabilities include: tracing drivers, waveform compare, schematic views, path schematics, and support for the highly efficient Synopsys compact VCD+ binary dump format. It also provides elegant mixed-HDL (SystemVerilog, VHDL and Verilog) and SystemC/C++ language debugging windows, along with next-generation asser-

tion tracing capabilities, that help automate the manual tracing of relevant signals and sequences.DVE further provides powerful capabilities for SystemVerilog testbench debug (including UVM, VMM and UVM methodologies) with several key features, including methodologyaware debug panes, object and component hierarchy browsers, and detailed constraint debug and constraint conflict resolution .

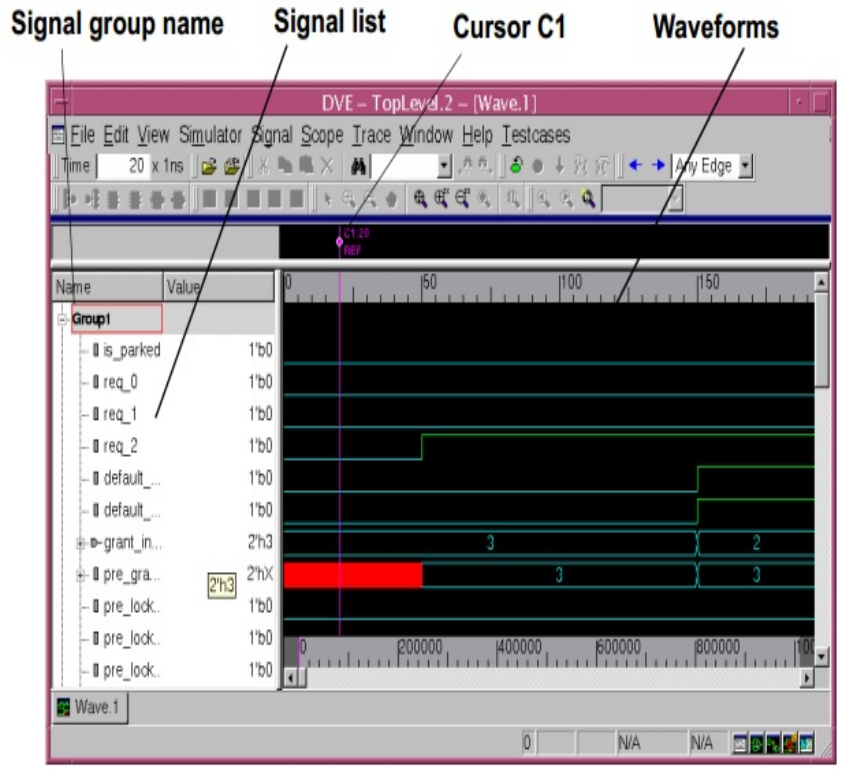


Figure 4.2: DVE waveform viewer

- **Support for Accellera UVM,VMM, and OVM**

  VCS powerful testbench engines are complemented by support for VMM, OVM 2.1.1, and the Accellera UVM methodologies. With these methodologies, users adopt industry best practices to get the optimum results from VCS. In addition, the VMM methodology provides a number of applications, such as Register Abstraction Layer (RAL) and others, to cut down on the time it takes to set up a powerful verification environment.

  VCS support for Accellera UVM also includes access to the VMM/UVM interoperability kit, which enables the use of VMM with UVM and vice versa. VCS

also provides a rich set of template generators for UVM, VMM, and the Register Abstraction Layer as well as wrappers for automatic TLM connectivity. All methodology applications, a detailed reference manual and examples are provided with the VCS solution.

4.1.2 Core Assembler

The various scaled up versions of the RTL have been obtained using the coreAssembler tool .The coreAssembler product is part of the complete set of IP reuse tools available from Synopsys. coreAssembler provides a graphical or command-based environment that guides the designer through the assembly and configuration of an IP-based subsystem. With coreAssembler, designers can easily generate the configured RTL of a subsystem based on the AMBA IP from the DesignWare Library or from IP with an interface that has been packaged for use with coreAssembler with coreBuilder or has an IP-XACT description of the IP to be integrated. IP that has not been packaged for re-use can also be directly imported into the subsystem. With coreAssembler, you can also easily create and package the complete IP-based subsystem for reuse.

- Intuitive graphical or command based environment
- Guides the IP integrator through the assembly of an IP-based subsystem
- Guides the IP integrator through the configuration of the components contained in the subsystem
- Generates the RTL configuration and interconnect logic
- Supports subsystem packaging
- Includes built-in interfaces to Synopsys tools including:
  - Design Compiler
  - Physical Compiler
  - Power Compiler
  - PrimeTime
  - Formality

  - VCS
  - TetraMAX
  - Synplify Premier

- Flexible TCL interface for tool customization
- Supports multi-language designs
- Supports the import of unpackaged IP
- Automatic generation of the IP-XACT XML representing the subsystem
- VMM or directed test-bench generation with DesignWare VIP
- Customizable activity list for design flow customizations

4.2 Languages

The languages learnt and used during project execution are:

- **System Verilog**
  System Verilog was used for coding the verification environment using the Universal Verification Methodology.
- **PERL**
  Automating tasks and preparation of GUI involved certain scripts usage which were made using PERL.

4.2.1 System Verilog

SystemVerilog is a Hardware Description and Verification Language based on Verilog. Although it has some features to assist with design, the thrust of the language is in verification of electronic designs. The bulk of the verification functionality is based on the OpenVera language donated by Synopsys. SystemVerilog has just become IEEE standard P1800-2005.SystemVerilog is an extension of Verilog-2001; all features of that language are available in SystemVerilog.
The following are some of the new features in System Verilog as compared to Verilog:

- **New data types**

  Multidimensional packed arrays unify and extend Verilog's notion of "registers" and "memories": Classical Verilog permitted only one dimension to be declared to the left of the variable name. SystemVerilog permits any number of such "packed" dimensions. A variable of packed array type maps 1:1 onto an integer arithmetic quantity. In the example above,each element of my_var may be used in expressions as a six-bit integer. The dimensions to the right of the name (32 in this case) are referred to as "unpacked" dimensions. As in Verilog-2001, any number of unpacked dimensions is permitted.

- **Enumerated data types** allow numeric quantities to be assigned meaningful names. Variables declared to be of enumerated type cannot be assigned to variables of a different enumerated type without casting. This is not true of parameters, which were the preferred implementation technique for enumerated quantities in Verilog-2001.

- **New Integer types** System Verilog defines byte, shortint, int and longint as two-state integral types having 8, 16 ,32 and 64 bits respectively. A bit type is a variable width two state type that works much like reg. Two -state types lack the X and Z metavalues of classical Verilog; working with these types may result in faster simulation.

- **Structures and unions** work much like they do in the C programming language. A SystemVerilog enhancement is the packed attribute, which causes the structure or union to be mapped 1:1 onto a packed array of bits.

- **Procedural blocks**

  In addition to Verilog's always block, SystemVerilog offers new procedural blocks that better convey the intended design structure. EDA tools can verify that the behavior described is really that which was intended. An always_comb block creates combinational logic. The simulator infers the sensitivity list from the contained statements.

  always_comb begin

tmp = b * b - 4 * a * c;

no_root = (tmp ¡ 0);

end

An always_ff block is meant to infer synchronous logic:

always_ff @(posedge clk)

count ¡= count + 1;

An always_latch block is meant to infer a level-sensitive latch. Again, the sensitivity list is inferred from the code:

always_latch

if (en) q ¡= d;

- **New data types** The string data type represents a variable-length text string. In addition to the static array used in design, SystemVerilog offers dynamic arrays, associative arrays and queues:

  int da[]; // dynamic array

  int da[string]; // associative array, indexed by string

  int da[$]; // queue

  initial begin

  da = new[16]; // Create 16 elements

  end

  A dynamic array works much like an unpacked array, but it must be dynamically created as shown above. The array can be resized if needed. An associative array can be thought of as a binary search tree with a user-specified key type and data type. The key implies an ordering; the elements of an associative array can be read out in lexicographic order. Finally, a queue provides much of the functionality of the C++ STL deque type: elements can be added and removed from either end efficiently. These primitives allow the creation of complex data structures required for scoreboarding a large design.

- **Classes** SystemVerilog provides an object-oriented programming model. SystemVerilog classes support a single-inheritance model. There is no facility that permits

conformance of a class to multiple functional interfaces, such as the interface feature of Java. SystemVerilog classes can be type-parameterized, providing the basic function of C++ templates. However, function templates and template specialization are not supported.

The polymorphism features are similar to those of C++: the programmer may specify write a virtual function to have a derived class gain control of the function. Encapsulation and data hiding is accomplished using the local and protected keywords, which must be applied to any item that is to be hidden. By default, all class properties are public.

SystemVerilog class instances are created with the new keyword. A constructor denoted by function new can be defined. SystemVerilog supports garbage collection, so there is no facility to explicitly destroy class instances.

- **Constrained random generation**

  Integer quantities, defined either in a class definition or as stand-alone variables in some lexical scope, can be assigned random values based on a set of constraints. This feature is useful for creating randomized scenarios for verification. Within class definitions, the rand and randc modifiers signal variables that are to undergo randomization. randc specifies permutation-based randomization, where a variable will take on all possible values once before any value is repeated. Variables without modifiers are not randomized.

- **Assertions** SystemVerilog has its own assertion specification language, similar to Property Specification Language. Assertions are useful for verifying properties of a design that manifest themselves over time. SystemVerilog assertions are built from sequences and properties. Properties are a superset of sequences; any sequence may be used as if it were a property, although this is not typically useful. Sequences consist of boolean expressions augmented with temporal operators. The simplest temporal operator is the ## operator which performs a concatenation

- **Coverage** Coverage as applied to hardware verification languages refers to the

collection of statistics based on sampling events within the simulation. Coverage is used to determine when the device under test (DUT) has been exposed to a sufficient variety of stimuli that there is a high confidence that the DUT is functioning correctly. Note that this differs from code coverage which instruments the design code to ensure that all lines of code in the design have been executed. Functional coverage ensures that all desired corner cases in the design space have been explored.

A SystemVerilog coverage group creates a database of "bins" that store a histogram of values of an associated variable. Cross coverage can also be defined, which creates a histogram representing the Cartesian cross-product of multiple variables. A sampling event controls when a sample is taken. The sampling event can be a Verilog event, the entry or exit of a block of code, or a call to the sample method of the coverage group. Care is required to ensure that data is sampled only when meaningful.

- **Synchronization** A complex test environment consists of reusable verification components that must communicate with one another. SystemVerilog offers two primitives for communication and synchronization: the mailbox and the mutex.The mutex is modeled as a counting semaphore. The mailbox is modeled as a FIFO .

4.2.2 PERL

Perl is an interpretd programming language .Most programming languages–such as C, C++, VisualBasic, etc.–are compiled languages. To run a program, you create a text document with the code, run a compiler on it to convert it into machine code for your OS, and then run it. Perl in an interpreted language, like Java, Pascal, awk, sed, Tcl, or Smalltalk. To run a program in such a language, you create a text document and tell the interpreter (or Virtual Machine) to run it as a program. The interpreter checks it, compiles it into machine code, and runs it. The 2-step process of interpreted languages makes them slightly easier to work with. You can constantly check your code by running it after every change.

Perl was designed in the mid 1980s by Larry Wall, then a programmer at Unisys. He

combined useful features of several existing languages with a syntax designed to sound as much as possible like English. Since then, Perl has mushroomed into a powerful and popular language, with lots of modules contributed by the open-source community. Perl is designed to be flexible, intuitive, easy, and fast; this makes it somewhat "messy". Perl has been called "a Swiss-Army chainsaw". Perl is also known as "the duct-tape of the Internet". At the most basic level, all windowing platforms (Apple Macintosh, X Windows, and Microsoft Windows) are very simple. They provide a low-level API to create and manage windows, to report interesting events such as mouse and keyboard events, and to draw graphical elements such as lines, circles, and bitmaps. The problem is that drawing even a simple form takes a considerable amount of code and reading thousands of pages of documentation (literally). As Perl is a much older language as compared to Python , Perl was preferred for the project. The way simple perl had been used in the project is explained in the subsequent chapters.

4.2.3 Building GUI using Perl Tk

Often-used patterns of GUI code have evolved into widgets (called "controls" in the Microsoft Windows world); examples include buttons, scrollbars, and listboxes. Building a GUI is now a simple matter of launching an interactive form designer and dragging and dropping these ready-made components into a layout of your choice. Object-oriented programming has never been easier. It turns out that widgets and scripting languages are a perfect match. Widgets have simple interfaces, and form-based GUIs are not performance-critical. Both of these attributes make GUIs a very fertile ground for scripting.Unlike other widget toolkits, Tk was developed expressly to be driven by a scripting language.

Perl/Tk (also known as pTk) is a collection of modules and code that attempts to wed the easily configured Tk 8 widget toolkit to the powerful lexigraphic, dynamic memory, I/O, and object-oriented capabilities of Perl 5. In other words, it is an interpreted scripting language for making widgets and programs with Graphical User Interfaces (GUI).

Tk, the extension(or module) that makes GUI programming in perl possible, is taken from Tcl/Tk. Tcl(Tool Command Language) and Tk(ToolKit) was created by Professor John Ousterhout of the University of California, Berkeley. Tcl is a scripting language that runs on Windows, UNIX and Macintosh platforms. Tk is a standard add-on to Tcl

that provides commands to quickly and easily create user interfaces. Later on Tk was used by a lot of other scripting languages like Perl, Python, Ruby etc.
There are many number of widgets available such as: Button ,Entry,Label,Frame,Text, Scrollbar,Scale, Grid,Radiobutton,Checkbutton,Listbox,Menubutton,Menu,OptionMenu, Canvas,Message,Adjuster,Scrolled etc . Each of the widget has got options to configure itself .
The main widgets used for making the GUI for the project are: Entry, label and button.

4.3 Summary

The chapter provided information about the various tools being used in the project. The main two EDA tools that are discussed are VCS(Verilog Compiler and Simulator) and CoreAssembler both from Synopsys. The chapter also has an overview of the languages used such as System Verilog which is used for coding the verification environment and Perl which is used for preparing scripts and Perl along with Tk package which can be used for preparing the gui.

Chapter 5

Implementation Details of Project

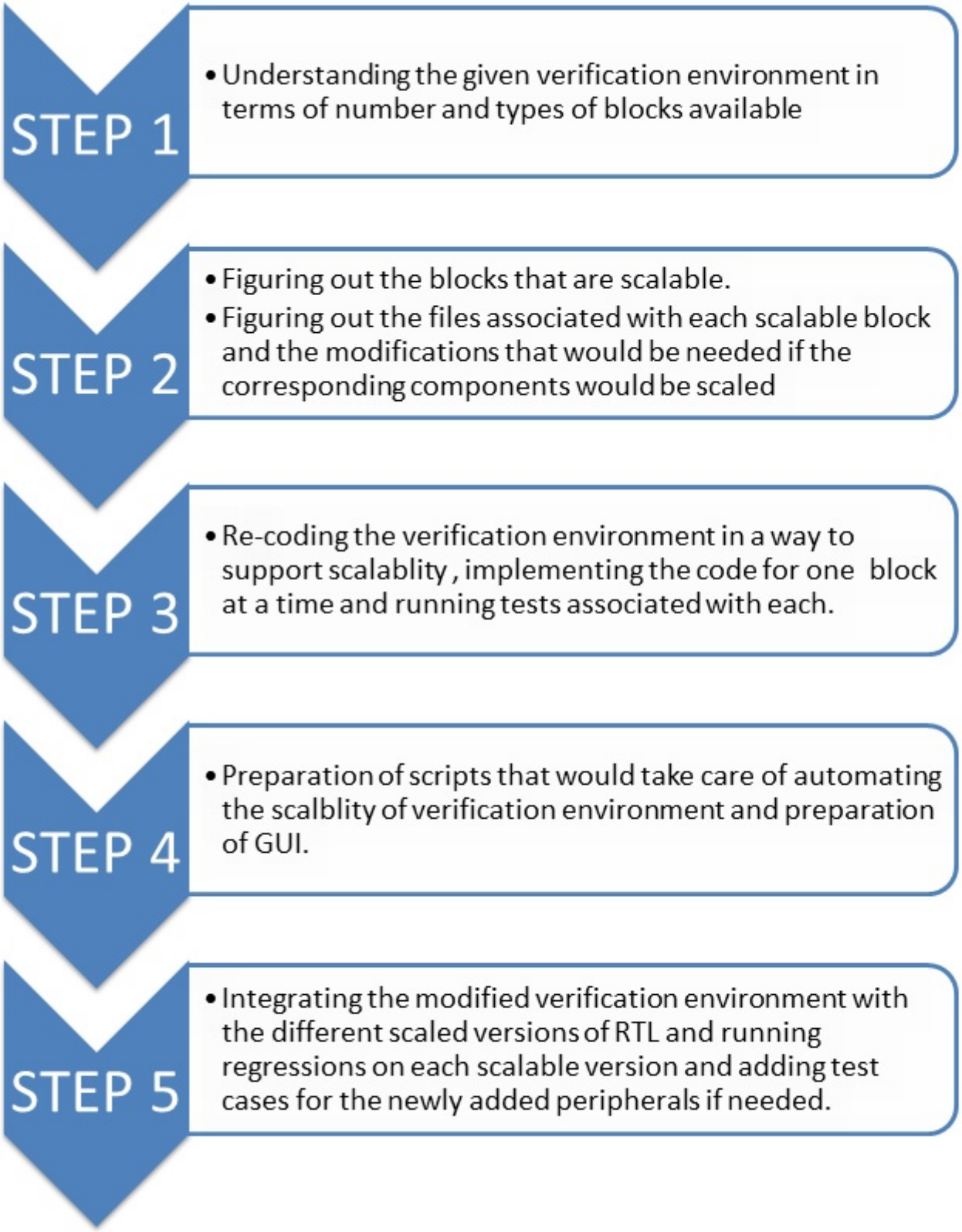
The final outcome is to evolve a tool which would take scalable parameters as inputs and would be able to scale up or down the existing verification environment. The following points were thought about:

- Exploring the availability of any tools available in the market, the output of which that can be used partially or fully.
- Make a customized tool that generates the entire verification Environment , providing a push button setup- a more useful but much time consuming process.
- Adapting a new coding style which supports scalablity including paramters, along with scripts to get a solution.

On exploring all the options we decided to go along with the third approach of adapting a new coding style and make all the necessary modifications in the existing code and take help of scripts for automating the entire process.

5.1 Project Implementation Outline

The following figure describes the way the utility is developed from the scratch. The approach followed can be used in general for developing a similar kind of utility for any SoC.


STEP 1
• Understanding the given verification environment in terms of number and types of blocks available
STEP 2
• Figuring out the blocks that are scalable.
• Figuring out the files associated with each scalable block and the modifications that would be needed if the corresponding components would be scaled
STEP 3
• Re-coding the verification environment in a way to support scalablity , implementing the code for one block at a time and running tests associated with each.
STEP 4
• Preparation of scripts that would take care of automating the scalblity of verification environment and preparation of GUI.
STEP 5
• Integrating the modified verification environment with the different scaled versions of RTL and running regressions on each scalable version and adding test cases for the newly added peripherals if needed.

5.2 Modications in the Existing Verification Environment

Once the details of verification environment has been understood we started modifying the verification environment . The main changes made can be listed as follows:

- Arraying each of the objects associated with each of the sub base envs , configuration objects, agents
- In class based files usage of dynamic arrays and simple arrays in other module based files .
- Using set config db in the base env , for defining the integer fields associated with number of scalable components , these fields can then be used using get config db in each of the respective files.
- Usage of normal for loops for signal assignments and placement of various virtual interfaces in the configuration database with the upper limit for the for loops defined by the various integer fields defined and stored in the configuration database
- Usage of generate for loops in the module based files for performing similar tasks.

Usage of UVM Configuration mechanism

The uvm_config_db class is the recommended way to access the resource database. A resource is any piece of information that is shared between more than one component or object. We use uvm_config_db::set to put something into the database and uvm_config_db::get to retrieve information from the database. The uvm_config_db class is parameterized, so the database behaves as if it is partitioned into many type-specific "mini databases." There are no limitations on the the type - it could be a class, a uvm_object, a built in type such as a bit, byte, or a virtual interface.

The set method The full signature of the set method is:
void uvm_config_db #(type T = int)::set(uvm_component cntxt , string inst_name , string field_name , T value);

- T is the type of the element being configured - usually a virtual interface or a configuration object.
- cntxt and inst_name together form a scope that is used to locate the resource within the database. The scope is formed by appending the instance name to the full hierarchical name of the context, i.e.cntxt.get_full_name(),".",inst_name.
- Field_name supplies the name of the resource
- value is the thing which is actually going to be put into the database.

In our case the syntax required is as follows :

uvm_config_db#(uvm_bitstream_t)::set(null,uvm_test_top*,parameter,value);

where the parameter may be any parameter denoting the name of the field associated with any of the scalable component and the value would be the number of components required. This parameter would be stored in the uvm configuration database and anyone below the heirarchy from uvm_test_top would be able to use it using an equivalent get config db.

The get method:

The general syntax of get config db would be :

uvm_config_db #(type T = int)::get(uvm_component cntxt , string inst_name , string field_name , ref T value);

where the various fields significances is similar to that of set config db . In our case the syntax implemented was something like:

uvm_config_db#(uvm_bitstream_t) :: get(this,,parameter name in config db,the name to which it has to be assigned)

All the files where the associated number of a particular scalable component is required would be using the get config within them.

Usage of UVM Configuration Mechanism For Placing virtual interfaces in the configuration database:

Apart from the various paramters denoting the number of scalable components , the virtual interfaces of each of the components need to be placed in the configuration databases with their appropriate scope defined.

The syntax required is as follows:

for(int i=0;i¡parametername;i++) begin
uvm_config_db #(virtual intftype)::set(uvm_root::get(),
$sformatf (uvm_test_top.env.subenv[%0d],i) ,fieldname , intfname[i]);
end

As mentioned earlier the upper limit for the for loop is defined by the parameter giving number of components which itself is stored in the configuration database.

Each virtual interface should be visible only in its corresponding sub env or agents , the second field in our case hence employs $sformatf which would set the scope of intf[0] to subenv[0] only and subenv[1] would only be able to access intf[1] and so on.

Usage of UVM configuration mechanism for placing various configuration objects:

For(int i=0;i¡no_components;i++) begin

Sub_cfg[i]=cfg_type::type_id::create(sub_cfg);

uvm_config_db#(configobjtype)::set(this,$sformatf(sub_env[%0d],i),cfg,sub_cfg[i]);

end

The configuration objects creation and making them visible in the corresponding sub envs gets taken care in a similar manner as above.

Usage of GENERATE statements in tb_top for signal assignments

In the tb_top various signals need to be assigned to each of the interfaces like clock, reset etc. The generate for loops can be used to assign these signals to similar interfaces. The syntax can be given as follows:

for(genvar i=0;i¡parametername;i++) begin
assign intf[i].signal1=signalname;
assign intf[i].signal2=signalname;
end

5.3 Scripts and GUI

Apart from the modifications in the code the role played by the scripts is summarized as follows:

Getting values from the GUI:

The parameters associated with the newer scaled up version have been provided from

the GUI. Apart from the number of scalable components the memory map associated with the slaves are also being taken.The GUI has been made using Perl/Tk. Availability of wide variety of widgets with the Tk package has been highly helpful in implementing various features of the GUI.

A snapshot of the GUI :

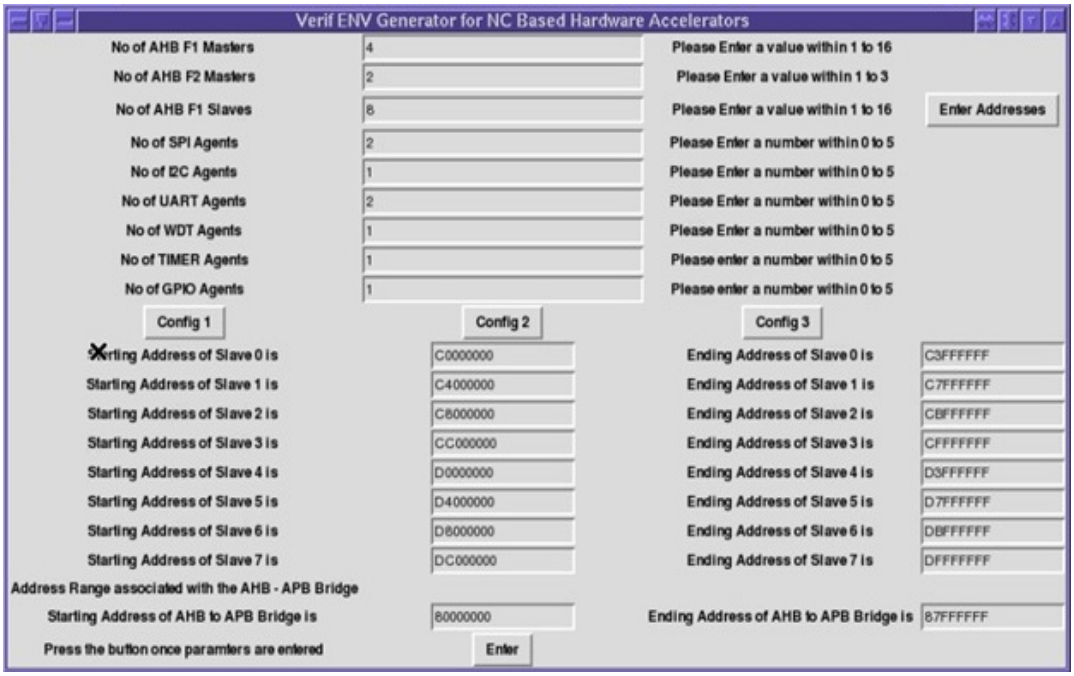


Figure 5.1: Front end GUI

Error checking mechanisms employed in the GUI:

For each entry of the various scalable components a range has been decided . If the user enters a value outside the range or the user forgets to enter a value then a new pop up window would be shown denoting an error.

Apart from detecting the invalid entries if the addresses entered are overlapping then a mechanism has been employed to report that too. It would show by itself the names of slaves whose addresses are overlapping.

Figure 5.2: Invalid entries error checking mechanism

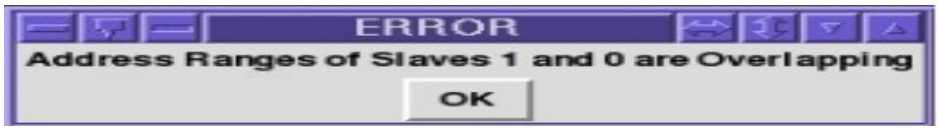


Figure 5.3: Address Overlapping error checking mechanism

Once the parameters have been entered and the user presses the enter button, a back-end script modifies the parameters in the inputfile which is taken as a source file for another script which is invoked once the parameters are written, the corresponding parameters are modified with the help of this script in the tb_top and also the modifications associated with the address map of various slave peripherals in the subsequent env files is carried out.

The memory map associated with the various slave peripherals is also entered in another file. Once the process is completed a new window pops up which shows the processes carried out:

Various buttons have been a part of it such as opening the input file, opening the memory map file and exiting the GUI.

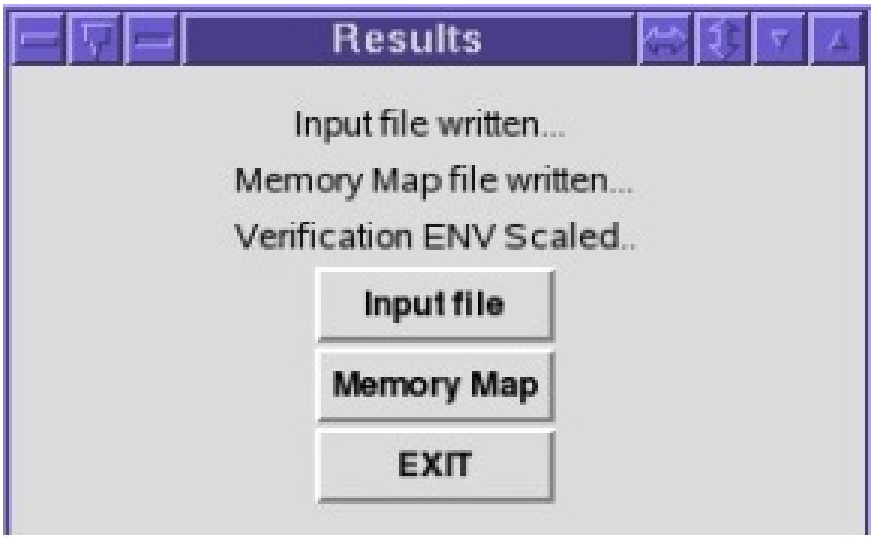


Figure 5.4: Process completion window

The backend scripts functioning:

The parameters associated with the no. of scalable components have been defined in the tb_top. The script takes these values from the input file which contains the values as entered in the GUI. It modifies these parameters in the tb_top and some other env files.

There is another script which takes care of modifying the memory map of the associated slave peripherals. If the existing verification environment has more number of slaves then the script is able to delete the extra entries and if the existing verification environment has less number of slaves then the script adds by itself the additional entries. If the number of slaves is the same the script simply modifies the address ranges as entered.

5.4 Summary

The chapter provides the details of the approach followed for designing the utility for obtaining a completely automated scalable verification environment solution. It discusses the coding guidelines that need to be followed for making a verification environment scalable. Also the part being played by Perl scripts is explained.

Chapter 6

Compatiblity for scalable data width and RAM size

6.1 Introduction

Scalability until now is targeted at number of different agents associated with AHB Fabric 1 and Fabric 2 masters/slaves as well as APB peripherals. The user had been provided with a choice of selecting the number of external master / slaves associated with AHB Fabric 1, masters associated with AHB Fabric 2, various number of peripherals associated with APB Fabric.

A new kind of scalability had to be taken to account consisting of scalable data width corresponding to AHB F2 Masters- a choice had to be provided between data width of 32 bit or 64 bit. The DMA peripheral is attached to the AHB Fabric 2 which could be either a 32 bit one or 64 bit one. Hence the need for adding the additional scalablity.Also the RAM was acting as a slave to AHB Fabric 2 Masters. The size of same was limited to 256Kb. An additional option for the user to select any size of RAM is also needed to be provided.

6.2 The modifications in the RTL

The RTLs for other configurations were uptil now tool generated . CoreAssembler a tool from Synopsys was used for the same, but for this particular configuration the RTL needed to be handcoded. The various RTL files associated with memory at various level of heirarchy were identified. The corresponding signal widths, the parameters that needed

a change were identified and correspondingly modified and a newer version of RTL with 32 bit of AHB Fabric 2 data width and 1MB of RAM size is generated.

6.3 The modifications in the Verification Environment

6.3.1 Modifications for supporting the scalable data width

The verification environment now had to support the scalable data width . The file in the verification env that corresponded to configuring of the AHB F2 Master had various options in it. These options included configuring the type of master to active/passive, selecting the address and data width of masters as well as slaves, selecting the number of master and slaves etc. The only change needed to support the AHB F2 data width was configuring the bit associated with the data width . The script was assigning the value to the bit as 32 or 64 as per the requirement from the user.

6.3.2 Modifications for supporting scalable RAM size

The verification files that had to be modified for supporting any RAM size were quite a few. First of them was associated with assigning the bits associted with total no. of RAM bytes, the no. of RAM banks.These bits needed to be changed with the change of RAM size. The same had to be taken care by a Perl script.

The other one was related to an active interface which had various tasks that performed operations with the RAM such as initializing the RAM with default values, reading RAM entries, writing to RAM. Each of these tasks had different macros associated with each of the memory banks.With increase or decrease in RAM size the number of these memory banks also would increase / decrease. The associated macros also needed to be added or removed.

The third thing was modifying the software test associated with memory. The modifications were quite minor only limiting to modifications to fields associated with the RAM block size and number of memory banks.

6.4 Simulations and Debug

Once the modifications had been done in the verification env, the test regression suite had been run.

The memory arbitration test cases were failing and the scoreboard was throwing failures. The working of the scoreboard was understood and various uvm_info statements had been added to help the debug. The mismatching transaction, the targeted memory bank to which the transaction belonged and the seed at which the test case was failing had been figured out. The mismatching transaction had been tracked manually on the waveforms at various levels of hierarchy of the rtl. The same transactions had been compared with the transactions being printed by the uvm_info statements. At first instance it seemed the scoreboard didnt support the parameterized data width. Inorder to test the same a new test case with variety of different sequences were prepared. Various sizes of transactions were sent from the AHB F2 Master to the memory. A write was performed and a simulataneous read was performed. The transactions seemed to match. Similar writes and reads were performed with a variety of different types of transactions with various burst sizes. But a match occured with all types. Hence a conclusion was made that the Scoreboard did support scalable data width.

The next step was to figure out the bugs in the RTL as the verification environment did seem to be supporting the scalable data width. The memory arbitration test case which were failing were re-run and the mismatching transaction was tracked right begining from the actual ram memory bank to the ram-top to the mcu-top. The transaction was seen at the ram-top but was found to be missing at the mcu-top. Another module of the rtl which lied in between the ram-top and mcu-top was found to be having issues. The said transaction was getting lost in the same module. The coding of the RTL was outside of the scope of our project. The same was conveyed to the concerned person and was to be fixed later. As far as the scalable verification environment supporting a scalable data width and scalable ram size, the same had been achieved.

6.5 Summary

The chapter has an explaination of the way an additional scalablity in terms of data width of AHB Fabric 2 and RAM size is provided. It describes about the modification

needed in the RTL design as well as that needed in the verification environment for the same. The end part of the chapter has a discussion about the way the debug is carried out and the simulation results that are obtained.

Chapter 7

Automation of Compilation and Elaboration

7.1 Introduction

Once the verification environment has been scaled, the test regression suite couldnt run before modifying a certain set of files specific to an Intel simulation environment relating to the order of compilation and elaboration. Inorder for specifying the order for compilation and elaboration for an Intel specific simulation environment there are a set of hdl and udf files. With the change in RTL for different configurations these files needed to be modified manually. Missing a mention of a particular file relating to a component added or removing the mention of that file from the corresponding hdl or udf files in the case if the component is removed would result into a failure of simulation. The error only would be known after the simulation is run which would sometimes taken much time. There was a need of a mechanism to modify the files with the help of script, an automatic way which would nullify the human prone error.

7.2 The working of script

The script prepared would sense the RTL and based on the available Fabric 1 and Fabric 2 masters, APB peripherals and AHB F1 slaves would modify the associated hdl files and base udf file. It would check for each of the available rtl components, their number and based on the count and availablity would make necessary modifications in the below mentioned files and also generate some new files. The files being modified are as follows:

- ia_mcu_ahb.udf
- ia_mcu_ahb.hdl
- ia_mcu_icm.hdl
- ia_mcu_spi.hdl
- ia_mcu_i2c.hdl
- ia_mcu_tmr.hdl
- ia_mcu_gpio.hdl
- ia_mcu_uart.hdl
- ia_mcu_wdt.hdl

A brief description of the above files is provided below:

- ia_mcu_ahb.udf Describes the entire rtl design as a whole . Specifies the compilation flow of the entire rtl.
- ia_mcu_ahb.hdl Describes the compilation order of the components associated with the AHB fabric 1.
- ia_mcu_icm.hdl Describes the compilation order of the components associated with slaves attached to AHB fabric 1.
- ia_mcu_spi.hdl Describes the compilation order of the files associated with the spi APB peripheral.
- ia_mcu_i2c.hdl Describes the compilation order of the files associated with the i2c APB peripheral.
- ia_mcu_tmr.hdl Describes the compilation order of the files associated with the timers APB peripheral.
- ia_mcu_gpio.hdl Describes the compilation order of the files associated with the gpio APB peripheral.

- ia_mcu_uart.hdl Describes the compilation order of the files associated with the uart APB peripheral.
- ia_mcu_wdt.hdl Describes the compilation order of the files associated with the wdt APB peripheral.

7.3 Outcome

The addition or deletion of any of the peripherals is being taken care by the script itself and the corresponding files get modified accordingly. The script is tested for different configurations as well. Once the main script is run, and then this particular script, the verification environment is ready to run the test regression suite. This script provides another time saving of two hours per configuration.

7.4 Summary

The chapter has a description of the additional script that automates the process of compilation and elaboration. For specifying the order of compilation and elaboration certain set of files specific to a Intel Simulation environment need to be modified. The chapter provides an overview of the different types of files that need to be modified and also the way the script modifies these files is discussed.

Chapter 8

Conclusion and Future Work

The project implemented is able to provide any scaled version of verification environment as per the need. The overall time spent for creating a verification environment for different scalble versions of the assigned SOC has been reduced and thereby speeding up the process of verification.The project has been tested for different scalable versions of the SOC and is working fine.The scalablity addresed at the various levels such as no. of AHB fabric masters/slaves, various number of APB peripherals, scalable data width of AHB fabric master and scalable RAM size.

Future Work

The work aimed at delivering a push button solution for a scalable verification environment as per the need. A similar utility can be designed for generating a scalable RTL and both can be integrated together. The SoC currently supports multi layer AHB Lite fabric in which there can exist only a single master per layer, an additional support can be provided for additional multiple masters per layer.The utility has been designed for a specific IA based SoC , a more generic tool can be made which is able to provide a similar utility for any of IA or perhaps any kind of SoC irrespective of IA being a part of it.

Bibliography

[1] Universal Verification Methodology (UVM) 1.1 User's Guide from accellera.

[2] Randal L. Schwartz, Tom Phoenix, Learning Perl, 3rd edition, O' Reilly publishers

[3] Steve Lidie and Nancy Walsh, Mastering Perl/Tk Graphical User Interfaces in Perl, O' Reilly publishers.

[4] Mark Glasser, Open Verification Methodology Cookbook, Springer Publications.

[5] IEEE Std 1800-2012 - Standard for SystemVerilog-Unified Hardware Design, Specification, and Verification Language

[6] http://blogs.mentor.com/verificationhorizons/blog/2013/07/15/part-5-the-2012-wilson-research-group-functional-verification-study/

[7] www.verificationacademy.com

[8] AMBA 3 AHB-Lite Protocol v1.0 Specification

[9] Multi-Layer AHB Overview , ARM.

[10] AMBA Specification (REV 2.0), ARM.